Technical Report

UCAM-CL-TR-761
ISSN 1476-2986

Number 761

UNIVERSITY OF
CAMBRIDGE

Computer Laboratory

Programming networks of vehicles

Jonathan J. Davies

November 2009

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted September 2008 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Churchill College.

ISSN 1476-2986

Programming networks of vehicles

Jonathan J. Davies

Abstract. As computers become smaller in size and advances in communications technology are made, we hypothesise that a new range of applications involving computing in road vehicles will emerge. These applications may be enabled by the future arrival of general-purpose computing platforms in vehicles. Many of these applications will involve the collection, processing and distribution of data sampled by sensors on large numbers of vehicles. This dissertation is primarily concerned with addressing how these applications can be designed and implemented by programmers.

We explore how a vehicular sensor platform may be built and how data from a variety of sensors can be sampled and stored. Applications exploiting such platforms will infer higher-level information from the raw sensor data collected. We present the design and implementation of one such application which involves processing vehicles' location histories into an up-to-date road map.

Our experience shows that there is a problem with programming this kind of application: the number of vehicles and the nature of computational infrastructure available are not known until the application is executed. By comparison, existing approaches to programming applications in wireless sensor networks tend to assume that the nature of the network architecture is known at design-time. This is not an appropriate assumption to make in vehicular sensor networks. Instead, this dissertation proposes that the functionality of applications is designed and implemented at a higher level and the problem of deciding how and where its components are to be executed is left to a compiler. We call this 'late physical binding'.

This approach brings the benefit that applications can be automatically adapted and optimised for execution in a wide range of environments. We describe a suite of transformations which can change the order in which components of the program are executed whilst preserving its semantic integrity. These transformations may affect several of the application's characteristics such as its execution time or energy consumption.

The practical utility of this approach is demonstrated through a novel programming language based on Java. Two examples of diverse applications are presented which demonstrate that the language and compiler can be used to create non-trivial applications. Performance measurements show that the compiler can introduce parallelism to make more efficient use of resources and reduce an application's execution time. One of the applications belongs to a class of distributed systems beyond merely processing vehicular sensor data, suggesting that the late physical binding paradigm has broader application to other areas of distributed computing.

Contents

Preface

Acknowledgments. I have been supported by many people throughout the course of the studies which have culminated in this dissertation. I am indebted to my supervisor, Andy Hopper, for generously arranging for financial support. Moreover, he has been the inspiration behind much of this work, providing many thought-provoking suggestions and ideas. Many thanks are due to Alastair Beresford for his day-to-day assistance, advice and encouragement, many times going beyond the call of duty to provide aid; much of the content of this dissertation has been discussed with him at length. I would also like to thank Alan Mycroft for allowing me access to his valuable expertise which has particularly influenced the latter part of the work.

Many others in the Computer Laboratory have provided assistance of various forms, from technical help through to proof-reading and moral support. These include David Cottingham, Andrew Rice, Brian Jones, Samuel Kounev, Ripduman Sohan, Richard Gibbens, Mbou Eyole-Monono, Robert Harle, Tom Craig, Marcelo Pias, Matthew Parkinson, Joseph Newman and John Fawcett. Thanks are due to Louise Driffill for proof-reading. Finally, I would like to thank my examiners, Ken Moody and Tim Harris, for their interest and helpful suggestions.

Some of the work contained in this dissertation originated from or was performed jointly with others. The list of applications involving in-vehicle computing (Section 1.2) was devised jointly with David Cottingham. The vehicular sensor platform (Chapter 3) was designed and built in collaboration with Brian Jones and David Cottingham. In particular, the communications infrastructure (Section 3.2.4) was set up by David and the power control (Section 3.2.5) implemented by Brian. Figure 3.10 was produced with the help of Andrew Rice. Some elements of the map generation algorithm (Section 4.1.2) were suggested by Alastair Beresford; their implementation is my own, with the exception of the contour follower, implemented by Andrew Rice. Richard Gibbens assisted with the statistics in Section 4.1.4.1. Keith Farkas made some suggestions about an earlier form of the content which contributed towards Chapter 4. Much of Chapter 5 evolved after several lengthy discussions with Alastair Beresford and Alan Mycroft. The invention of split tasks

(Section 5.5.1) and the exponentiation example (Section 5.5.7.2) are due to Alan Mycroft. Samuel Kounev suggested considering quasi-static applications (Section 5.4.1).

Stylistic conventions. As is commonplace in academic publications, the use of the plural form of the subjective first-person pronoun—*we*—is adopted although it refers to a singular author. The exceptions to this convention are highlighted above, where some aspects of the work described have been done in conjunction with other researchers.

The pronouns *that* and *which* are used interchangeably to introduce restrictive clauses. Non-restrictive clauses always begin with *which* and use commas to delimit the clause.

Citations are generally placed at the end of a sentence which introduces or describes the main theme of the academic work referenced. When this gives rise to ambiguity, citations are placed adjacent to the word or phrase which most closely identifies the work. When reference is made to a part of a work and not the work as a whole (such as an individual statement contained in it), the page number or section number is included with the citation. Where a reference pertains to the items in a list, the citation is made after the paragraph preceding the list.

Publications

Some of the contributions presented in this dissertation have appeared in the following publications:

1. Alastair R. Beresford, Jonathan J. Davies, and Robert K. Harle. Privacy-sensitive congestion charging. In *Proceedings of the 14th International Workshop on Security Protocols (SPW 2006)*, to appear in *LNCS*, Cambridge, UK, March 2006.

2. Jonathan J. Davies, David N. Cottingham, and Brian D. Jones. A sensor platform for sentient transportation research. In Paul Havinga, Maria Lijding, Nirvana Meratnia, and Maarten Wegdam, editors, *1st European Conference on Smart Sensing and Context (EuroSSC 2006)*, volume 4272 of *LNCS*, pages 226–229, Enschede, Netherlands, October 2006. Springer.

3. Jonathan J. Davies, Alastair R. Beresford, and Andy Hopper. Scalable, distributed, real-time map generation. *IEEE Pervasive Computing*, 5(4):47–54, Oct–Dec 2006.

4. David N. Cottingham and Jonathan J. Davies. A vision for wireless access on the road network. In *Proceedings of the 4th International Workshop on Intelligent Transportation (WIT 2007)*, pages 25–30, Hamburg, Germany, March 2007. Technische Universität Hamburg-Harburg.

5. Jonathan J. Davies and Alastair R. Beresford. Scalable, inter-vehicular applications. In *On the Move to Meaningful Internet Systems 2007: OTM 2007 Workshops (Part II)*, volume 4806 of *LNCS*, pages 876–885, Vilamoura, Portugal, November 2007. Springer.

6. Jonathan J. Davies, Alastair R. Beresford, and Alan Mycroft. Language-based optimisation of sensor-driven distributed computing applications. In José Luiz Fiadeiro and Paola Inverardi, editors, *11th International Conference on Fundamental Approaches to Software Engineering (FASE 2008)*, volume 4961 of *LNCS*, pages 407–422, Budapest, Hungary, March 2008. Springer.

CHAPTER 1

Introduction

Over recent decades, computer technology has impacted on many areas of society. These innovations have brought increases in efficiency, productivity, safety and many other benefits. Transportation has enjoyed many aspects of this revolution, with innovations ranging from electronic stability control in vehicles to urban traffic control systems and electronic toll collection systems on the road network.

Recent trends in communication technology mean that a further range of applications will become feasible in the near future. In particular, applications involving communication amongst large numbers of vehicles across a wide geographic scale are likely to be implemented. The primary focus of this dissertation is to examine how these applications can be designed, programmed and deployed.

We begin to introduce this work by speculating about the nature of these inter-vehicular applications in Section 1.2, then highlight some of the challenges in implementing them in Section 1.3. Finally, in Section 1.4 we describe the approach that is taken in this dissertation to aim towards achieving these goals.

1.1 In-vehicle computing

Traditionally, computing has been primarily applied to vehicular technology for safety-related purposes. After these applications, the next beneficiaries of computer technology in the vehicle industry have been entertainment-oriented applications.

Beyond these areas, there have been very few examples of computing in vehicles. However, as on-board processing and communications become cheaper and less invasive, more applications will become possible.

At present, vehicle manufacturers produce proprietary applications which make use of built-in hardware. Other vendors supply after-market products that can be added to the vehicle. However, it is rare for these products to integrate seamlessly with each other and with the vehicle's own computer software and hardware.

As a result, applications that could share common hardware tend to be deployed in isolation from each other. This inevitably causes the need for lots of separate wires and circuit boards, which is expensive. Instead, it is preferable for applications to share common hardware to avoid unnecessary redundancy. For example, GPS units are required by a range of in-vehicle devices: navigation units, black-box devices used in pay-as-you-drive insurance, and certain implementations of road-user charging. Provided that the vehicle can supply a GPS unit of sufficient accuracy to satisfy all these applications' demands, using this single device would be a preferable solution.

It is conceivable that in future vehicles will contain general-purpose sensing and computing platforms to avoid this situation. This may be spurred by an increasing demand for a variety of applications involving in-vehicle computing.

1.2 Applications

We predict a variety of such applications, some of which are already emerging: [49]

Entertainment. Entertainment applications will expand beyond listening to music and watching films to those which involve interaction with the World-Wide Web and receipt of streaming media. The AMI-C organisation (Automotive Multimedia Interface Collaboration) has published standards for automotive interfaces, which enable a wide variety of envisioned applications that are primarily entertainment-oriented [5].

Mobile commerce. These applications involve purchasing products and performing other business transactions whilst on the move [235].

Location-based services. If a vehicle is aware of its geographic position, content relating to that area can be delivered. This may include local weather or traffic data, or advertisements from local businesses.

Remote operation. These applications involve the operation of devices in a remote location such as the home or the office from a moving vehicle, such as switching on the heating in advance of arriving at a destination.

Asset tracking. A delivery or haulage vehicle can report its position and its current inventory.

Congestion information. An in-vehicle navigation unit could receive a broadcast of current traffic conditions determined for example by aggregating sensor data gathered from in-road vehicle sensors. Further, the navigation unit could proactively query a remote congestion information service regarding the roads that it considers to incorporate into a route. Such a service could collect data known as ‘floating car data' which consists of detailed movement data from vehicles [55]. This data can be processed to provide accurate estimates of journey times on particular roads.

Real-time weather. Similarly, vehicles could download current weather observations and forecasts for the local region. In addition, the network could gather data from vehicles which contain meteorological sensors and redistribute the aggregated data back to interested vehicles. There is a huge market for short-term weather forecasts which would benefit from high-resolution knowledge of current conditions.

Road hazard detection. Potential hazards on the roads could be detected if data from many vehicles' braking systems is gathered [94, §3.2]. When a substantial number of vehicles are found to brake sharply at a particular location, it could be marked as a potential hazard. Again, this could be a service which vehicles query and from which they receive notifications of upcoming hazard-spots.

Map generation. Similarly, the location histories of large numbers of vehicles could be combined and aggregated to provide updates to digital road maps in real-time. For example, new roads opening or road closures could be detected in this way and the information delivered back to the vehicles [54]. This application is the subject of Chapter 4.

Slot booking. Motorway slot-reservation systems, slip-road metering systems [197] and systems which co-ordinate the efficient flow of traffic through an intersection can be implemented by vehicles communicating with a known Internet host to negotiate timing and payment.

Fleet management. Organisations owning a number of vehicles need to be able to manage them centrally, perhaps so that their future movements can be planned and their routes optimised.

Gaming. Occupants of vehicles can play games—perhaps location-aware games [171]—with occupants of other vehicles or non-mobile participants.

Road user charging. Many suggested implementations of electronic toll collection or congestion charging schemes involve the transmission of location data to a governmental organisation. Schemes involving dynamic pricing require the in-vehicle unit to keep track of current prices and pay on behalf of the driver. Alternatively, a peer-to-peer implementation of congestion charging involving communication between vehicles that preserves the privacy of the users has been proposed [101].

Intersection collision avoidance. This class of safety applications involves the self-organising co-ordination of the movement of traffic, such as negotiation between vehicles approaching a road junction [59].

Accident notification. In-vehicle systems that detect collisions and automatically notify the emergency services of the location and the nature of the collision enable faster responses.

Journey scheduling. A personal device could proactively suggest not only routes by which to travel to a destination, but also the modes of transport to use. These recommendations could be derived from the aggregated information it receives from a journey-time and timetable service. For example, a driver travelling towards a city centre could be advised that it is more optimal, in terms of both money and time, to park on the outskirts of the city and take a bus to the centre.

1.3 Challenges

Some of the applications described above involve the co-ordination of computation to process sensor data amongst large numbers of vehicles. This can make the design and implementation of these applications challenging. Specifically, there are four particular challenges facing application designers:

1. Vehicles need a means of wireless communication with other vehicles and with the Internet.
2. The mobility of vehicles means that the characteristics of the communication links will vary over time. Applications must be able to adapt to changes in connectivity.
3. Applications will have goals that relate to the entire network rather than to specific individual vehicles. This means that the set of participating vehicles will not necessarily be known before the application is executed. Applications must therefore be able to adapt to whichever computational resources are available to execute it at run-time. This demands a declarative approach to programming, in which *what* needs to be executed is described rather than *how* and *where* it is executed.
4. A network including computers on-board vehicles will be heterogeneous. Participating computers will differ in terms of their levels of computational, storage, energy and communication resources. Applications should make the most efficient use possible of these resources whilst ensuring that those in limited supply are not exhausted.

1.4 Dissertation outline

Of these challenges, this dissertation will focus mainly around addressing the third and fourth challenges. We will touch briefly upon the former two challenges but leave these for future work to address.

Chapter 2 lays the foundations of this dissertation by exploring related research. A network of vehicles can be treated as a sensor network or as a classical distributed system; approaches to programming both of these kinds of system are described.

Chapter 3 describes the design and implementation of a vehicular sensor platform, highlighting the differing nature of data from a variety of sensors and how it can be sampled and stored.

Chapter 4 describes a novel application in which data sensed on multiple vehicles is collected and processed. The chosen application is the map generation application suggested above. This exemplifies the challenges described in Section 1.3 by questioning the most appropriate architecture for executing this application. Existing approaches to programming applications in wireless sensor networks tend to assume that the nature of the architecture is known at design-time; this is not an appropriate assumption to make in vehicular sensor networks.

Chapter 5 describes an approach to addressing the challenges, without needing to make this assumption, through the paradigm of 'late physical binding'. Designing applications according to this paradigm involves describing their computation at a higher level in terms of a graph of inter-related tasks. The decision about how and where the application's components are to be executed, called automatic task assignment, is then made by a compiler that decides where each task is best executed. This approach brings the benefit that applications can be automatically adapted and optimised for execution in a wide range of environments. The chapter describes how a task graph can be transformed to change the order in which components of the application are executed, whilst preserving its semantic integrity, in order to optimise the task placement.

Chapter 6 describes a concrete programming language and a compiler which together implement the late physical binding paradigm.

Finally, Chapter 7 demonstrates the use of the paradigm and language to design and implement two different kinds of application, showing the value of late physical binding. Performance measurements show that the compiler can introduce parallelism to make more efficient use of resources and reduce an application's execution time.

CHAPTER 2

Background

This chapter presents the foundations on which the work in the forthcoming chapters is built.

We will begin by examining emerging trends in computing in vehicles and consider the nature of the inter-vehicular computing environment in Section 2.1. The inter-vehicular computing environment can be viewed as a distributed system: a parallel system of communicating devices which collaborate to achieve a common goal. Hence, Section 2.2 tours a variety of approaches to abstracting distributed computation, focussing particularly on the impact these have on the programmer.

As well as being viewed as a form of distributed system, many inter-vehicular computing applications can be thought of as sensor networks as they are driven by sensor data collected from vehicles, perhaps processed by other vehicles. In the field of sensor networks, much work has been done in devising techniques by which sensor data can be sampled, shared, processed and disseminated. Section 2.3 surveys models for programming sensor networks in the light of lessons learned from distributed computing in general.

2.1 Computing in vehicles

The term *ubiquitous computing* [243] was coined as an area of Computer Science research in which computers are deployed pervasively throughout everyday environments. Various related terms have been employed—sentient computing [114], pervasive computing, ambient computing, and more—to indicate the goal of computers shrinking in size and becoming embedded into the fabric of our environment.

As this vision is increasingly realised, our environment becomes populated with intelligent devices. Smart buildings, smart desks, and other 'smart' nouns become a reality.

The field of transportation has not evaded electronic intelligence. *Intelligent Transport Systems* (ITS) is a term applied to advances in this area, covering a variety of fields, including integrated transportation systems, urban traffic control systems, and driver assistance systems.

If the vision of ubiquitous computing is to be attained, all vehicles, all traffic lights, all road lanes and all road signs will eventually become addressable nodes on the Internet. In this section, we explore what it might mean for a vehicle to be a participant in this environment. We will look at what a vehicle might provide to other parties, and how it could benefit from being connected to, rather than isolated from, surrounding nodes.

2.1.1 Data processing in vehicles

A modern vehicle contains many microprocessors governing various aspects of the vehicle's operation, from the anti-lock braking system through to the CD player unit. A vehicle's engine management unit (EMU) is responsible for monitoring data from sensors attached to the components of the engine and controlling actuators such as that which controls the fuel pump.

Some systems installed by the manufacturer of the vehicle use general purpose CPUs running commodity operating systems. For example, BMW's iDrive system, a software interface to allow the driver to interact with the vehicle, uses the VxWorks real-time operating system.

There are many instances of after-market, third-party devices being used in vehicles. Off-the-shelf navigation units, which are becoming increasingly popular, employ similar technology. Pay-as-you-drive insurance schemes [157], pioneered by Norwich Union in the UK, involve a 'black box' inside the vehicle that records parameters relating to the vehicle's whereabouts and communicates over a cellular network to report back to the insurance company so that a premium can be computed. Also, several proposed schemes for implementing road-user charging such as the HGV tolling scheme in Germany [130] involve an in-vehicle device which communicates with roadside interrogators.

Rather than all of these systems—and many others—being installed in separate black boxes, each with their own CPUs, it is conceivable that, in the future, vehicle manufacturers could construct vehicles fitted with general-purpose CPUs that are shared by all such applications.

Due to the highly competitive nature of the vehicle market, manufacturers are keen to keep both the fuel consumption and the mass of its vehicles as low as possible. Therefore, if general-purpose CPUs are to be provided by manufacturers, it must be the case that they bring utility to the driver which outweighs any increase in the vehicle's fuel consumption or mass. In some cases, the overall mass may decrease if a CPU is shared between applications. Either way, in the future, embedded computing in vehicles will become increasingly feasible if Moore's Law [180] continues to hold.

There may also be other business reasons for manufacturers to incorporate general-purpose CPUs into vehicles. For example, they may be able to charge customers a substantial premium for an integrated system compared to an after-market solution, or they may be able to produce future revenue through manufacturer-supplied subscription-based applications.

2.1.2 Sensors

Modern vehicles contain a vast array of sensors. Some monitor various aspects of the engine's operation, such as thermometers and fuel flow rate sensors. Others participate in driver-level applications, such as sensors which detect whether a door is closed, and rear-mounted range-finding sensors which detect the distance to the nearest object behind the vehicle.

Sensors may vary widely in terms of the frequency of sampling and the amount of data they produce. For example, a thermometer may output a single byte and be sampled every few seconds; a video camera to assist with reversing may output several megabytes per second.

In the past, wiring was added to vehicles for each new sensor added. However, as vehicles gained increasing numbers of sensors, this added considerable weight, consumed a significant amount of space and made adherence to reliability standards difficult [146, p88]. This led to the adoption of a bus-based approach, with wiring shared between sensors. In the mid-1980s, the controller area network (CAN) was developed, derivatives of which are still in widespread use in modern vehicles [146]. Modern vehicles may contain multiple CAN buses: perhaps a low-speed bus for the comfort electronics and a high-speed bus for real-time systems involving engine management.

2.1.3 Communications

In order to share data (processed or unprocessed) with other vehicles, and to obtain data from other nodes on the Internet (which may themselves be vehicles), extra-vehicular wireless communications are required.[1]

Historically, the only form of extra-vehicle communication in the electro-magnetic spectrum was the receipt of broadcast radio signals via an analogue tuner. Later, this system was augmented by small amounts of digital information being broadcast on public radio channels via the Radio Data System (RDS).

Using near-ubiquitous cellular networks such as GSM, GPRS and UMTS, vehicles are able to connect to the Internet in the same way as mobile telephones. Recently, with systems such as General Motors' OnStar system, two-way communication to and from vehicles has become available. OnStar provides a remote door-unlocking facility and an automatic crash detection system which notifies the emergency services of the vehicle's location.

Details about current and predicted future trends in communication technologies are contained in Appendix A.

[1]Due to the mobility of vehicles, communications intuitively seem to be necessarily wireless. However, a wired system is conceivable, involving data-lines strung above roads like power-lines for trams. But this is deemed to be impractical to implement so will not be considered further.

2.1.4 Vehicular networks

A vehicular network can be centralised or decentralised. A centralised network uses a ubiquitous wireless communication technology such as GSM or UMTS, and vehicles have globally unique identifiers. On the other hand, a decentralised network uses local communication between nearby nodes. As vehicles move around, they may move into and come out of range of other nodes. Hence this variety of network is largely ad-hoc, and its topology is highly variable and may be somewhat unpredictable.

Vehicular networks have particular characteristics that distinguish them from other classes of network:

- Despite the fast speeds of vehicles, their movements are constrained and their mobility is hence somewhat predictable. Vehicles can only use roads, which occupy a comparatively small proportion of the surface of the Earth.
- Due to the nodes' mobility, energy resources are necessarily constrained. However, energy is in more plentiful supply than in battery-powered devices. Vehicles have a plentiful supply of fuel, and the consumption of energy due to computing and communications equipment is likely to be significantly lower than that consumed as a result of the vehicle's other operations.
- Vehicles may be assumed to have an accurate knowledge of their position [188, §1], by using a global, outdoor positioning system such as GPS, and may also possess a road map of the local area [158, §4].

We will consider vehicular ad-hoc networks in Section 2.1.4.1, contrasting them with other classes of ad-hoc network, and vehicular sensor networks in Section 2.1.4.2, contrasing them with other classes of sensor network.

2.1.4.1 Vehicular ad-hoc networks

The term 'mobile ad-hoc network', or MANET, is used to describe networks involving mobile nodes which do not rely on the availability of a ubiquitous wireless network to communicate. MANETs are usually characterised by nodes having limited computational resources (processor and storage). The vehicular ad-hoc network, or VANET, is a particular class of MANET in which the mobile nodes are vehicles travelling on roads.

The characteristics of vehicular networks outlined above have important implications for VANETs:

- Networks are highly dynamic due to the mobility of vehicles. Vehicles passing each other on a motorway may only be within communication range for a matter of seconds, so communication links between vehicles are frequently established and broken.
- The traffic density on roads varies significantly throughout the day, perhaps by a few orders of magnitude. Hence, at off-peak times, vehicles may be disconnected from other vehicles if none are within range [186, p7].

Blum et al. have attempted to quantify the distinctions in mobility in VANETs, compared to general MANETs, through simulation [30]. They have established that vehicular networks experience very rapid changes in topology due to the high relative speed of vehicles. Furthermore, even links between vehicles travelling in the same direction are short-lived: for transmissions of 500 ft range, the links last about one minute on average. In addition, the inter-vehicular networks were found to be subject to frequent fragmentation, in which chunks of the network become isolated from each other.

These factors make the implementation of useful VANETs difficult [188]. Due to the highly dynamic mobility of vehicles, there is no guarantee that the vehicles nearby in one instant will be nearby in the next. Hence, reliably routing a message through a network is a challenge. Vehicles passing on motorways can pass at up to 140 mph, whilst the density of vehicles may vary from as little as one vehicle per kilometre of road to five hundred vehicles per kilometre. In low density situations, a wide transmission range is desirable but in high density situations it would cause too much contention.

Point-to-point routing. Point-to-point communication is required in applications which involve collaboration between vehicles which are not necessarily collocated. There have been many protocols suggested for routing messages between nodes in MANETs. They can be classified into three categories: proactive, reactive and position-based protocols: [83, §4.1]

Proactive protocols employ classical routing strategies such as distance-vector routing or link-state routing. These protocols are unsatisfactory for use in VANETs since they maintain state about paths, expecting this information to stay fairly constant over time, which in a vehicular network cannot be assumed.

Reactive protocols create new routes for each message sent, so do not need to store state about the paths which are not currently in use. Dynamic Source Routing (DSR) [125] is a *source routing* protocol, in which a message's sender (the source) specifies the entire path to the destination in the message's header. In contrast, the ad-hoc on-demand distance vector algorithm (AODV) [198] is an example of *destination routing*, which adopts a hop-by-hop approach. Intermediate nodes use a local look-up table to determine which node to forward a message to. In a vehicular network, the destination routing approach is likely to fare better than source routing as the dynamic nature of communication links may render a specified path invalid before the message has reached its destination [248, §2.3].

Position-based protocols assume that nodes have knowledge of their location, which is periodically broadcast to neighbouring nodes and registered with a centralised location service [103]. Routing can then be stateless, performed solely based upon the positional displacement of neighbours relative to the destination node whose location was looked up in the location service. A well-known example of a position-based protocol is Greedy Perimeter Stateless Routing (GPSR) [135]. This protocol attempts to move a message greedily towards its destination. When greedy routing can get the message no further, it is routed around the perimeter of the region between that point and the destination.

Most routing protocols are commonly evaluated against a random-waypoint model. However, this is inappropriate for VANETs where mobility is far more tightly constrained. Füßler et al. have instead compared DSR and GPSR by simulating the movements of vehicles in a traffic simulator [83]. Their results show that the position-based approach performs best for communications spanning more than a few hops.

One of the assumptions about VANETs mentioned above is that vehicles may be assumed to know their locations and the local road topology; this makes position-based protocols appear most promising for routing in VANETs. This has given rise to a number of variants of position-based routing protocols tailored specifically to the characteristics of vehicular networks.

Due to the characteristic constraints of vehicle mobility, routing can be optimised by knowledge of such movement patterns and maps of the local roads. The Anchor-Based Street and Traffic Aware Routing (A-STAR) protocol [158] exploits this knowledge for position-based routing within cities using urban bus route information and knowledge of street topology.

Similarly, Vehicle-Assisted Data Delivery (VADD) is a strategy to facilitate a delay-tolerant query and response mechanism using multi-hop routing which takes account of vehicles' mobility patterns [259]. This protocol is based upon the *carry-and-forward* principle, where nodes carry the data when routes do not exist and forward the data when a new node is within range. Vehicles are assumed to have a road map which incorporates information about traffic densities and traffic signal schedules. This knowledge can be used to determine the best road along which to route a data packet.

The CarNet project observed that the use of a centralised location service for position-based routing can hinder easy deployment and scalability [182]. Rather than the location service depending on the existence of some fixed infrastructure, they propose a distributed location service [151].

Message dissemination. In addition to routing for point-to-point communication, many vehicular applications require the ability to disseminate messages within a group of nodes. For example, vehicle platooning [153] benefits from the participating vehicles regularly sharing acceleration, speed and position data [52]. Platooning involves a group of vehicles travelling in close proximity at high speeds; this is desirable from the point of view of maximising road utilisation and fuel economy. A particular challenge of this type of communication is the organisation of access to the radio channel.

Briesemeister et al. have proposed a protocol for disseminating a message about a road hazard to surrounding vehicles [37]. Their proposed system was simulated and results suggested that vehicles as far away as 5 km from the source of the message could receive the message in less than a second. Wu et al. have simulated message propagation on a particular road to investigate the effectiveness of multi-hop message passing to disseminate information [249]. Their results show that the density of traffic is critical to the distance and speed with which a message can travel.

MDDV is a system for disseminating a message to all nodes within a target region [248]. It combines opportunistic forwarding, trajectory-based forwarding and geographic forwarding. Unlike other geographic approaches, knowledge of the locations of other vehicles is not assumed.

In the Segment-Oriented Data Abstraction and Dissemination system (SODAD), data is associated with a 'segment' of road, which is the region of road in which it was generated [246]. This per-segment information is disseminated through local broadcasts and store-and-forward propagation. This system is demonstrated in the SOTIS traffic information application, whereby data about the traffic conditions in a region is propagated to vehicles outside the region [247].

The RT-STEAM middleware provides guaranteed real-time message propagation without relying on a centralised event broker or look-up service [118]. Real-time guarantees are achieved via a *space-elastic* model which maintains a dynamic proximity bound which allows changes to membership and topology.

Chisalita et al. have proposed an approach to data-sharing using inter-vehicular and vehicle-to-roadside communication adopting techniques from peer-to-peer networking [47]. It is claimed that the use of this paradigm in VANETs can facilitate the development of self-organising, fault tolerant and scalable vehicular networks.

2.1.4.2 Vehicular sensor networks

We now consider the form of vehicular network which exists for the purpose of sensing. Appendix B contains a general introduction to the field of sensor networks. As well as the lower constraints on resources and the nuances of vehicle mobility, a vehicular sensor network differs in several respects from a traditional, non-mobile wireless sensor network:

- The high potential speed of vehicles means that sensing may take place much more frequently than with static sensors. If the sensors measure variables which vary with position then high sampling rates are necessary in order to build an accurate picture of the variable. This implies that the volumes of data collected by vehicular sensor networks may be far larger than for static networks.
- A vehicular sensor network may cover a vast geographic area and may contain several orders of magnitude more participants than typical static sensor networks.
- Furthermore, in a vehicle, there may be lower constraints on available physical space and energy consumption, so larger processors and storage may be acceptable.

These characteristics mean that protocols and architectures designed for traditional sensor networks are not necessarily directly applicable to vehicular sensor networks [145, p53]. Techniques used in traditional wireless sensor networks, such as those described in Appendix B, tended to assume that the devices are motionless, low-power devices in networks containing no more than tens or hundreds of nodes. These assumptions often lead to proposals for protocols which flood the entire network with metadata; these are not appropriate for larger networks.

Although constraints on available communications bandwidth may be far more relaxed in a vehicular network, it is still important to employ protocols which keep communications to a minimum and to keep a tight rein on nodes' energy consumption. When the communication medium is broadcast-oriented, as with wireless communications, there is a risk of overloading the network and rendering it unusable by other nodes. A single overpowered node could overload the whole cell or the entire network [185, §3.5].

Similarly, whilst storage space may be less constrained, it is still finite. Xu et al. address the problem of deciding which data to keep and disseminate and which to discard given limited storage capacity in a vehicular sensor network [250]. Spatio-temporal data becomes increasingly worthless as the spatial displacement and age increase. For example, information about a parking space in one city is of no relevance to drivers in another city. Similarly, the data is worthless even in the original city on the next day. The decision about which data to store can be determined by using a 'relevance function', which scores spatio-temporal data in terms of distance and age.

To date, there has been a limited amount of research into vehicular sensor networks, and very few actual deployments. The extant research falls largely into two main categories: environment monitoring and traffic information systems.

Environment monitoring. There is a range of applications involving vehicles acting as sensor nodes monitoring the environment they move within. This may involve sampling pollution in the atmosphere or monitoring the surface of the road [68].

MobEyes is a system which provides data dissemination facilities for a vehicular sensor network [145]. The target application domain is urban monitoring, which has a particular use in post-processing for the detection of criminal activity. Vehicles proactively gather information about their locality and generate summaries of the data. These summaries are opportunistically disseminated with nearby vehicles, along with context information. The style of diffusion is specified by the application; for example, data may passively diffuse by no more than k hops from its origin. Experimental results have shown that small values of k are adequate even for moderately widespread diffusion of a datum. MobEyes highlights a departure in vehicular sensor networks from traditional sensor networks: processing the data and sending it to a distinguished sink is deemed infeasible due to the large number of participants and large volume of data. Instead, data is stored within the network, which can be viewed as a large, decentralised, mobile store, which may be queried at a later point in time.

The CarTel system takes a different approach [119]. Sensor data collected from vehicles is sent to a central 'portal' where it is stored in a database. There are various means by which data can travel to the portal. Vehicles opportunistically exploit open WiFi connections as they encounter them to communicate directly to the portal over the Internet. Alternatively, vehicles may be used as 'data mules' to disseminate the data to the portal in a delay-tolerant multi-hop fashion.

Whilst not oriented around motor vehicles, the BikeNet project involves the collection of sensor data from bicycles [64]. Because of the lack of a plentiful energy supply, sensor data is only communicated back to base at the end of a journey or via a mobile phone if present.

Traffic information systems. By far the most popularly researched application for VANETs is the traffic information system. In these systems, vehicles act as *congestion sensors*, detecting the density of traffic that they are experiencing. This information can be shared with other vehicles so that they can plan a route that minimises delays or avoids areas of high congestion.

Conventional traffic information systems use sensors embedded in roads or located by the road-side which count the number of vehicles passing by. Data from these sensors is communicated to a traffic information centre which then broadcasts aggregated information over a radio channel using a protocol such as RDS [247, §1]. On the other hand, using vehicles as congestion sensors avoids the need for such costly infrastructure. It also has the pleasant property that the sensor density is naturally highest where congestion is heaviest, so higher-resolution sensing takes place where it is most beneficial.

The TrafficView system [186] enables vehicles to know the locations of nearby vehicles, to enable a driver-assistance application. Vehicles periodically broadcast the position and velocity information they have received from other vehicles, updated with their own information. In order to minimise the volume of data transmitted, compression and aggregation techniques are employed.

The Self-Organising Traffic Information System (SOTIS) [247] is a traffic information system based on the SODAD architecture (see Section 2.1.4.1). The opportunistic data dissemination scheme takes advantage of vehicles travelling in different directions encountering each other long enough to share their data. Simulation results show that accurate traffic information can be propagated more than 50 km from the source with low delay, even when only a small fraction of vehicles support SOTIS.

A similar traffic information system called StreetSmart has also been proposed [58]. In this system, vehicles construct their own maps of traffic density and exchange them with each other. Each node stores a compressed version of every other node's traffic information, expressed as summary statistics about clusters of traffic. To minimise the number of messages sent, only the interesting data are shared. Data about traffic behaving normally is considered not interesting; only data exhibiting signs of abnormal behaviour are communicated.

Other applications. The intent of the VEDAS system is to remotely monitor the health of vehicles in a fleet [134]. Data mining techniques are used to detect unusual driving patterns such as may be caused by drowsy drivers. VEDAS aims to minimise the volume of communication and minimise the energy consumption of in-car devices whilst also respecting the privacy of drivers.

Sivaharan et al. present a visionary application in which next-generation vehicles drive themselves autonomously [218]. Collisions between vehicles are avoided based on data shared via communication between vehicles. Versteegt and Verbraeck describe a deployment of automated guided vehicles (AGVs) for a logistics system under Schiphol Airport in Amsterdam [237]. In this system, AGVs can move autonomously between endpoints, even travelling within centimetres of other AGVs. Error accumulation is prevented through accurate dead reckoning, with low-slippage wheels and precise odometry.

2.2 Distributed computing

A network of vehicles can be thought of as a distributed computer. The study of distributed computing is a vast topic, the subject of a large quantity of research over several decades [44]. This section will merely scratch the surface of a number of areas of distributed computing, particularly those from which we can learn lessons and ideas to apply to large-scale networks of vehicles. We will focus upon drawing attention to issues related to the programming of distributed computing systems.

A good place to start is with Flynn's taxonomy of computer organisations [74]. He categorises computer systems based on the magnitude of interactions between their instruction and data streams:

Single instruction stream–single data stream (SISD) represent machines containing a single sequential processor, with no parallelism.

Single instruction stream–multiple data streams (SIMD) represent computers which exploit operations which are naturally parallelised, such as array processors which perform a single operation on many items of data at once.

Multiple instruction streams–single data stream (MISD) represent a hypothetical situation in which multiple processors apply different instructions to the same datum, generally deemed to be impractical [61, p6].

Multiple instruction streams–multiple data streams (MIMD) are computer systems in which multiple processors operate independently on different instruction streams.

Historically, the majority of programming language theory has been built up on the SISD model. Different approaches are required to handle the kind of distributed systems we are interested in, which fall largely into the MIMD category.

2.2.1 The von Neumann architecture

The 'von Neumann architecture' is a term given to the heavily influential model of computing proposed by John von Neumann in 1945 [238]. This model epitomises SISD systems, consisting of a single memory unit, an arithmetic logic unit and a control unit which decodes and executes instructions loaded from memory.

In an SISD or SIMD system, each instruction has a unique successor. Since execution is deterministic, an imperative style of programming may be readily adopted. However, this is not the case in an MIMD system due to interleaving. Parallel processing implies that instructions executed sequentially by one processor may be interleaved with instructions executed sequentially by another. This interleaving is not deterministic and can lead to defects such as race conditions. Kennaway and Sleep sum up the failing of the von Neumann architecture in the following way [137, p111]:

> "The underlying concern of a conventional programmer is to guide a single locus of control through a cunningly designed maze of assignment, conditional

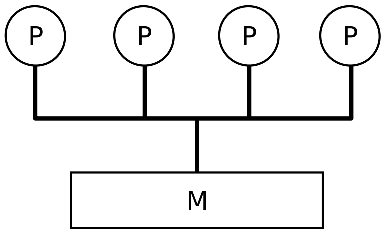


Figure 2.1: A tightly-coupled system in which memory is shared between four processors.

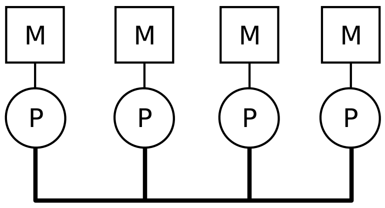


Figure 2.2: A loosely-coupled system with four processors each having a local memory.

> and repetitive statements. At each step the programmer has (perhaps quite unconsciously) as a major concern the details of *how* things are done rather than getting right *what* is done."

But it is improper to require the programmer to consider the individual control loci in each processor of an MIMD system; this would be difficult and prone to errors at best, and infeasible in general. This demands a declarative style of programming, where it is the *what* rather than the *how* which the programmer expresses.

Distributed computing systems are often classified into two categories: (*i*) tightly-coupled and (*ii*) loosely-coupled systems. Tightly-coupled distributed systems are characterised by memory being shared between processors, depicted diagrammatically in Figure 2.1. On the other hand, a loosely-coupled system consists of a set of processors each with their own local memories and a shared interconnect, as in Figure 2.2. (These diagrams are deliberately simplistic; the practical systems described below vary in their precise topology.)

Appendix C gives an overview of tightly-coupled systems. Whilst there is much of interest in this area, these systems have little bearing on large-scale, vehicle-oriented networks which are necessarily loosely-coupled. Section 2.2.2 surveys a variety of approaches to abstracting and programming loosely-coupled systems. Then Section 2.2.3 tells of the problem of determining where computation should take place in a distributed system. Complementary to the practical systems described here, Appendix D examines a variety of theoretical approaches to modelling distributed computation.

2.2.2 Loosely-coupled systems

Advances in computer networking technology have meant that it has become feasible for a network of computers to co-operate to execute a single application. This has given rise to loosely-coupled distributed systems. Unlike in tightly-coupled systems, components tend to suffer from independent failures, and delays due to communication may be unpredictable.

As a proposal for a more appropriate successor to the von Neumann architecture, Valiant has proposed a Bulk Synchronous Parallel (BSP) model[2] [233] that abstracts underlying hardware to allow algorithm designers to ignore low-level architectural concerns. In this model, each processor has its own local memory, and a router delivers data through the inter-connection network. The model assumes a synchroniser that enables a subset of the processors to execute in a co-ordinated fashion. In this way, communication is decoupled from synchronisation. Computation is organised into a series of 'supersteps', in which a set of independent local computations are performed before a communication phase followed by a global synchronisation. Skillicorn et al. have recently re-evaluated BSP with the benefit of hindsight, concluding that it is closely compatible with commercial multi-processor parallel computers that have recently emerged [220].

There is a wide spectrum of systems that can be classified as loosely-coupled, some of which can tolerate independent failures of components. At one end of the spectrum are network processors, which contain multiple processing elements on a single chip. These multi-core chips consist of a systematically arranged matrix of many independent processing elements, each of which is a fully-fledged computer with local memory, sharing a communication bus. Network processors are well-suited to high-speed packet processing applications commonly found in communications networks.

Then there is the non-uniform memory access (NUMA) machine, which consists of a set of processors in a single box. In these computers, memory is globally addressable but distributed, causing memory access times to be dependent on which address is being accessed.

Cluster computing and grid computing are terms which are broadly used to refer to loosely-coupled systems often involving large numbers of commodity computers. If there is a distinction to be made between these terms, it is that cluster computing tends to imply that the constituent computers have a common owner. Sometimes cluster computing is used to refer to a set of homogeneous servers, but can also apply to a network of workstations (NOW).

The term 'grid' was applied to this style of computing as a result of the vision of treating a large, distributed computing resource in a manner similar to other utilities such as electricity or water [78]. The dream is for 'computing power' to be a resource available to domestic and industrial users on demand, provided by an external supplier, rather than each party generating its own computing power.

Grids tend to be classified as either *data grids* or *compute grids*. Data grids are common in the scientific community where a large corpus of data is made available to collaborators across the globe. Compute grids involve participating computers co-operating to act like a supercomputer to perform a large computational task. We can distinguish between compute grids intended for a specific application from *cloud computing*, which aims to fulfill the vision described above. Cloud computing is usually implemented using a virtu-

[2]This model is similar in spirit to the PRAM model for tightly-coupled systems, described in Section C.2.

alisation layer such as Xen that allows multiple, mutually isolated, commodity operating systems to run in tandem on a single processor [21].

In loosely-coupled systems, memory is inherently distributed. Thus there is a decision to be made about how to support communication between processes[3]. Is the best approach to emulate a tightly-coupled system, and present the illusion of a globally shared memory? Or should a message-passing approach be adopted?

We will consider a number of points on the spectrum of possible approaches, starting with shared memory in Section 2.2.2.1, moving through RPC in Section 2.2.2.2 to message passing in Section 2.2.2.3. Publish–subscribe systems are examined in Section 2.2.2.4 and dataflow systems in Section 2.2.2.5. We then visit the related concept of code mobility in Section 2.2.2.6. Finally, Section 2.2.2.7 takes a step back and considers declarative programming techniques which do not require the programmer to consider individual inter-process interactions.

2.2.2.1 Distributed shared virtual memory

Distributed shared virtual memory (DSVM) is the name given to a layer of abstraction which presents the illusion of tight-coupling on top of a loosely-coupled system. This is achieved through a single, globally-addressable memory which all participating processors have the same view of. In this way, a programmer does not need to worry about which processor an item of data is located on, or how to obtain its value, as would be the case in a message-passing approach.

Shared memory is simple to use and familiar from imperative SISD programming. From a programmer's point of view, on a tightly-coupled multiprocessor machine, shared memory is inexpensive to deal with[4]. To prevent race conditions as other processes interfere with an operation on shared memory, locking strategies would be applied. This adds only a small overhead to the cost of reading from and writing to memory. However, in a loosely-coupled system, communication between processes could be many orders of magnitude more expensive.

Lenoski et al. commented that "a single address space enhances the programmability of a parallel machine by reducing the problems of data partitioning and dynamic load distribution, two of the toughest problems in programming parallel machines. The shared address space also improves support for automatically parallelizing compilers, standard operating systems, multiprogramming, and incremental tuning of parallel applications—features that make a single-address-space machine much easier to use than a message-passing machine" [147, p64]. However, implementing shared memory as a layer of abstraction on top of a loosely-coupled system brings with it the cost of creating and maintaining the illusion of a single, coherent memory. This is likely to entail more communication than is necessary, which a study by Lu et al. confirms leads to worse performance [162]. Network bandwidth becomes the limiting factor as the system scales.

[3]Here, and throughout the rest of the chapter, we use the term 'process' to refer to the encapsulation of computation and not necessarily to refer to an operating system process except where stated. In the literature, a variety of terms are used; we will use 'process' throughout for homogeneity.

[4]See Section C.1 for a summary of strategies.

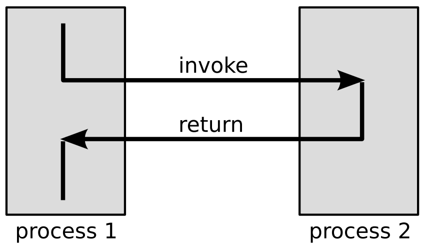


Figure 2.3: Remote procedure calling.

DSVM may be implemented in a user-level software library or at the hardware level, as is the case in a non-uniform memory access (NUMA) computer. In these computers, DSVM is used to present a globally addressable memory. This is implemented by each portion of memory being owned by a particular processor, and there being a central register recording these ownerships. A variety of practical implementations of DSVM with varying degrees of abstraction are outlined in Appendix E.

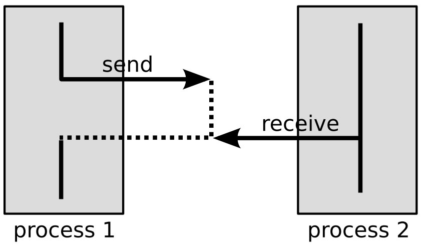
2.2.2.2 Remote procedure calling

Remote procedure calling (RPC) is the name given to techniques in which procedures located in another memory space can be invoked. A remote procedure call has a similar appearance to a local procedure call, meaning that single-process programs can be transformed into multi-process programs with moderately low cost. However, there will be some differences in the semantics of a local and remote call [156, p41]. For example, remote calls typically do not support call-by-reference semantics but only call-by-value semantics, where the arguments are copied. This means that the local and remote processes work on independent copies of the data. Moreover, transmitting the arguments and return values requires a means of *marshalling*: converting in-memory values to streams of data which can be sent over a network. Furthermore, there is a greater range of error conditions that can arise in a remote call compared to a local call; these must be dealt with through additional supporting error-handling code. Also, even in garbage-collected languages, there is rarely support for identifying remote objects as candidates for deletion.

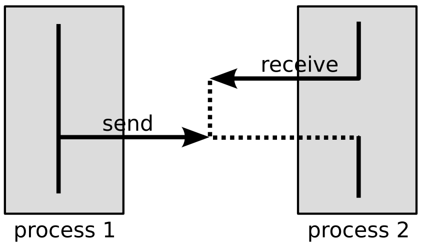
However, whilst RPC is used in distributed computing, it offers little scope for parallelism because invocations of remote methods are inherently synchronous (just as with invocations of local procedures). The caller must wait for the method to return before proceeding with its execution; hence, there is only one locus of program control, although it hops between processes. See Figure 2.3.

When the RPC paradigm is applied to an object-oriented language, it is usually referred to as remote method invocation (RMI). This usually entails support for references to objects across processes.

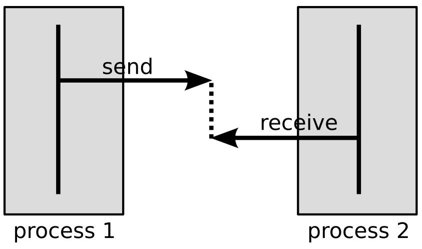
The Common Object Request Broker Architecture (CORBA) is an implementation of RMI that is language-neutral. Objects' interfaces are described in a common interface definition language (IDL) which has mappings to many implementation languages. An object request broker (ORB) communicates with the remote object on behalf of the local client, creating the illusion of it being local by forwarding arguments and return values. The ORB performs marshalling, converting the data to and from a common data representation.



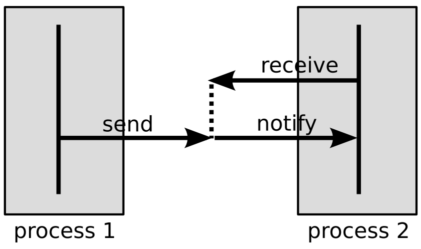
(a) Synchronous, with *send* before *receive*.



(b) Synchronous, with *receive* before *send*.



(c) Asynchronous, with *send* before *receive*.



(d) Asynchronous, with *receive* before *send*.

Figure 2.4: Message passing.

The Java implementation of RMI is built into the language [163]. It uses a 'registry' of remote objects; client-side 'stub' classes are automatically generated whose instances interact with the remote object, in a similar fashion to an ORB. Marshalling is achieved through Java serialization. Serialization is the process of writing an object's fields into a byte array, along with the fields of the objects in the graph of the referenced objects.

Web Services. Just as the World-Wide Web is a means for human interaction over the Internet, Web Services were conceived as a means for computer interaction over the Internet [4]. Computers publish descriptions of operations which they support in an XML-based language called the Web Services Description Language (WSDL). These operations are often procedural and thus their remote invocation can be thought of as RPC, and the client's programming model is synchronous.

Some Web Services generalise beyond RPC semantics and follow a message-passing model where, again, the schema for the messages is defined by the service's WSDL specification. The Business Process Execution Language (BPEL) is used to describe high-level 'business protocols', which are stateful workflows between Web Services involving message-sending [8]. This standard separates the deployment information (where the services are executed) from the description of the protocol.

2.2.2.3 Message passing

At the other extreme from a shared-memory programming model is the message passing model. In the former, the programmer can be oblivious to which machine a piece of memory being accessed resides on, and the library or framework performs the necessary inter-process communication to transfer data between processes. In the latter, the programmer must know precisely which process to communicate with, and at what time. All communications are explicitly instructed by the programmer, and the library or framework is reduced to merely implementing queueing of incoming messages and ensuring reliable delivery of outgoing messages.

Compared to shared memory, message passing is more naturally suited to loosely-coupled systems, as message passing forms the basis for the lower-level network communications. Moving the control over precisely what communication is performed from the framework to the language domain brings potential increases in efficiency, as the programmer can avoid unnecessary communications, but comes at the cost of being a greater burden for the programmer. Low-level concerns such as physical addressing, marshalling and flow control are moved into the application layer [70, §3.1]. Unlike in shared memory, a message-passing scheme requires the programmer to identify when to communicate, what to communicate and with whom to communicate. Furthermore, a message-passing scheme adds the overhead of requiring data to be marshalled before it can be communicated.

In a message-passing paradigm, there is a decision to be made as to whether the send and receive calls are blocking or non-blocking. If blocking, the communications between processes are synchronous: a call to *send* will not return until a matching call to *receive* by another process has been made. If non-blocking, communications are asynchronous, and processes require buffers to store incoming messages which have not yet been picked up. Figure 2.4 depicts the difference between asynchronous and synchronous message passing.

The Erlang language was designed and built up around the notion of concurrency [11]. It permits the simple creation of lightweight processes, independent of the processors on which they execute, without shared state between processes. This is achieved through a functional programming style, with message passing between processes. Message passing is asynchronous and is implemented by each process having a 'mailbox'. Pattern-matching capabilities allow processes to deal efficiently with messages potentially arriving from several other processes concurrently.

The PICL library is oriented around message passing, with bindings to C and Fortran [87]. The message-passing paradigm is supported through *send* and *receive* calls in which a symbolic identifier is attached to each sent message, indicating the message's type. Receipt of messages can be done selectively based on the value of the identifier in the message received. PICL also supports high-level routines such as finding the extrema of a distributed dataset, or finding the product of a distributed set of vectors.

As well as supporting the sharing of regions of memory, the PVM library includes support for message passing. As with PICL, this is achieved through typed messages [227, §2.2]. Variants of *receive* include a version which returns with an error value if no matching message is received within a certain time period, and a version which returns with an error value if a particular number of non-matching messages have been received without a matching message.

Perhaps the best-known communication library is MPI, the Message Passing Interface [242]. This library supports both blocking and non-blocking inter-process communication. A non-blocking *receive* notifies the system that a process would like to receive a message.

Later, the process checks to see if a matching message has arrived [196, p32]. A blocking *send* will not return until the local message buffer can be modified without corrupting the message. There are three derivatives of *send*, which apply to both the blocking and non-blocking cases [242, §2]: (*i*) standard send—a message may be sent regardless of whether a corresponding receive has been initiated; (*ii*) ready send—a message is only sent if a corresponding receive has been initiated; and (*iii*) synchronous send—the send operation will only return when a corresponding receive has been initiated.

BSPlib is a communications library (with bindings to several conventional languages) for programming in the Bulk Synchronous Parallel (BSP) model, consisting of twenty basic operations [110]. The superstep style of programming is achieved as processes perform a series of computations on data held locally at the start of the superstep, perhaps even communicating with other processes. The end of a superstep is marked by a call to *bsp_sync* at which point all processes synchronise. BSPlib provides facilities to support several parallel programming paradigms. The BSP message-passing model involves a non-blocking send operation which delivers a message to another process's buffer, guaranteed to be accessible at the beginning of the next superstep. If the message is not read out from a buffer during that superstep, it is discarded. A second paradigm is 'direct remote memory access', whereby a process registers data, via *bsp_put*, which may be accessed from a remote process via *bsp_get*.

Grid computing. Message-passing is common in applications deployed on large-scale compute grids. An example is the SETI@home application, in which large volumes of radio data received from space are processed to search for the presence of particular patterns. The designers of this application created a small client program which can be installed on volunteers' commodity computers and which communicates with the central repository to fetch a batch of radio data. This is then processed locally using otherwise-idle CPU cycles. A summary of the results of each dataset are then sent back to the co-ordinator. Similar projects include Folding@home, in which the data processing involves simulations of protein folding, and the Great Internet Mersenne Prime Search. General frameworks in which similar volunteer-computing projects are deployable include Distributed.net, which has tackled a number of brute-force mathematical problems, and grid.org, which has tackled brute-force biological problems.

The Globus toolkit has been widely adopted as a framework for implementing grids in scientific communities. It is an implementation of various open standards, most notably OGSA which defines the notion of a grid service and the protocols which can be used to invoke it [79]. It contains the Grid Resource Allocation Manager (GRAM) which controls the scheduling and execution of jobs in a compute grid [53]. Descriptions of jobs are submitted to GRAM in terms of the resource requirements, files which need to be imported before execution, and the path to the binary to be executed along with any required arguments and environment variables.

Popular systems for converting collections of networked workstations into compute grids include Butler [190] and Condor [229]. In Condor, computers with spare CPU cycles register themselves with a centralised 'agent' to join the grid. Users submit jobs to the agent, which allocates resources to execute it. Butler operates in a similar manner, but differs from Condor in that it does not provide automatic process migration [229, §7.1].

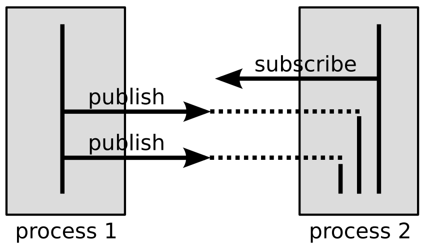


Figure 2.5: Publish–subscribe.

2.2.2.4 Publish–subscribe systems

The message-passing approach required processes to possess knowledge of other processes with which to communicate. This coupling between processes is broken in publish–subscribe systems [70]. Instead, messages are sent to a broker. Processes make 'subscriptions' with the broker which declare their desire to receive messages matching a certain pattern. This approach to communication is inherently asynchronous on both the sender and receiver (known as 'synchronisation decoupling') and is characterised by processes not needing to know any details about the process or processes with whom they eventually communicate (known as 'space decoupling'). Furthermore, the persistence of the broker means that the publisher of a message need not be connected at the time when the message is delivered to a subscriber ('time decoupling'). Figure 2.5 depicts the publish–subscribe paradigm diagrammatically.

Two kinds of publish–subscribe systems can usually be distinguished based on the method by which a message is identified as fulfilling a subscription. In topic-based matching, each message has meta-data associated with it which describes the nature of the data contained in it. This description could come from a hierarchical ontology. On the other hand, in content-based matching, it is the content of the message which is examined, using a collection of comparison and string-matching operators. For example, the Java Message Service is a topic-based publish–subscribe mechanism with a Java API. On the other hand, the Cambridge Event Architecture is content-based [16].

Linda is a model of process communication which provides a handful of operations to add to a base language which uses a tuple space to share data [41]. When a process wants to communicate, it generates a new tuple and emits it into the globally-shared tuple space. When another process wants to receive that communication, it removes that tuple from the tuple space. This paradigm is familiar from how applications tend to communicate with each other, or with future versions of themselves, by writing to files.

However, Linda is not a publish–subscribe system in the strictest sense because there is no synchronisation decoupling; tuples are extracted from the tuple space in a synchronous manner. On the other hand, in a true publish–subscribe system, messages are delivered to subscribers asynchronously. Nevertheless, the programming model is closely related. Indeed, in many recent implementations of Linda such as JavaSpaces [81], the tuple-space model is augmented with asynchronous delivery [70, §3.4]. In Linda, the 'broker' can be thought of as the shared tuple space along with the pattern matching mechanisms built into the framework.

Closely related to publish–subscribe systems are *message queueing* systems [70, §3.5].

These systems tend to be oriented around providing point-to-point communication between two parties rather than providing general communication from m producers to n consumers. Messages are delivered with guarantees over ordering, which is not necessarily the case in publish–subscribe systems. Moreover, messages are received based on their presence in a particular queue rather than based on the content of the message matching a particular pattern. Widely available commercial implementations include IBM WebSphere MQ and Microsoft MQ.

2.2.2.5 Dataflow

Rather than allowing arbitrary bi-directional communication between processes, as is the case with message-passing, a dataflow approach restricts each process to receive data from one or more input streams and output data to one or more output streams.

The River programming environment adopts a dataflow approach to describing the communication relationships between processors [13]. In an attempt to avoid the problems caused by heterogeneity in a loosely-coupled MIMD system, where some processors might run significantly slower than others, River employs 'distributed queues' between processes which connect multiple data producers to multiple data consumers. Then, processes produce and consume at whatever rate they are capable of, meaning that load balancing occurs automatically without the need for additional co-ordination communication between processes. This leads to a straightforward programming model in which processes (called 'modules') are described in a functional manner with no explicit communication to or from named processes.

2.2.2.6 Code mobility

Mobile code is the term given to a system in which the processor on which a particular piece of code is executed changes dynamically [82]. This occurs when the data which is communicated between processors includes executable code. A familiar example of code mobility is PostScript, in which programs describing the layout of a document are sent to printers to execute. Another example is Java, which supports dynamic class-loading, potentially from remote locations. This is a facility which is particularly well-known due to applet viewers in web browsers, in which the required classes are loaded from across the web and executed locally.

There are a number of subtle distinctions between code mobility paradigms, charaterised by the location of code and data before and after execution and where the code is executed. Three broad categories are as follows: [82, §IV-A]

Remote evaluation paradigm. A process has the know-how necessary to achieve a goal but lacks the data or resources required to execute it. It sends the code to a process on a different processor, which executes the code using its own resources and sends the results back to the original process.

Code on demand. A process has the data that it needs to manipulate, and the facilities to manipulate it, but does not not have the information about how this processing should occur. The know-how about how to manipulate the data is obtained by requesting code from a process located on another processor.

Mobile agents. A process on a particular processor has the know-how necessary to achieve a goal, but the resources it needs to execute it are located on a different processor. The process therefore migrates to that processor and is executed there.

Distinctions are also made over the granularity of the mobility. Process migration is the migration of entire processes and their address spaces; object migration is the migration of objects between address spaces. The Emerald language provides object-level migration [129], using a global namespace to identify objects and a 'forwarding address' scheme to keep track of the location of objects. The Rover toolkit [127] works at the same granularity, terming mobile objects as 'relocatable dynamic objects'. In both of these approaches, it is the programmer who dictates when migrations occur.

Rather than being at the programming language level, the DEMOS/MP distributed operating system supports process-level migration at the operating system process level [201].

In the compute grid frameworks described in Section 2.2.2.3, the programs to be executed were assumed to be installed through separate means. For example, an assumption in GRAM, Condor and Butler is that they may be accessible through a filesystem shared by all participants. On the other hand, in data grids, the execution of programs generally involves the transmission of the programs over the grid to avoid the cost of transmitting large datasets. For example, in the OGSA-DAI framework [9], the data processing to apply to a dataset is described in a scripting language which is independent of the nature of the organisation of the data. These scripts are sent to the data source for execution, and the results transmitted back in the response.

2.2.2.7 Declarative approaches

An alternative approach to programming loosely-coupled systems is to avoid nominating a specific style of inter-process communication and leave this decision to the compiler. This demands a higher level of programming abstraction, in which it is likely that the programmer is not concerned with hand-crafting individual processes, but rather in describing the intended overall behaviour.[5]

MapReduce. MapReduce is a framework designed by Google for distributing the processing of large, static datasets in a cluster of commodity computers [56]. The programming model entails the user describing the nature of the processing in a functional style. The program is automatically parallelised and executed on a cluster of computers, which can scale to several thousands of members. The run-time system partitions the input data, schedules the program tasks, manages inter-task communications and handles failures.

[5] This is reminiscent of the mathematical approach mentioned in Section D.4 whereby the program is expressed at a high level in a non-parallel manner and a compiler performs equational transformations to parallelise it.

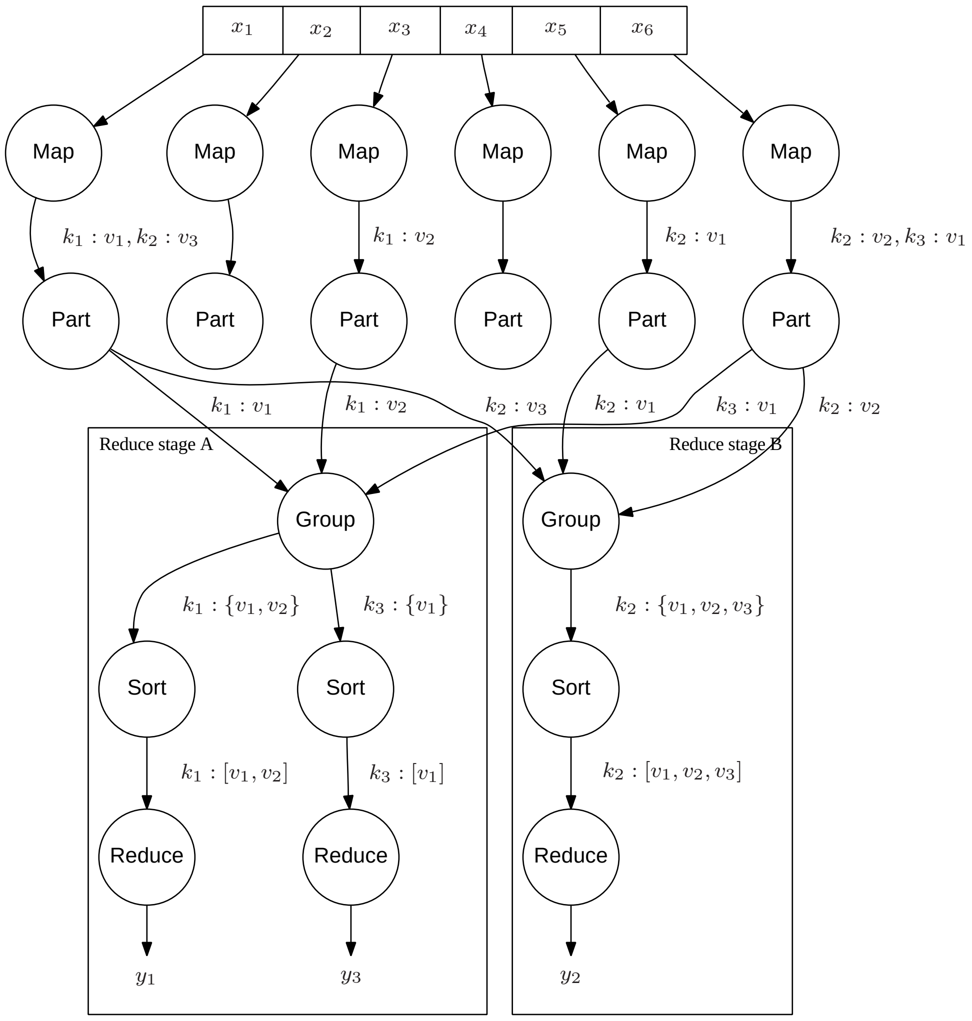


Figure 2.6: An example of the computational model of a MapReduce program. The Map and Reduce functions are user-specified. Reduce stage A is responsible for reducing keys k_1 and k_3; stage B is responsible for k_2.

The nature of the computational model in the MapReduce framework is exemplified in Figure 2.6. The edges in this graph indicate the data flow which arises at run-time with the values labelled.

The two functions which the programmer must specify[6] are the *map* and *reduce* functions[7]. The map function takes a chunk of the input data as a single key–value pair and returns a set of 'intermediate' key–value pairs. For example, in a word frequency-counting application, the input keys may be URLs and the values the textual content of the pages referred to by the URLs, and the output keys would be words, with the values being their frequencies.

The run-time system gathers together and sorts all the values associated with a particular intermediate key, passing them to the reduce function responsible for that key. This function then aggregates the input values and returns an output. Typically this output is either empty or a single value. In the word frequency application, the reduce task would sum the frequencies received from the map tasks for a given word, yielding its frequency of occurrence across the original set of documents.

Usually, for a given application, the number of map function instances is chosen to be proportional to the size of the input dataset. An optionally programmer-specified *partitioning* function indicates which reduce stage is to be responsible for executing the reduce function for which keys. Usually the number of reduce stages is chosen to be a small multiple of the number of machines available in the network; the number of reduce functions executed by each reduce stage is related to the number of intermediate keys generated by the map functions.

As well as the computational model, the MapReduce framework also offers fault-tolerance. This is achieved through the co-ordinating process (known as the 'master') pinging the machines running each task and dynamically re-assigning the work to a different machine if a response is not received after a short period of time.

Dryad. Microsoft's Dryad system [123] has similar goals to MapReduce. It provides a general-purpose distributed execution engine that exploits data parallelism identified by the programmer, and has been shown to scale to machines with multiple CPU cores and data-centres containing thousands of computers. The main difference to MapReduce is in the programming abstraction. Whilst MapReduce restricts programmers to a rigid model in which computation must be expressed in terms of map, sort and reduce stages, Dryad permits the programmer to describe the application as an arbitrary directed acyclic graph.

The graph contains nodes that contain code to be executed sequentially and edges that indicate the direction of data flow between nodes. A programmer constructs a graphs using a number of graph composition operators. Furthermore, the physical nature by which data is transmitted between two nodes on an edge is specified by the programmer. This could use a variety of transport mechanisms, such as files, TCP pipes and shared-memory FIFOs.

[6]This approach is justified by the Bird-Meertens Formalism which is discussed in Section D.4.1.

[7]Confusingly, the user-specified functions are referred to as 'map' and 'reduce' functions, although these are the merely the functions which are conceptually passed to the *map* and *reduce* functionals, rather than being drop-in replacements for those functionals themselves.

Dryad offers the facility to re-write the application graph during run-time [123, §5.2]. Depending on the size of data and the speed of computation, which may only become known at run-time, it may be possible to make more efficient use of resources with a different graph. For example, an aggregation tree could be formed to increase the efficiency of a set of associative and commutative operations. Such dynamic re-writing allows the application to self-optimise and adapt to changing network conditions.

2.2.3 Task assignment

In addition to the decision about which inter-process communications paradigm to adopt, a distributed computing framework needs to provide facilities to decide which physical processor will execute each process. This is known as the problem of 'task assignment', since processes can be thought of as collections of program tasks. This could either be implemented as a language feature to leave the task assignment decision up to the programmer, or a feature of the execution framework so that the decision is made automatically.

In some architectures, such as a NUMA computer, the number of processors and the communication links between them are fixed. Therefore it is theoretically reasonable to leave the task assignment decision to the programmer. However, a good assignment requires the processors' characteristics to be predictable; this may not be the case if the computer is shared between applications. In other architectures, such as cluster or grid computing, the programmer cannot make this decision, so the mapping of processes to processors can only be made by the framework. However, even when designing applications for highly regular, well-understood topologies, applications in which the programmer binds processes to particular processors will not be portable to different configurations. Moreover, it may be that automatic task assignment can resolve complex trade-offs better—and reach a decision more quickly—than a manual approach performed by experts.

A closely related problem is that of task partitioning. This is the problem of determining which program components should be grouped together into a logical unit to execute on a single processor, and is thus a pre-requisite for task assignment. In a sense, task partitioning is a technique for converting a monolithic program into a distributed program, by identifying components that are suitable for execution on separate processors. However, whilst the execution can be distributed, this area of research tends to assume that there is only a single locus of control in the program, and that the various partitions interact via a mechanism such as RPC. This avoids the question of whether the program can be parallelised, where each partition would be executing concurrently. Appendix F contains a discussion of the nature of task partitioning techniques.

2.2.3.1 Static automatic task assignment

Kremer et al. have developed a compiler which statically analyses a program which would normally run solely on a battery-powered mobile device and considers whether it would be better to off-load the execution of a part of it to a fixed computer with more resources [141]. The static, compiler-based approach means that the whole program can be analysed and that predictions of future behaviour are not solely based on observed past behaviour. The decision about which of the program's functions to off-load is based on whether remote execution would save more energy than local execution. This is computed based on an analysis of the state of which objects would need to be copied to the remote computer. In order to save further power, the compiler causes the mobile device to hibernate by inserting code to perform power state transitions. During hibernation, it may choose to wake up periodically to check on the progress of the remote execution, perhaps requesting the partial results of computation so that the processing can continue even in the case of disconnection.

The Titan framework [160] has been designed to allow programmers to design applications for body-area sensor networks in a task-oriented manner. An application is designed by selecting tasks from a library of pre-defined tasks, and connecting them together to form a task graph. A 'network manager' determines which processing node should execute each task, based on a greedy algorithm which respects nodes' capacities to execute only a certain amount of tasks. Although we classify this work as implementing merely static automatic task assignment, the network manager can also recompute the assignment of tasks to processing nodes in the event of a node failure, at run-time.

Task assignment has also been applied to distributed multimedia applications. Applications such as teleconferencing involve high-bandwidth data streaming over a network. In these environments, the computers which could be employed are usually heterogeneous. Hagin et al. address the task assignment problem by proposing a heuristic-based algorithm called SIGMA which minimises total communication and computational cost in polynomial time [96]. They note that quality of service is an important issue in streaming multimedia applications, so SIGMA ensures that a desired level of quality of service is met by the assignments which are produced.

In the domain of query processing, the task assignment problem can be restated to question whether query operators (select, project, join) are best executed at the client or at the database-server. Executing them on the client is known as 'data-shipping'; executing them on the server is known as 'query-shipping'. In practice, neither approach will always be preferable, so a 'hybrid-shipping' approach is often adopted [80]. In this approach, a plan about where operators should be executed is made based on the available CPU resources at the client and the server.

2.2.3.2 Dynamic automatic task assignment

Dynamic task assignment refers to modifying the assignment of tasks to processors during run-time. Not only does this mean that a technique for re-evaluating the best assignment based on changing data is needed, but also that there must be a mechanism to provide code mobility (see Section 2.2.2.6). Without this, the execution of the program would need to be halted; another static task assignment performed; and execution started again, at regular intervals.

Job scheduling systems such as Condor provide a primitive form of dynamic task assignment. During the course of the execution of an application, the agent co-ordinator performs dynamic load balancing. However, this load balancing is very coarse-grained—at the granularity of jobs rather than at the granularity of functions.

The Abacus system is a run-time system that dynamically changes the assignment of processes to processors [6]. Programmers specify programs in C++ in which some objects are syntactically marked as 'anchored' and others as 'mobile'. Anchored objects need to be executed on a particular processor (for example because they access local storage on that machine). Mobile objects may execute on whichever processor is deemed best. The specification of mobile objects must satisfy two criteria: they must not expose state publicly except that which is accessible through the exported interface, and they must support methods to serialise and deserialise their state. The Abacus run-time system monitors method calls, which are executed synchronously (RPC-style), keeping track of (*i*) the sizes of arguments passed to and results returned from method calls, recorded in a data flow graph which maintains moving averages; (*ii*) the volume of memory allocated to each object; and (*iii*) the duration of each method's execution in terms of CPU time and time spent blocked [6, §5.3]. From this data, the Abacus run-time system migrates mobile objects to different processors when this would seem to reduce the application's average response time.

The goals and achievements of the Equanimity system are similar [108]. The authors of Equanimity term this concept 'service rebalancing', and similarly seek to alter the interface between the client and server portions of a program to optimise some property via dynamic migration.

In pervasive computing environments, maximising the battery life of mobile devices is of paramount importance. The Spectra system seeks to achieve this through computation off-loading onto better-endowed fixed processors [73]. Spectra gathers data about an application which goes beyond standard profiling to also monitor battery energy use and the states of devices' caches. Based on this information, it decides where each task constituting an application is to be executed. As well as this, it also decides the quality of the computation which each task is to perform. Programmers specify various versions of a task which fulfill its specification to different fidelities [212]. For example, a function to find the maximum of a set of values could compute an accurate value by iterating through them all (perfect fidelity), or return the maximum of a small sample of values, which approximates the true result (low fidelity).

Research by Chen et al. seeks not only to provide a framework to dynamically decide whether to execute a method remotely or locally, to minimise energy consumption, but also whether just-in-time (JIT) compilation of Java bytecode to native code should be performed remotely or locally [46].

At the operating system process level, the Remote Processing Framework (RPF) runs on a mobile device and initially performs static assignment of processes to either the device itself or to a fixed server [210]. The decision about whether to run a process locally or remotely is rather primitive: the process is executed both locally and remotely on several occasions, with the total energy consumption measured each time. The lowest-cost approach is adopted on most future occasions, with occasional use of the higher-cost approach to confirm that it still has a higher cost [210, §III.6].

In the Java-based toolkit devised by Omar et al. [192], application performance and power consumption on a battery-powered portable device is monitored. Their run-time system uses this information to determine whether objects should be instantiated locally or remotely. This decision is transparent to programmers via the use of proxy objects.

DFuse is a framework for programming sensor network applications which involve data fusion [142]. Applications are specified as task graphs, whose nodes represent data fusion functions. The assignment of data fusion tasks to physical processors is initially performed statically through a simple algorithm in which tasks are assigned naïvely then optimised through each node locally deciding whether it would be better to transfer its tasks to a neighbour. This mapping is re-evaluated against a programmer-specified cost function regularly during run-time, to cope with changing conditions in the sensor network such as node failure.

As well as providing an automatic partitioning service, MagnetOS performs dynamic task assignment of Java programs, primarily aimed at ad-hoc and sensor networks [22]. At run-time, the pattern of communication between the application's components is profiled. Each node makes a local decision about where to move the application components it hosts to. Components are migrated in the direction of greatest communication in an attempt to minimise communication costs.

2.3 Programming sensor networks

In Section 2.1, we determined that networks of vehicles can be thought of as a particular kind of sensor network. Vehicular networks are also a form of distributed system, so techniques for programming distributed systems were explored in Section 2.2. However, above the low-level message routing and message dissemination protocols in Section 2.1.4.1, there has been little research into higher-level programming languages or frameworks for wireless sensor networks. This section describes approaches to programming wireless sensor networks in an attempt to shed some light on the problem of programming vehicular networks.

In traditional sensor networks[8], it is typically expected that programmers write code for each sensor node individually, and compile and download the executable image into them manually [34, p187]. A popular programming language in which this can be done is nesC, an extension of C which adds a component-based programming model and increases safety through reduced expressive power [85]. nesC is an entirely static language: there is no dynamic memory allocation, and the call-graph is fully known at compile time. These restrictions mean that programs can be easily analysed by a compiler, and optimisations can be simply and accurately applied.

nesC programs are expressed in terms of a graph of components. Components expose interfaces to which other components can be wired. A component is either a 'module', which provides application code, or a 'configuration', which wires other components together. The graph is arranged such that commands flow downwards between components and events flow upwards between components.

TinyOS is an operating system for resource-constrained sensor nodes, implemented in nesC [109]. TinyOS provides a set of reusable system components, such as an analogue-to-digital converter, a timer and a radio component. Components can create 'tasks', which are short-lived units of computation to be performed. A scheduler is provided

[8]A light introduction to sensor networks is provided in Appendix B.

which arranges for the execution of tasks in FIFO order. Tasks run to completion without being pre-empted (except by interrupts triggered by the receipt of an event); this means that only a single stack is needed, which avoids wasting memory by allocating stacks to non-running tasks. Longer-running operations (such as performing a radio transmission) are defined in a *split-phase* manner: the command requesting to perform it returns immediately and its completion is signalled at some later time by an event.

Developing sensor-network applications in nesC with TinyOS has rapidly become the norm; an approach adopted by over a hundred research groups worldwide [109, p2]. However, there are a number of limitations to this way of programming:

- Programming standalone nesC applications or implementing applications within TinyOS is inherently a static approach since the application's call-graph is known at compile-time. This means that there is no facility to change the code that is executed on a node other than reflashing its ROM and rebooting.

  However, many sensor networks are long-running and intended to operate with minimal human intervention, perhaps for months or years. Over this length of time, it is common to expect that the nodes may be desired for use in a variety of applications. Even if just running a single application for their lifetime, it is likely that the application may need bugs to be fixed or new features to be added as new needs come to light, in which case the ability to upgrade a running application is desirable.

- The compilation of TinyOS applications to a statically-linked system image means that there is no facility to run more than one application on a sensor node.

- The requirement for the programmer to decide which node to use to execute which part of the application means that the programmer must have knowledge of which nodes will participate in the network, and what the topology of the network will look like. This is not a practical assumption to make when the network is highly variable or the application is intended to be deployed in a variety of physical topologies.

- It is tricky to program non-trivial applications with TinyOS. The split-phase paradigm means that complex logic cannot be encapsulated within a single abstraction but must be encoded by stitching together many smaller command handlers and event handlers [222, p766].

These deficiencies have led to the suggestion of using higher-level programming languages and run-time systems on nodes which manage the running of the application, allowing reconfiguration, dynamic upgrading and multi-tasking. One possible solution could be to hard-code a collection of core algorithms into each node and to provide a facility to select which one to use by sending a request over the wireless network interface. However, it is not feasible for this set of algorithms to be general enough to suit all possible sensor network applications whilst not occupying an unreasonable amount of memory space.

Another potential solution would be to allow nodes to be programmed over the air, by transmitting executable images from a base station. However, this would be extremely costly in terms of network traffic, and may even be infeasible in networks where nodes are sometimes unreachable from other nodes [34, p187].

Various approaches have been adopted between these two extremes. In Section 2.3.1 we describe approaches which view the network as a distributed database. In Section 2.3.2 we consider approaches which employ a virtual machine running on nodes and use higher-level programming languages.

2.3.1 Distributed database

One approach is to treat the sensor network as a distributed database. Each sensor can be viewed as a part of a database which can be queried to extract information about the environment. With this approach, an application can employ a high-level declarative language to describe what information is to be extracted from the data, as is the case with SQL in conventional DBMSs. In this way, the programmer describes what data is required, addressing the query to the network as a whole, rather than addressing individual sensor nodes. On the other hand, this approach limits the programmer to use a restricted set of operators to process the data. For example, it might be hard to express complex interactions between collaborating nodes.

With reference to conventional DBMSs outside the field of sensor networks, there has been much research into the closely-related area of distributed query processing [140]. As well as concerning techniques for caching and replication of data, this research involves determining where best to execute data processing operators in order to minimise communication costs. Even in a non-distributed sense, a given SQL query could be executed in a variety of alternative ways (known as execution plans). These plans differ in terms of the shapes of the trees of the relational operators, the permutations of these operators, the choice of implementation of operators (in particular, join has a number of implementations) and the permutations of sub-trees [31, p13].

Unlike in DBMSs, where a query runs to completion and returns a single result, in sensor networks applications often want to execute *continuous* queries [253]. Continuous queries run indefinitely (or for a specified duration) and report results during their execution with a specified frequency. Executing a continuous query once is preferable to repeatedly executing conventional queries from the point of view of minimising network traffic: in the absence of continuous queries, a polling strategy would be necessary. In turn, this gives rise to large energy savings—crucial in networks of battery-powered devices. In Motes[9], transmitting a single bit of data has been found in one study to be equivalent in terms of energy consumption to executing 800 instructions [165, p132]. In another study, it has been observed that the cost of executing 3000 instructions is equivalent to sending a bit one hundred metres by radio [200, p55].

2.3.1.1 Approaches based on SQL

SQL supports aggregation operators such as min, max, count, sum and average. Some DBMSs offer additional operators. With both conventional and continuous queries, there is a question over the point at which aggregation should be performed. One possibility is to gather all of the data (after selection and projection) at the point from which the

[9]Motes are an implementation of low-power sensor nodes. See Section B.1.

query was issued (the 'sink') and perform the aggregation there—the centralised approach. However, this may result in a far greater volume of communication than if the aggregation is performed closer to the source of the data. Hence this approach does not scale to large networks. Experiments have shown that in-network processing can contribute considerable savings in network traffic [104, §6.1].

Tiny Aggregation (TAG) [165], part of the TinyDB [167] sensor-network querying framework, extends SQL to allow continuous queries. In-network optimisations are possible based on knowledge of the particular aggregation function which is requested in a query. These optimisations reduce the amount of communication required. Further optimisations can be attained through snooping messages on the shared channel [166, §3.3].

As well as the standard SQL aggregation operators, TAG also permits custom aggregation operations to be defined. They can be expressed in terms of three functions: a merging function, an initialiser and an evaluator [165, §3.1]. The initialiser expresses how a single value is to be wrapped into a form which can be passed to the merging function to combine it with another. The evaluator deconstructs a value of this form into the result of the aggregate.

Similarly, the Cougar system provides a means of accessing data from nodes in a sensor network using an SQL-derived syntax [32]. As with TAG, continuous queries are supported. Madden and Gehrke, the creators of TinyDB and Cougar respectively, have shown how the two systems share similar goals and implementations [86].

The Sensor Information Networking Architecture (SINA) also uses an SQL-like syntax [224]. The variation on SQL provides a spreadsheet-like facility: the results of a query can be stored in named 'cells' and relationships between cells can be set up. Sensor nodes in SINA can arrange themselves into clusters, based on power levels and proximity, in order to reduce the number of messages and to remove the need for long-range wireless communication on all nodes.

IcedB is a continuous query processing system which takes account of the intermittent connectivity of mobile nodes [258]. Applications address continuous queries to a central server known as a 'portal' which distributes the queries to mobile nodes and collects results from them. Intermittent connectivity means that the results of a query need to be buffered at the mobile node, to be streamed to the portal later. Even when there is a connection from a node to the portal, it may not be that the available bandwidth always exceeds the amount of data to be sent. Instead of making this assumption, ICEDB prioritises the delivery of query results so that the data most highly valued by the application is transmitted first. Programmers express the prioritisation of data through an extension to SQL. Data is prioritised on both a local and global basis. Locally, items are ranked so that either the more interesting data or a coarse form of the data is delivered first. Globally, the application can provide feedback that certain data is to be considered more valuable than other data. For example, an application may wish to prioritise receipt of information concerning a geographical region about which little is presently known.

2.3.1.2 Other approaches

Heidemann et al. have proposed an approach to performing in-network processing that is similar in spirit to the SQL-based approaches but uses a different methodology [104].

Rather than performing routing based on named sensor nodes, routing is entirely data-oriented, using directed diffusion as the mechanism by which data flows through the network (see Section B.5.1). Applications indicate 'interests', which describe the sensor data and values that are to be matched; sensors listen for interests that overlap with what they are capable of providing and emit their data.

IrisNet is a distributed database analogous to the SQL approaches described above, but using an XML database [187]. As before, queries are deployed into the network from a sink and results are delivered back to the sink, whilst the precise nature of the query processing is hidden from the programmer. Aggregation is performed as close as possible to the sensors to minimise communication overhead. For a given sensor, programmers create a 'senselet': an executable which transforms the sensor data into an XML representation. The programmer also specifies a database schema that describes metadata about the sensors. Queries are expressed in the XPath query language.

The DataSpace paradigm embodies the notion of data being geographically distributed throughout space [121]. Whilst this does not solely apply to sensor networks, this is a natural application of the concept. Queries contain both a spatial element and a content-based element. The spatial element indicates which nodes are being queried (which drives the geographic routing) and the content-based element provides filters for the data. If a node is located within the specified region and holds data that matches the content filters, it replies by providing its data back to the sink.

The MiLAN framework provides a programming abstraction for sensor networks in which mobile nodes' battery life is maximised whilst respecting applications' quality of service requirements [105]. This optimisation is effected by adjusting properties of the network, such as altering which sensors send data and which nodes act as routers. For example, if there is a high density of sensors in one geographic region, but the application only requires a small amount of information about each region, some sensors in the densely populated region could be switched off to save energy. Quality of service requirements are honoured as applications specify the quality they require for each variable (expressed as a 'state-based variable requirements graph') and each sensor specifies the quality to which it measures each variable (expressed as a 'sensor quality-of-service graph'). The MiLAN framework will then attempt to find sets of network nodes that can satisfy the required total quality of service for all variables required by the application.

DSWare is a middleware which provides a data-centric view of sensor networks [152]. It provides a set of common database-like services which applications often require, hiding the low-level complexities of sensor networks. These services include event detection (which includes support for compound events), in-network data storage (involving distributed hashtable techniques and replication), data caching (spread intelligently to minimise communication costs and query response time) and data subscription.

2.3.2 Active sensor networking

Treating the sensor network as a distributed database leads to the view of sensor nodes as mere sources of data rather than general computational devices. Hence, the query paradigm of database-oriented techniques tends not to be flexible enough to efficiently express complex algorithms and interactions between nodes. Many applications in sensor networks can benefit from collaboration between nodes, where decisions are better made locally in a peer-to-peer fashion compared to being controlled from a potentially distant sink. For example, the tracking of a moving target is most efficiently achieved by involving only a small number of neighbouring nodes.

Other problems inherent in the distributed database approach include reply implosion at the sink which causes a bottleneck, and the need for a communication link from all nodes back to the sink [124, §1].

Hence, an alternative approach to programming sensor networks is to allow the sensors to execute arbitrary instructions. In a number of systems, this is achieved using a scripting language with an interpreter or a virtual machine running on the sensor nodes. The language must be powerful enough to allow sensor data to be handled appropriately and to allow complex interactions between nodes to be described, but must hide the low-level details of the platform. As well as providing an abstraction from the hardware, the virtual machine may even permit multiple applications to execute concurrently on a single node.

To steer clear of the danger of requiring the programmer to write code for each node individually—which was the case for programming in TinyOS—the paradigm of mobile agents is often adopted. We introduced mobile agents as a specific case of mobile code in Section 2.2.2.6. The computational model for mobile agents in sensor networks is that rather than the raw sensor data travelling from the sensor nodes to the sink, the code to be executed travels from the sink to the sensor nodes [203]. Because the agent code will typically be smaller than the data, this saves on bandwidth usage, which in turn may save energy. Furthermore, since the locus of control no longer solely exists at the sink, there is lower latency in interactions between nodes and there is increased tolerance for disconnection of parts of the network. Another advantage of using mobile agents is the ability to inject new programs into any node, from where they will migrate to the nodes where the processing is to be applied [33, §1].

However, there are several differences between a treatment of mobile agents in sensor networks and mobile agents in fixed, wired, data networks: [35, §III.C]

- The resource-constrained nature of a sensor network means that care must be taken over the decision of when to migrate an agent. The replication and distribution of agents is significantly more costly.
- The low cost of computation compared to the cost of communication in sensor networks means that an interpreted scripting language can be adopted rather than a compiled language. This makes it easier to provide a safe 'sandbox' in which applications can execute. Whilst this requires a lightweight interpreter for the language, experience has shown that the overhead of interpreting is negligible [150, §4].
- In a traditional data network, a typical application may just consist of a single agent migrating around. However, in a sensor network, an application is more likely to consist of a number of tightly collaborating agents. This impacts on the addressing scheme used by the agents, perhaps meaning that local peer-to-peer communication amongst neighbours needs to be supported rather than solely using direct addresses from a global namespace.

- The attitude towards security is often different in a sensor network. Sensor networks are typically under the control of a single authority. Hence, authentication is required upon code admission, but other security or safety checks can be reasonably overlooked.

2.3.2.1 Programming active sensor systems

An approach to abstracting the complexities of programming for the TinyOS operating system on resource-constrained sensor nodes is provided by the Maté virtual machine [149]. This virtual machine is designed to run on low-power sensor nodes. It is an interpreter for a simple, stack-based language, providing dynamic code-loading. The complexity of sending and receiving messages is hidden: ad-hoc routing is performed automatically. Maté programs are written in terms of 'capsules' consisting of up to twenty-four instructions. These capsules have the ability to self-transfer using a particular instruction, leading to the notion that they can be thought of as mobile agents. Although Maté's language is very primitive, efforts are underway to provide a high-level language such as TinyScript [148] which can compile to it in much the same way as Java compiles to bytecode.

However, the designers of Maté have noted three areas in which it is inadequate [150, §2.4]: (*i*) it is written with a single class of applications in mind and is not suitable for all kinds of sensor network application; (*ii*) programmers are forced to use synchronisation primitives to ensure concurrency safety between co-located applications; and (*iii*) code propagation policies, which are hard to express, must be coded by the programmer.

These difficulties have led to a proposal for a general architecture for implementing an *application-specific* programming model. The application-specific virtual machine (ASVM) architecture allows the programmer to produce a custom instruction set and set of events [150]. This approach encourages a 'vertical' application design process, whereby the programmer chooses what operations the virtual machine supports and then writes code for it. The advantage of this approach is that the programming interface can be pushed up to a very high level whereby single instructions give rise to large computations. This makes programs smaller, reducing the requirements for large program memory on the nodes, and reducing the communication cost of propagating programs.

Scylla is a virtual machine for mobile sensor network nodes similar in spirit to Maté, although designed for more powerful nodes [225]. It supports application migration between nodes and an on-demand compilation technique. Scylla applications consist of application code, fault handler code and a memory image carrying the application's state, all of which are transferred in the case of a migration. The fault-handling code is executed upon receipt of an application if it is deemed too expensive to execute, perhaps to cause the application to migrate onwards to a different node. The cost of executing code can be computed either by an application-provided estimate of the application's dynamic instruction count (and hence energy usage), or this can be estimated at run-time. Fault handlers are restricted to not use backward flow control so their dynamic instruction counts are guaranteed to be no more than their static instruction count.

SensorWare is another implementation of a mobile agent framework for sensor networks [34]. It employs a high-level scripting language based on Tcl, whose core interpreter is deemed to be lightweight enough for sensor networks although it is too big for the memory of a Mote. The programming model is to define events and event handlers. SensorWare scripts tend to look like state machines driven by events such as receiving a message from a peer, data being sensed or the expiry of a timer. An event handler may perform some computation which may trigger further events and communicate with other agents.

Agilla is another framework for programming and deploying mobile agents in a sensor network [75]. Agilla's distinguishing feature is to provide a local tuple space on each sensor node, into which agents can insert tuples and from which agents can extract tuples via pattern matching. This facilitates the sharing of data between agents on a node, which provides a simple means by which an agent can discover its execution context. The tuple data can be shared locally using instructions which allow an agent to access a neighbouring node's tuple space; data can be shared globally via geographic addressing of nodes. Agents can self-clone and migrate through the network, optionally carrying their state with them, but the tuple spaces do not migrate.

Smart Messages [132] is a similar library and run-time system based on Java to implement mobile agents in a sensor network. It implements the Spatial Programming paradigm [33], in which the network is viewed as a single address space in which resources are accessed using 'spatial references'. A spatial reference uses the expected location of a node of interest and the name of a property. For example, room1.camera may be used to name any node in location room1 having the property camera. In this way, naming is *content-based*, avoiding the use of physical addresses, enabling the framework to hide the networking details of discovering nodes and routing messages.

The SpatialViews framework [189] is built on top of Smart Messages, using a high-level programming language which hides the volatility of a mobile network from the programmer. SpatialViews abstracts a network as a set of virtual networks called 'spatial views'. A spatial view consists of a set of physical nodes which exist in a particular location, although the membership of this set may change over time. Each spatial view has some computation to be done; this computation migrates amongst the physical nodes of the virtual network. For example, a programmer may define a spatial view naming the nodes within a region r which provide a service called LightSensor. Then the programmer specifies code to be executed by each such node n in r such as n.read(). When executed, the framework will migrate the code into r, visiting each of the nodes in random order—potentially in parallel—then migrating back to the sink to deliver the result. To save energy, the programmer can optionally specify that not all nodes within the region need necessarily to be visited. For example, if there are two sensors located very close to each other, it may be acceptable to only sample one of them, yielding a 'best-effort' result. This is specified by defining a spatial granularity within which nodes are considered equivalent and a description of the 'quality of result' that the algorithm will yield for a given set of nodes. The framework establishes a trade-off between the quality of result and the energy consumption.

The designers of the Sensor Querying and Tasking Language (SQTL) [124] distinguish two classes of operations performed on sensor networks: querying and tasking. Querying is a synchronous mechanism in which the querier blocks until a response has been delivered; tasking is an asynchronous operation whereby sensors mutually co-ordinate to achieve an action. SQTL is a scripting language supporting both classes of operations. It is a procedural language with some object-oriented features. The language is interpreted by the Sensor Execution Environment (SEE) virtual machine which deals with incoming messages and transparently forwards all messages which are marked as being for the attention of multiple nodes. It also provides some location-awareness, so that applications can tell where the nodes they are executing on are positioned relative to other nodes.

Abstract Regions is a programming paradigm and library which aims to simplify the expression of applications which involve communications within local regions of a sensor network [244]. The programmer defines a 'neighbourhood relationship' which defines regions within the network. For example, the relationship might be 'within n hops' or 'within radius r' or 'the k neighbours with the best quality communication link'. Nodes may thus belong to several regions. The programming model is of sharing data amongst nodes within a region. This is done through accessing data associated with a known key from one or more nodes. A node can also perform a 'reduce' operation which performs an associative operation on all of the data associated with a particular key within a region.

Abstract Regions also exposes to applications the ability to tune the trade-off between reliability and energy consumption. For example, repeatedly sending a message makes it more likely that it will be received, at the cost of a higher energy consumption than sending it once. Also, varying the rate of broadcasting location advertisements affects the quality of region discovery. This trade-off is exposed by each operation returning a measure of 'quality', representing how completely or accurately the operation was performed, and the provision of a low-level interface to tune communications parameters.

The Hood system [245] is largely similar to the Abstract Regions paradigm. In Hood, attributes to be shared with other nodes are broadcast locally. Nodes which hear the broadcast can choose, based on a programmer-defined filter, whether to cache the attribute's value locally. This filtering decision being taken by the receiver means that there is an asymmetry between two nodes; nodes may hold different opinions on whether they are both part of a particular neighbourhood; there is no notion of a 'group', in which every node is aware of the membership of every other node. In contrast to Abstract Regions, Hood does not provide data aggregation facilities or any ability to manipulate the accuracy–resource trade-off.

Dynamic code upgrading. Some systems provide the facility to dynamically upgrade applications by diffusing new code through the network. This can be thought of as a particular instance of the mobile agents paradigm. The SOS sensor-node operating system provides this facility natively [100]. An application running on SOS consists of a number of interacting 'modules', which can be dynamically linked.

Similarly, the Contiki sensor-node operating system supports dynamic loading and replacement of applications (or parts of applications) [62]. Like TinyOS, the Contiki kernel is event-based (thus similarly avoiding per-thread stacks and the need for locking), but Contiki provides an application library to allow pre-emptive multi-threading, which simplifies the programming of long-running computation.

Impala [159] is a middleware layer which allows applications to be upgraded dynamically, in situ. It was originally designed for use with the long-running, large-scale ZebraNet mobile sensor network [128]. It also allows dynamic adaptability of applications to aim to minimise energy usage whilst maximising performance. For example, this adaptation might involve the decision to change to use a wider-range communications protocol if the device's battery level is found to be high and the number of other devices nearby is low. However, this adaptation is on a per-node basis and is driven by monitored sensor values or system properties.

2.3.3 Other approaches

In general, the distributed database and active sensor programming paradigms described above either involved writing a single program to execute on all nodes or writing separate programs for individual nodes. An alternative approach is to allow programmers to describe the behaviour of the system as a whole in a high-level declarative language.

Such an approach has been adopted by Mainland et al. [168]. They use a 'market-based macroprogramming' paradigm in order to organise the execution of applications in sensor networks. In this paradigm, nodes receive price information and 'sell' actions. Actions that contribute towards the network's overall goal result in a payment to the node performing the action. Each action constitutes the expense of some energy which is subtracted from nodes' pre-allocated energy budget. Examples of typical actions include sampling a sensor, aggregating data, forwarding a message and sleeping. With all nodes acting autonomously in their own self-interest, with no knowledge of any global aims, the behaviour of the network is controlled by modifying the prices of actions, rather than by directly instructing the node to perform a different computation. Nodes will select actions which maximise their utility given their current energy level and the availability of sensor data or the presence of pending messages to forward. Hence, 'programming' a sensor network using this paradigm involves expressing policies for how the prices should change. The authors describe this as an instance of the Self-Organising Resource Allocation (SORA) paradigm.

2.4 Summary

This chapter has surveyed extant research that is most closely related to the problem of programming large-scale networks of vehicles. Broadly, this has covered two areas: programming paradigms for distributed systems in general (Section 2.2), and for the specific case of sensor networks (Section 2.3).

In loosely-coupled distributed systems, there are a variety of approaches to managing communication between processes, ranging from high-level abstractions such as distributed shared virtual memory to low-level schemes such as message-passing (Section 2.2.2). The nature of task assignment is also a fundamental characteristic of a distributed system (Section 2.2.3). This is the problem of determining what computation should take place on which processors. A system which provides automatic task assignment relieves the programmer of the burden of needing to decide this in advance.

Wireless sensor networks are usually programmed by writing code for each node individually, using a language such as nesC for the TinyOS platform. The low-level nature of this approach has led to suggestions of programming sensor networks in terms of database-style queries (Section 2.3.1) or via injecting code into virtual machines running on the nodes (Section 2.3.2). The former approach suffers from a lack of flexibility; the latter approach is considered to still be too low-level.

Chapter 5 will present a programming paradigm which seeks to avoid these deficiencies by allowing programmers to describe applications at a high level and which exploits automatic task assignment.

CHAPTER 3

A vehicular sensor platform

This chapter concerns the practical feasibility of treating a vehicle as a sensor platform that could be a node in a sensor network. Such a platform has been implemented; various aspects of its design and implementation are described. This platform has been used to provide the sensor data for the application which will be described in Chapter 4.

3.1 Managing sensor data

Modern vehicles contain many sensors fundamental to the operation of the vehicle (see Section 2.1.2). In the future, it is conceivable that vehicles will carry sensors whose data is consumed by third parties.

This gives rise to the questions of when and where sensor data should be processed. For some applications, immediate processing of the data is required. For example, a vehicle's movement data must be shared with other vehicles if it is travelling in a platoon. For certain other applications, the processing of data can be delayed until a later time. This may be because it is not needed until later, or because it may be discarded later based on data subsequently received. Moreover, it may be infeasible to process some data in real-time, perhaps due to constraints on the speed of processing in the vehicle, such as may be the case for high-frame-rate video data.

Each application and each type of data will have different requirements. Hence, a vehicular sensor platform must support a means of multiplexing several applications, meeting the demands of as many as possible, providing the facility to both store data and process data in real-time. Since storage space is limited, strategies must be adopted for prioritising data and determining which data is least costly to discard in order to make room for newer readings.

An example of a vehicular sensor platform is the Instrumented Car [228]. This vehicle is equipped with a variety of sensors which obtain data about the driver, the vehicle's engine, emissions and its environment.

A vehicle with its own repository of data is useful for its owner, but the value of multiple similarly-equipped vehicles can be significant. This is reminiscent of the principle known as Metcalfe's Law which states that the utility of a communications network is proportional to the square of the number of participants. This means that vehicular data repositories need not contain data pertaining only to the host vehicle; it may be useful for some applications to also acquire data from other vehicles. Hence, a vehicle becomes a node in a vehicular sensor network (see Section 2.1.4.2). This requires a means of communication between vehicles (see Section 2.1.3) and gives rise to privacy issues as vehicles share their sensor data with others.

3.2 A sensor platform

To investigate these ideas, and to begin to realise the vision of vehicles producing sensor data and communicating to share it, we have prepared a vehicle equipped with computing and communication equipment. It acts as a general-purpose sensor and computing platform which forms the basis on which research into applications such as automatic map generation (to be described in Chapter 4) has been conducted.

We begin by outlining some practical requirements for its implementation in Section 3.2.1 and describe the internal sensor and computing infrastructure in Section 3.2.3. The section concludes with a consideration of practical issues such as power management (Section 3.2.5) and the means of user interaction (Section 3.2.6). Finally, Section 3.2.7 provides a taste of the data that has been collected.

3.2.1 Requirements

We identified various requirements for the platform that would allow it to be used as a normal vehicle that also collects data as it is used in a manner transparent to its driver:

Power. Because the vehicle is mobile, it is not practical to expect that it will have any permanent access to an external power supply, so it must be able to cope with extended disconnection. Thus, the computing equipment contained within it should exploit the vehicle's own means of power generation, making use of the available fuel resources.

Communication. In order to support a wide range of applications, a diverse variety of short-range and long-range communication interfaces are to be supported.

Transparent nature. The computer and sensing equipment must function in a fully autonomous fashion and be non-intrusive. The driver should not need to treat the vehicle any differently to any other vehicle.

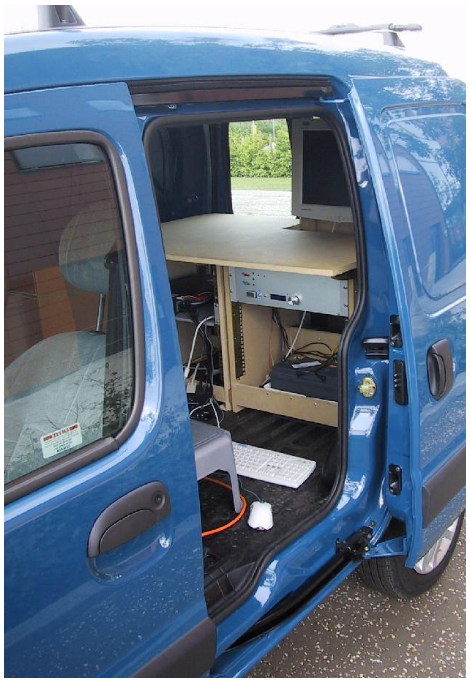


Figure 3.1: View through the side door of the vehicle, showing the placement of the equipment rack.

Several further requirements relate to the nature of the vehicle as a research platform:

Extensibility. It must be simple to install new sensors of any kind, as new research needs are identified or as new technologies become available.

Developer-friendly. The need to develop new applications and modify existing ones—even when the vehicle is away from its home—implies that it is necessary to have a comfortable means of interfacing with the computer.

Type of vehicle. The vehicle must be based on a car, so that it can travel in the same places and at the same speeds as the majority of road users, but should permit the easy installation of electronic equipment.

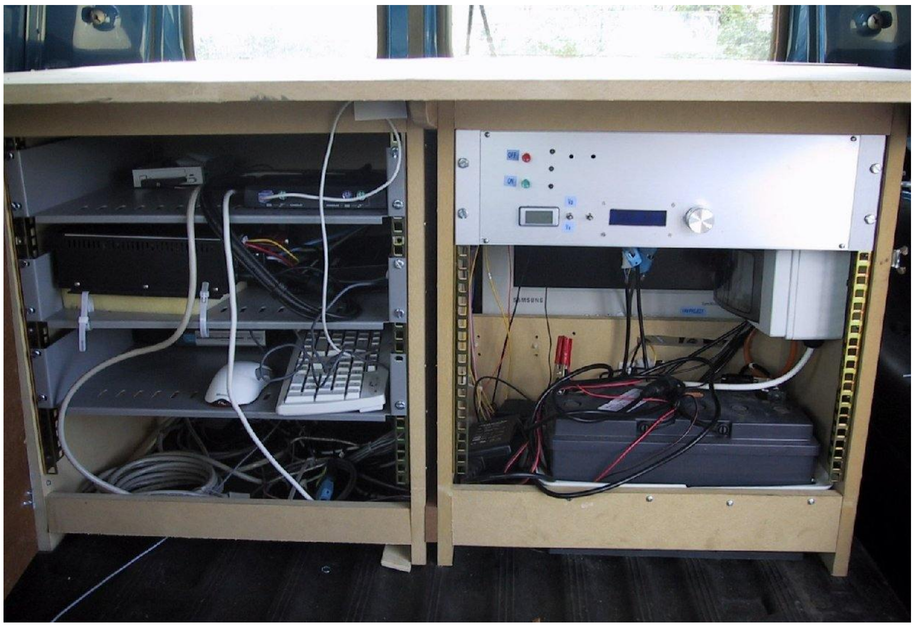
The subsequent sections indicate how these requirements have been met.

3.2.2 Vehicle

A Renault Kangoo van formed the physical basis for the platform. The vehicle's carry-space provided ample space to install the computing equipment, shown in Figure 3.1. The equipment is held in an equipment rack shown in Figure 3.2.

3.2.3 Sensor infrastructure

The goal of extensibility requires a wide range of sensors to be able to be installed with minimum effort, with their data being logged to permanent storage. This was achieved


(a) Front view, monitor in retracted position

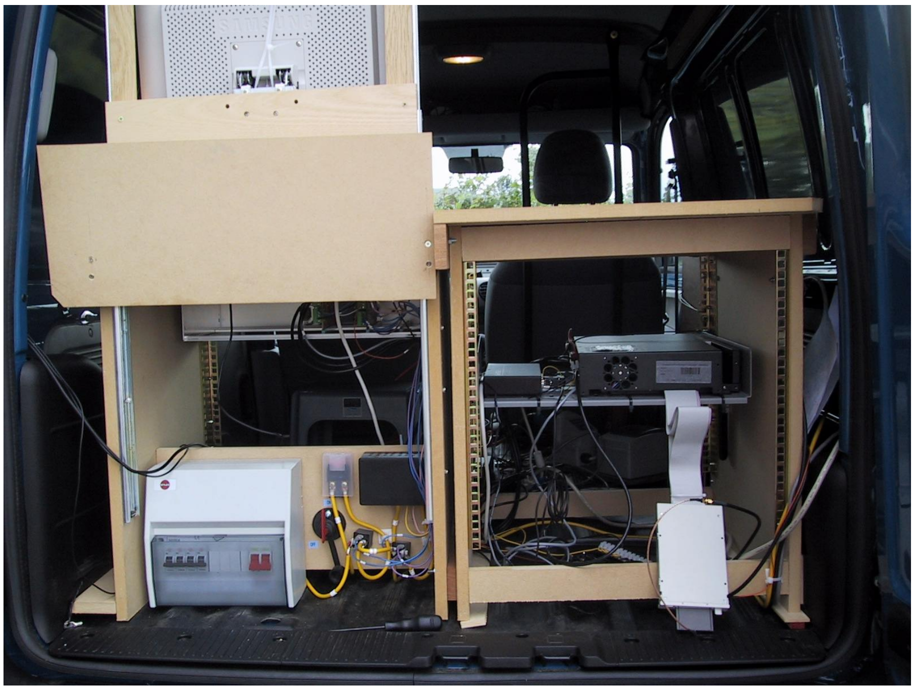

(b) Rear view, monitor in extended position

Figure 3.2: The equipment rack inside the vehicle.

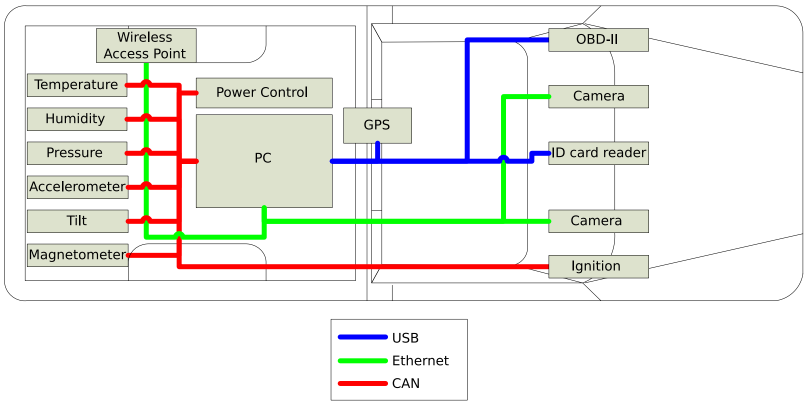


Figure 3.3: Overview of sensor infrastructure in the vehicle.

through the installation of three separate data buses and a power distribution network wired throughout the vehicle; see Figure 3.3 for an overview. Data from sensors attached to all of these buses is logged to disk on the embedded computer, which is a small form-factor, commodity PC based on the VIA EPIA motherboard.

3.2.3.1 CAN bus

Typically, modern vehicles contain their own Controller Area Network (CAN) bus, or a derivative thereof, to control many of the vehicle's sensors and actuators (see Section 2.1.2). The features of the CAN standard which make it particularly well-suited to in-vehicle sensing include:

- automatic retransmission that guarantees delivery of messages;
- dominant-recessive signalling meaning that collisions do not result in corrupted packets;
- differential signalling on two wires, meaning that the effect of electrical interference is minimised; and
- the widespread availability of CAN interface chips which work at a wide range of voltages and which are able to operate on a bus over a distance of a few metres.

A dedicated sensor CAN bus was installed, kept separate from the vehicle's own bus for safety reasons to guarantee that it would not interfere with the vehicles' devices. In particular, the addressing scheme of the existing nodes and the traffic density patterns were not known. The sensor CAN bus was wired in a modular fashion such that sensors could be added and removed with minimal disruption, inside and outside the vehicle.

The sensor CAN bus is connected to the on-board computer via a Lawicel CAN-to-USB converter, and a CAN daemon arbitrates between multiple separate processes to send and receive messages from the devices on the CAN bus. This is preferable to having each process try to separately control the CAN-to-USB device with no co-ordination, which may have led to the starvation of some processes from CAN bus access, and may not have coped with the interleaving of concurrent operations on different devices attached to the CAN bus.

Several groups of sensors are attached to the sensor CAN bus, including:

Meteorological sensors:	temperature, humidity, barometric pressure;
Position sensors:	two-axis accelerometers, tilt sensor, two-axis magnetometers;
Environmental sensors:	CO_2 sensor.

The magnetometers are orthogonally aligned in the horizontal plane and are useful to determine the orientation of the vehicle. When the vehicle is stationary, its orientation would otherwise be unknown since the heading would not be deducible from the displacement between consecutive pairs of position readings.

Further sensor deployment is facilitated by the re-use of the circuit layout for interfacing between sensors and the CAN bus used for each of these sensors.

3.2.3.2 Usb

To facilitate the installation of sensors that are capable of relaying their data over USB, and which have a higher data-rate than the CAN bus can support, a USB tree was wired throughout the vehicle.

For sensors supporting the RS-232 protocol, we attached them to the USB tree via AlphaMicro Components RS-232-to-USB converters. At present, the following devices are attached in this way:

- three GPS receivers;
- an RFID card reader with which to identify the driver of the vehicle by an identity card; and
- the OBD-II scan-tool to interface with the vehicle's diagnostics subsystem (described below).

Connecting these RS-232 devices over USB provided three benefits:

- there was no need for a further separate bus;
- an almost unlimited number of RS-232 devices could be connected rather than being limited by the number of physical ports, permitting future extensibility; and
- it meant that Linux's *udev* facility could be used to create static names for the RS-232 devices, by relying on the unique identifiers of the RS-232-to-USB converters rather than relying on the particular physical socket the devices are plugged into.

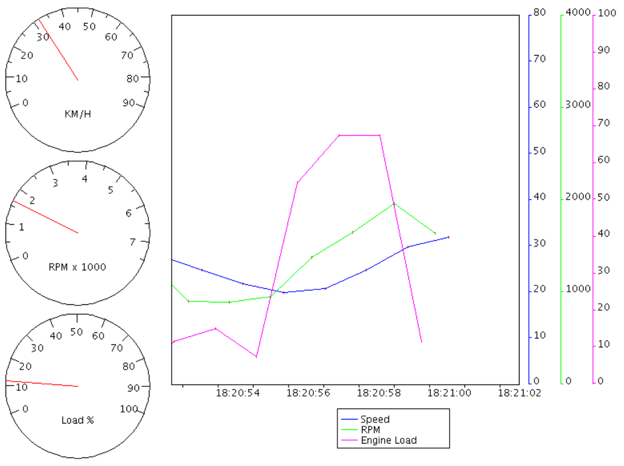


Figure 3.4: An application showing current and recent data from three sensors accessed via OBD-II.

The operating environment in the vehicle was found to exceed the capabilities of USB 2.0, by requiring a USB tree stretching over several metres. The symptoms of this that were observed included the devices at the leaves of the tree experiencing an erratic connection to the computer, sometimes failing to be detected, sometimes failing to initialise properly. This problem was solved by using USB 1.0—significantly decreasing the bandwidth of the bus but increasing its tolerance to greater distances. This was achieved by installing a USB 1.0 hub at the root of the USB tree, causing all of the descendant devices to fall back to USB 1.0 speeds. It was also found to be important that all USB hubs were externally powered, rather than consuming power from the bus, so that the deepest leaves of the tree would be assured of power.

Via the OBD-II interface, data from sensors in the vehicle's engine can be accessed. The data available over this port is vehicle-specific. In the vehicle used, the available data includes the engine speed (RPM), road speed, engine load (as a percentage of the peak available torque), air intake temperature, fuel rail pressure, intake manifold pressure, and engine coolant temperature.

Sensor data is obtained over the OBD-II interface by polling. The maximum rate at which data can be read from the OBD-II port is approximately 2 Hz. Unfortunately, this meant that it is not possible to sample the rapidly varying sensors as frequently as desired; for example, the engine's velocity and load can vary substantially over the course of half a second. Coupled with the fact that only one sensor can be polled at a time, it means that the data obtained via this interface is of limited quality. To minimise this problem as much as possible, the sensors are sampled at varying rates, matching as closely as possible the rates at which their values typically change. Figure 3.4 shows a screenshot of a simple application visualising data from a subset of OBD-II sensors.

3.2.3.3 Ethernet

In order to support media-rich sensors with high data rates, a 100 Mb/s Ethernet network was also installed. It has been used to stream video data from two cameras: one front-mounted and one rear-mounted. For such sensors with high data rates, it is best for them to perform as much processing as possible to relieve the on-board computer from the burden of this processing, to reduce the volume of network traffic. For example, on-camera MPEG encoding reduces the data rate from several megabytes per second to several hundred kilobytes per second.

Even despite this encoding, the data rate of the video is still very high. Thus, given a finite amount of storage capacity, it is important to prioritise what data is logged permanently. At a high frame rate, all available disk space could be consumed over the course of a relatively short journey. But it is common for pairs of consecutive video frames to depict largely the same information, so it is more space-efficient to only save frames with a significant difference to the previous saved frame; this observation was the inspiration behind MPEG encoding. This could be theoretically achieved by dynamically processing each pair of frames and determining whether the difference is large enough according to some heuristic, but this processing would be very computationally expensive. Instead, we observed that the probability of the occurrence of a significant event is related to the speed of the vehicle: when travelling fast, greater distances are covered so successive frames are very different; at very slow speeds, the opposite is true. The rate at which video frames are saved is thus adapted based on the vehicle's speed. In general, the decision about whether to save an image could be based on inputs from any on-board sensors, as is the case with the SenseCam system [112].

3.2.4 External communications

A variety of wireless networking equipment is employed in the vehicle using high-gain, roof-mounted external antennas.

An IEEE 802.11b/g interface has been used to communicate with an access point up to 300 m away. Experiments have found that line-of-sight is important. An IEEE 802.11a interface is also used and requires line-of-sight to the access point. Experiments have found that the maximal range is less than that for 802.11b/g. Furthermore, a GPRS/UMTS datacard provides near-ubiquitous Internet connectivity at a rate of up to 300 kb/s.

As well as acting as communications interfaces, these devices also act as sensors which evaluate the strength of the signal which they receive. This data is treated in the same way as any other sensor data by logging it.

The vehicle automatically detects when it is within range of its home WiFi network and uploads newly collected data to a fixed server. This allows in-vehicle storage space to be reclaimed for use on a future journey.

3.2.5 Power

Careful management of power is necessary in a vehicular environment, since the mobility of vehicles implies that they cannot rely on any permanent connection to an external power source. On the other hand, vehicles generate electricity through their alternators, which may be exploited to power on-board equipment. Further challenges are also raised by the requirement of transparency, meaning that the presence of the equipment should not distract the driver from any normal driving habits.

The following goals were identified for a vehicular sensor platform's power infrastructure:

- avoiding flattening the vehicle's battery, which would incapacitate the vehicle;
- making use of the vehicle's alternator to provide the power to run the equipment;
- starting the computing equipment and logging sensor data automatically when the ignition is started;
- turning off the computer and sensors automatically when the sensor data has been uploaded after the ignition is turned off;
- having a failsafe power cut-off facility to ensure that the computer will not consume energy indefinitely when the vehicle is parked; and
- having the facility to connect the vehicle to an external power source in order to recharge the battery that powers the sensor equipment when possible.

The on-board equipment is powered from an auxiliary battery, separate from the vehicle's main battery. This was preferred to using the vehicle's battery as it could not be expected to supply a steady 12 V, as required by the on-board computer, and also enabled us to be confident that we could never flatten the main battery by over-use of the equipment.

Circuitry was developed to use the vehicle's alternator to charge the auxiliary battery in parallel with the vehicle's main battery, but with a limit set on the charging current to avoid overloading the alternator if the auxiliary battery should be particularly low.

Power control circuitry was implemented that powered the computer and sensors automatically when the ignition is started, causing the computer to start logging sensor data; and to send a signal to the computer indicating that the computer should shut itself down, when the ignition is turned off.

A hardware failsafe power cut-off was implemented which is triggered if the computer does not shut itself down within a few minutes, to ensure that if the computer has inadvertently entered into a stuck state then it will not drain the battery. This was preferred to a watchdog scheme—whereby the computer would be expected to send regular messages requesting that the power be kept on—since this would not provide the same degree of guarantee that power would indeed eventually be cut.

Figure 3.5 shows an overview of the power control scheme which satisfies the following usage cycle:

1. the driver starts the ignition; computer and sensors automatically turn on;
2. the driver drives the vehicle whilst the sensor data is logged to disk;
3. the driver turns off the ignition and leaves the vehicle;
4. the data collected is uploaded if the vehicle is within range of a suitable wireless network; and

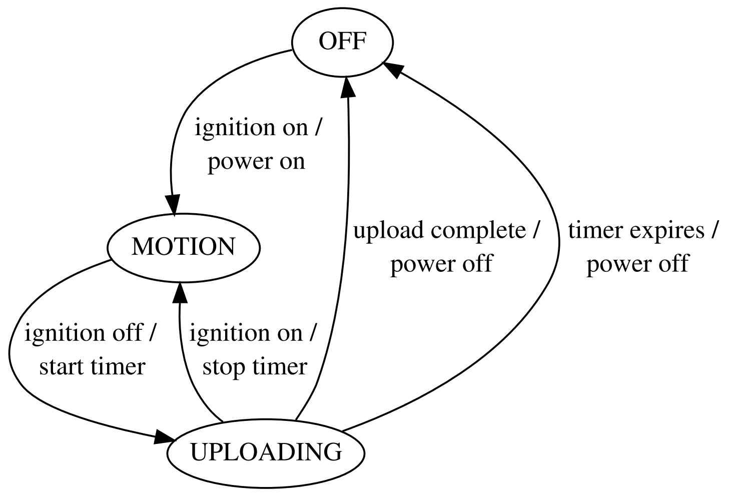

OFF
ignition on / power on
MOTION
upload complete / power off
timer expires / power off
ignition off / start timer
ignition on / stop timer
UPLOADING

Figure 3.5: State machine representing the power control scheme (edges are labelled *input / output*).

5. the computer shuts down gracefully when the data upload is complete or it has been idle for a certain period of time.

In order that the computer equipment could be left running when the vehicle is parked at its home location without draining the battery, a means of connecting it to an external mains power source was implemented. When this is connected, the failsafe power cut-off mechanism is overridden.

3.2.6 User interaction

In the front of the vehicle, two windscreen-mounted 8” LCD touchscreen monitors, one on the driver’s side and one on the passenger’s side, provide means of user interaction, shown in Figure 3.6. These monitors can be run in identical, independent, or shared desktop modes, to permit the passenger to interact with the computer and optionally control what information is displayed on the driver’s monitor. This was implemented by hosting two desktops in local VNC sessions, as shown in Figure 3.7, and using a modified VNC client to allow specified portions of the desktop to be displayed.

As is common in many modern vehicles, there is a stalk of audio controls mounted on the steering column, shown in Figure 3.8. This stalk has been modified to allow it to act as an input device to the computer, emulating a motionless six-button wheel mouse. This facility permits the driver to control applications in a similar fashion to the vehicle’s other functions.

A comfortable development environment is also necessary for deployment and debugging purposes. This was achieved by converting the main carry-space of the van into an area where developers can interact with the computer when the vehicle is stationary. A 17” LCD monitor, keyboard and mouse are provided to mimic a typical office working environment. The monitor is fixed onto telescopic vertical rails so that when it is not in use it is out of sight. It can be seen in Figure 3.2b.

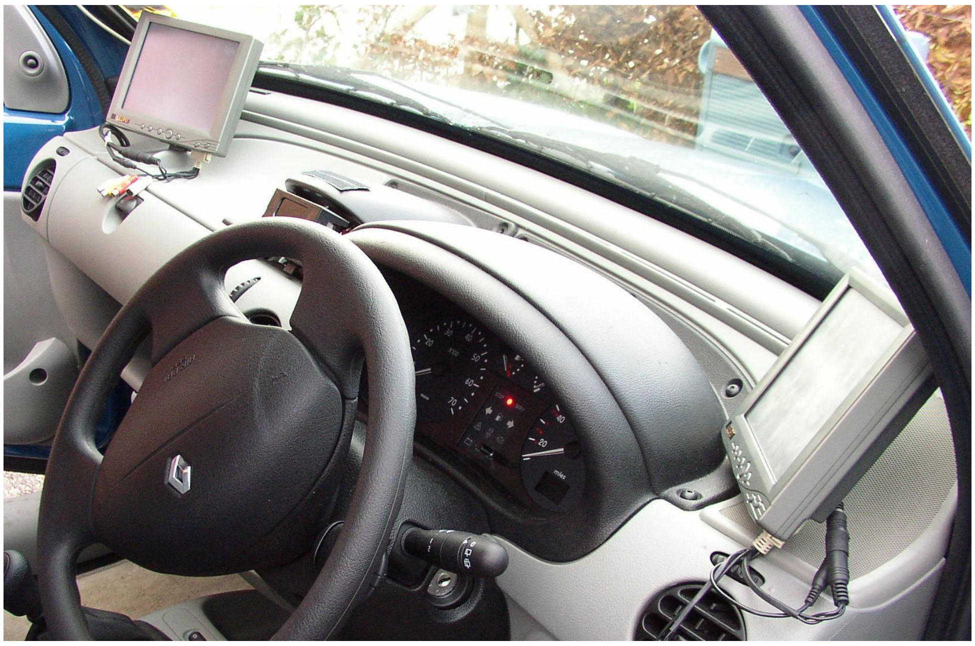


Figure 3.6: Two LCD touchscreen monitors mounted on the dashboard.

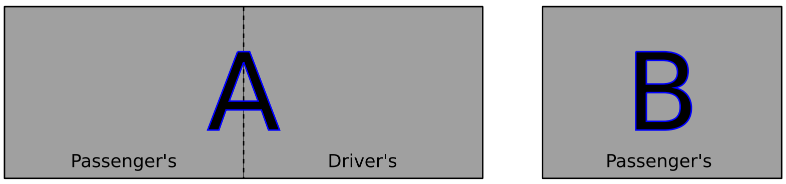


Figure 3.7: Two VNC desktops, with A twice the width of B. Desktop A is divided into two halves, one for passenger and driver each. Desktop B is the passenger's independent desktop.

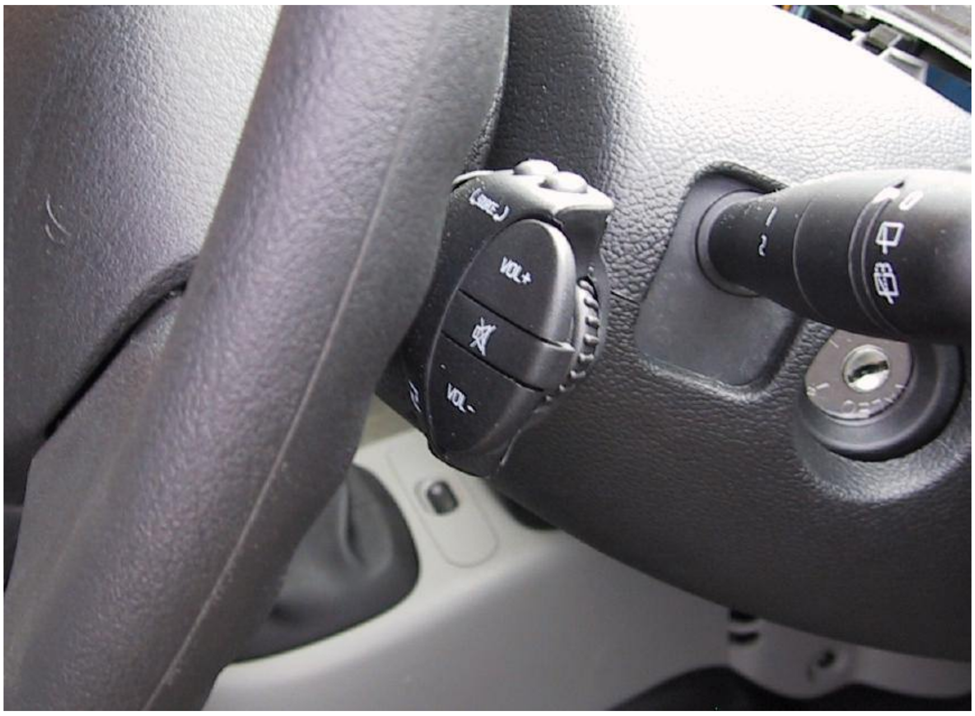


Figure 3.8: The audio control stalk mounted on the steering column.

3.2.7 Data collected

At the time of writing, over fifty million sensor readings have been collected, including a third of a million images. These are broken down into categories in Table 3.1. Since the vehicle is in active use, these numbers are still rising.

Figure 3.9 shows a histogram of temperature values recorded by the thermometer, grouped into 1°C buckets. This data reflects the ambient temperatures at the times at which the vehicle has been driven, and thus contains daily and seasonal bias.

Figure 3.10 shows a map of CO_2 values recorded over the region around the city of Cambridge, UK. Each CO_2 reading is associated with the geographic position at which it was sampled by interpolating readings from the GPS sensor. This data is then grouped into two-dimensional cells and averaged before being rendered on top of a map produced from OpenStreetMap data.[1]

3.3 Further work

Further research into vehicular sensor platforms could involve the following:

- An interface could be devised to abstract the sensors which may be present on a vehicle. This should be sufficiently general to provide support for all kinds of sensors,

[1]The image contains OpenStreetMap base map data and is classed as a derived work that may be distributed under the Creative Commons Attribution-Share Alike 2.0 license, `http://creativecommons.org/licenses/by-sa/2.0/`. Base map data is © 2002–2008 OpenStreetMap Contributors.

Data type	Values
Location	3 441 849
Horizontal orientation	5 062 123
Three-axis acceleration	9 784 737
Wireless signal strength	23 566 778
OBD-II readings	3 843 545
Humidity	891 889
Temperature	893 150
Barometric pressure	2 463 960
CO_2 concentration	4 147 651
Images	355 294
Total	54 450 976

Table 3.1: Number of recorded sensor readings.

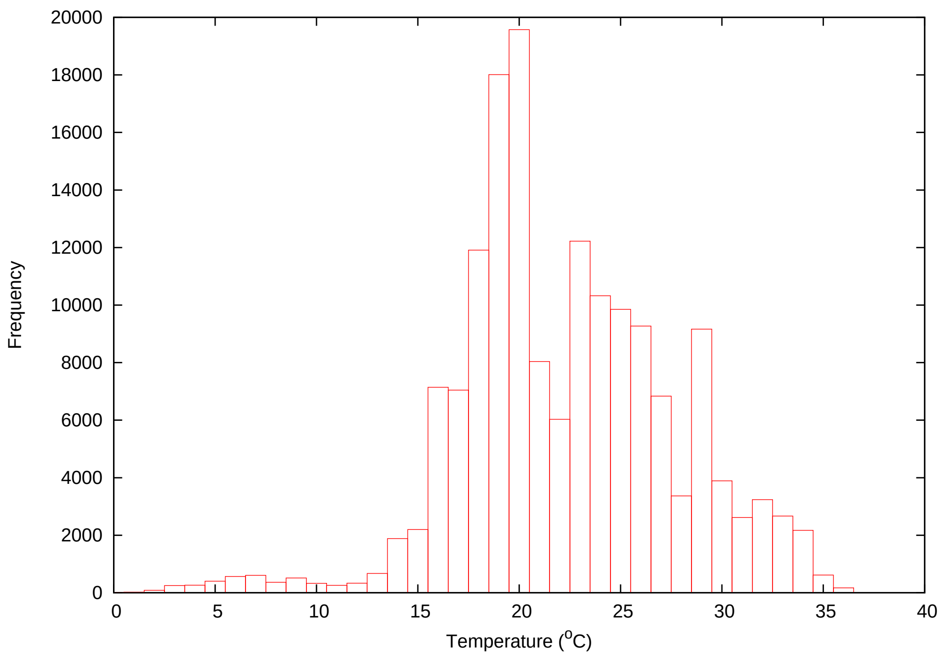


Figure 3.9: Histogram of temperature data sampled by the thermometer.

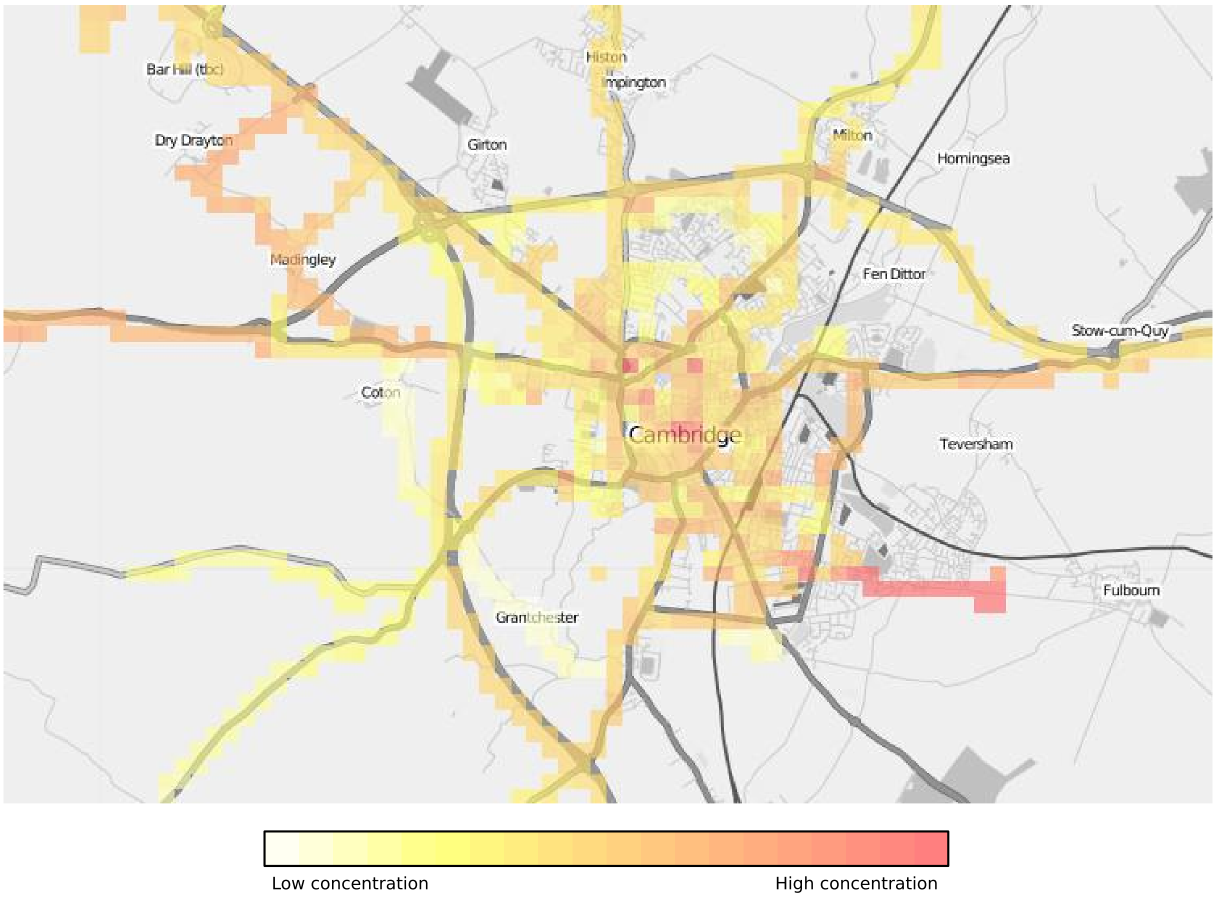


Figure 3.10: CO_2 data plotted on a map of Cambridge.

including those not available or not commonly fitted in vehicles today. This interface would provide a basis on which in-vehicle applications could be implemented, meaning that applications would become portable to any vehicle even if it supported different numbers or models of physical sensors.

- A protocol could be implemented for the exchange of data between vehicles. This would provide a means by which vehicles can learn information gathered by vehicles which have travelled in other parts of the road network. This exchange of data need not necessarily be direct between vehicles and may be supported by centralised or regional data repositories. Investigation will be required into how the privacy of the originators of the data can be preserved, and into appropriate techniques for how older data can be aged and eventually discarded.

3.4 Summary

A vehicular sensor platform requires a means of sensing and storing data from a variety of sensors. These sensors may have different characteristics which influence how the sensor is interfaced with, and how its data is collected and stored.

The design of a research platform for vehicular sensing has been described. Amongst other things, careful consideration is required for how the supply of power is managed. An implementation of this design has, to date, produced over fifty million sensor readings from a wide variety of sensors. Some of this data will be used in the application that will be presented in Chapter 4.

CHAPTER 4

Scalable, distributed, real-time map generation

This chapter describes an example of the inference of higher-level information from raw sensor data collected from vehicles.

Arguably the most important sensor to include in any mobile platform is one which yields the platform's location, since it is the variation in location that defines mobility. The vehicular sensor platform described in Chapter 3 contained several GPS receivers with which to determine its location.

Data from any other sensor can be associated with the location at which it was sampled to enable it to be spatially indexed. However, information about the vehicle's location is not solely useful to provide context for other data; it is valuable in itself. By inspecting the locations the vehicle has visited and those it has not, we can infer higher-level information about the behaviour of the vehicle's drivers and the nature of the road network in which it has travelled.

In this chapter we will consider in turn:

- what information about the nature of the road network can be inferred from location data (Section 4.1);
- how to maintain an up-to-date model of the road network (Section 4.2); and
- the value of using location data from multiple vehicles (Section 4.3).

A discussion of the implementation of this application will be postponed until Chapter 7.

4.1 Inferring a road map from location data

Information about a road network can be represented as a directed graph with metadata associated with its edges. The edges in the graph represent roads (or road lanes) and the vertices in the graph represent junctions. Such a graph can be used to both render a graphical depiction of the road network and serve as an input to in-vehicle navigation systems.

4.1.1 Producing road maps

The map data in a vehicle's navigation unit forms the basis of its routing decisions. However, this data can be prone to cartographic errors and inaccuracies due to recent changes in the road network, including temporary road closures, which may make it differ from reality. This can cause frustration for drivers as they are given an instruction that is not possible to follow. Consequently, organisations which produce digital maps, such as Navteq and Tele Atlas, must invest significant effort in maintaining and improving the accuracy of their data. Details about the creation of new roads and the modification or closure of existing roads must be incorporated into the databases in a timely fashion.

Data about changes to the road network are obtained by mapping organisations from various sources, including local authorities and building contractors, but these data tend to be highly inaccurate. Aerial photographs can be used to deduce the presence and shape of roads, but are prohibitively expensive to update frequently. Instead, mapping companies typically own fleets of probe vehicles which can be used to investigate discrepancies or explore new roads. Tele Atlas spends tens of millions of dollars each year in North America to keep its databases up-to-date, while in 2004 Navteq employed over 500 analysts who drove a total of 3.5 million miles throughout North America and Europe.

In this chapter, an algorithm is proposed which aims to keep digital maps up-to-date without the need for fleets of dedicated probe vehicles.

4.1.2 Automatically generating a directed graph

Automatically generating descriptions of the environment purely from records of location fixes in an indoor environment has been investigated by Harle [102]. However, whilst it is unrealistic to expect exhaustive coverage of a typical indoor environment, we can expect vehicles to exhaust all available road-space, because the road network imposes greater constraints on the movement of users.

This section presents a map generation application which transforms a set of location traces from multiple vehicles into a road map. A location trace is a temporally-ordered set of positions in three-dimensional space which a vehicle has visited. Processing occurs in several stages:

1. generating a two-dimensional histogram indicating the number of location fixes that were found in each part of space (from an overhead view);

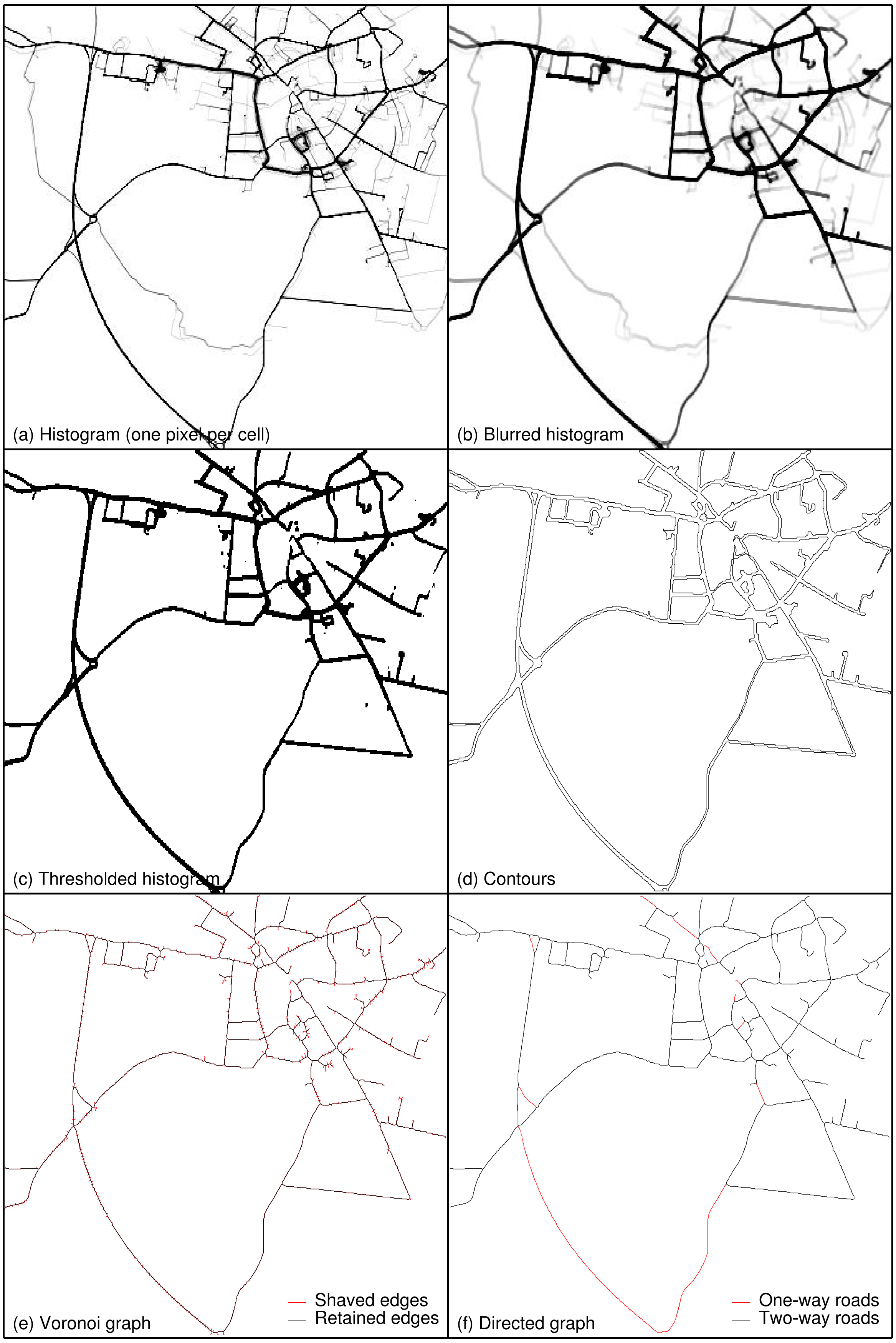


Figure 4.1: The stages in the generation of a map of the city centre of Cambridge, UK.

2. deducing the positions of the edges of the roads;
3. computing the positions of the centrelines of the roads; and
4. determining whether roads are uni- or bidirectional.

We now examine these stages in detail, using the generation of a map of Cambridge in Figure 4.1 as a running example. An evaluation of the algorithm will be given in Section 4.1.4.

A trace of locations obtained from a GPS unit in a single vehicle will show which roads the vehicle has travelled along, albeit with some errors introduced by uncertainty in the location fixes and missed sightings when the view of the sky is obscured. From a single trace it is not possible to distinguish between junctions and bends in roads unless the vehicle crosses its own path. Moreover, the errors inherent in the GPS readings may mean that the trace misrepresents the true positions of the roads. However, if we superimpose the traces from several journeys on different routes, junctions will soon become apparent, and the true positions of the roads will become clearer as noise due to errors becomes less significant.

4.1.2.1 Creating a histogram

Splitting up two-dimensional space in the horizontal plane into small, tessellating units of area or *cells*, we wish to determine the likelihood of each cell constituting part of a road. Whilst cells do not necessarily have to be square, we will assume that this is the case for simplicity. The Nyquist-Shannon sampling theorem dictates that the cell width should be at most half the minimum road width in order to prevent aliasing. In practice, this equates to a few metres. A GPS reading falling in a cell is a good indication that the cell may be part of a road, so if we associate with each cell a count of the number of GPS points that fall in it, cells with high counts will be those that are most highly believed to be parts of a road. In this way, we group our two-dimensional real-valued GPS fixes into discrete cells.

Discretising the space over which sensor readings are made into a grid of cells is established practice. In the area of robotics, *certainty grids* [181] or *occupancy grids* [65] are used to store the probability of there being an obstacle in any particular cell in robotic environment exploration [231]. This differs from our work in that we do not have the luxury of being able to determine the presence of obstacles as well as their absence. Furthermore, map data intended ultimately for human consumption may tolerate lower accuracy than robotic control.

At motorway speeds, if GPS fixes are obtained at a frequency of 1 Hz, consecutive fixes will fall about thirty metres apart. Thus, with a cell size of a few metres, successive fixes will not lie in adjacent cells, leaving us with disjoint regions of motorway. However, if the GPS fixes are temporally-ordered, we know that road must exist between consecutive fixes, so we can also increment the count in the intermediate cells between those in which the fixes lie. If the frequency of readings is greater than a few hertz then linear interpolation between the GPS fixes is likely to be acceptable. However, higher-order interpolation may yield more realistic results.

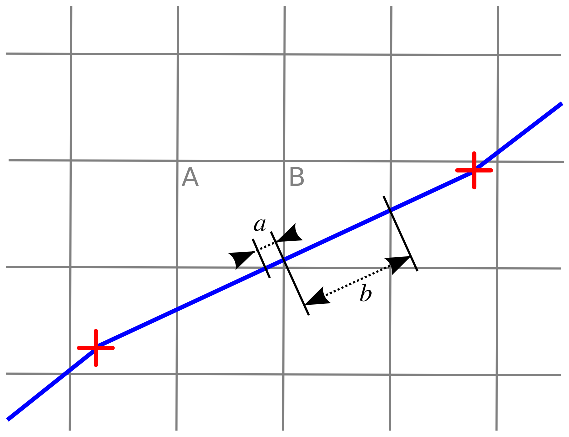


Figure 4.2: A line interpolating two GPS fixes. The value in cell A is incremented by a smaller amount than the value in cell B because $a < b$.

We want the value in each cell to represent the confidence of the cell being part of a road. Rather than simply incrementing in units, the value of a cell is incremented by an amount proportional to the length of the line which passes through the cell between successive location fixes. In this way, if the line only traverses the corner of a cell then the value in that cell is only incremented by a small amount. For example, in Figure 4.2, the value in cell A is incremented by a small amount proportional to a and the value in B by a larger amount proportional to b.

After all of the available GPS fixes have been processed in this way, we have a two-dimensional 'histogram'[1] which estimates the confidence of each cell constituting part of a road, based on all of the journeys traced out in the data. Figure 4.1a shows an example of such a histogram, with one pixel per cell and the shade of the pixel related to the value in the cell.

Even despite the interpolation between successive GPS fixes when preparing the histogram, it is likely that there will be gaps where cells with a low value may be surrounded by cells with high values. This may be due to either:

- a random paucity of data collected in those cells;
- systematic errors intrinsic in the original GPS data;
- real-world features where vehicles cannot travel such as lamp-posts or barriers in the centre of a road; or
- blackspots where there is no GPS coverage such as under bridges.

In any case, we are aiming for a directed graph of the topology of the road network rather than a detailed picture of the shape of the road, so these gaps are undesirable and should be removed.

[1]Whilst we will continue to refer to it as a histogram, it is not a genuine histogram as the values associated with the cells are no longer merely frequencies.

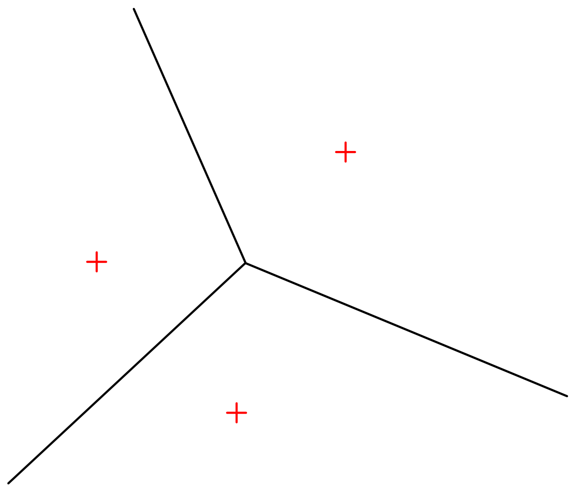


Figure 4.3: A Voronoi graph (black) generated from three sites (red). All points on the edges of the Voronoi graph are equidistant from the nearest two sites. Each line goes to infinity.

We can remove such gaps in the histogram by applying a blur filter to the histogram. Figure 4.1b depicts the histogram convolved with a 3 cell by 3 cell uniform blur convolution filter. Small gaps will be removed as cell values are averaged with neighbouring cells whilst larger gaps will persist. Similarly, jagged boundaries will be smoothed. However, it is possible that performing a blur introduces error since it carries the danger of undesirably merging two nearby but distinct roads.

4.1.2.2 Deducing positions of road edges

At this stage in the processing we have a good idea of whether a road exists in any given cell and we binarise this metric to a boolean value: in each cell, is there a road or not? This is achieved by applying a global threshold to our cells. The particular value chosen for the threshold will relate to the degree of confidence we want to have in the graph of the road network which is deduced from the algorithm. A lower threshold will be more susceptible to noise introduced due to GPS errors; a larger threshold may cause certain roads to be overlooked. The value chosen for the threshold is thus determined empirically based on the desired trade-off. The result of thresholding the data in the example is shown in Figure 4.1c.

After thresholding, a contour follower [255] is applied to the image. This extracts a set of closed polygons describing the outline of the regions of road, as shown in Figure 4.1d. This outline may not coincide precisely with the real-life edges of the roads, due to errors in the original GPS data. However, if these errors are symmetrically distributed then the centreline between the edges will coincide with the real-life centreline of the road.

4.1.2.3 Computing centrelines

For a given set of points, a Voronoi graph is the locus of all points which are equidistant from the nearest two points [15]. The points are referred to as *sites*. See Figure 4.3 for

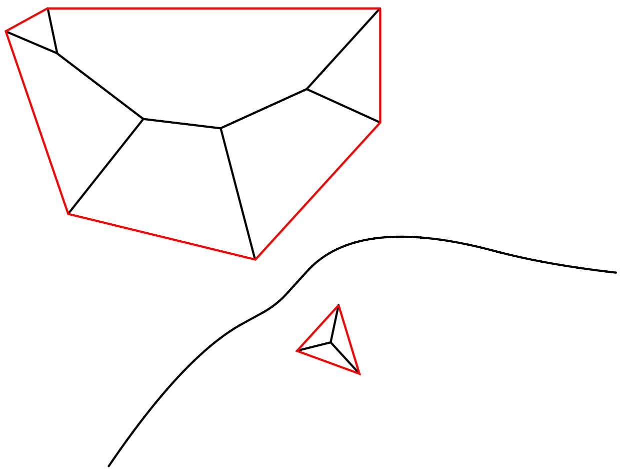


Figure 4.4: A Voronoi diagram (black) generated from a set of closed polygons (red).

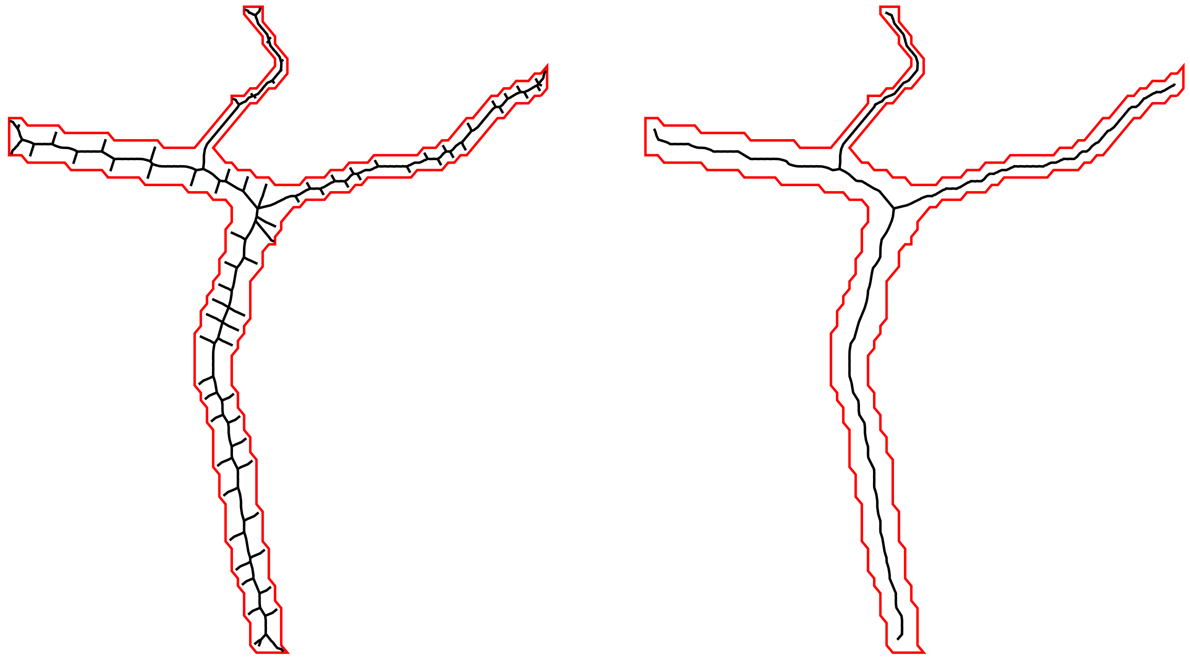
an example. Nodes of the Voronoi graph correspond to positions equidistant from the nearest three points. In the example, there is one node and three edges which stretch to infinity.

This concept can be extended to a diagram which is the locus of points that are equidistant from the nearest two points on the boundaries of a set of closed polygons. See Figure 4.4 for an example. Topologically, the diagram is similar to the Voronoi graph, having nodes where the nearest three boundaries are equidistant. We will treat the diagram as a graph in which each edge has an associated shape, and will refer to it as a graph. In the literature, it is sometimes referred to as a *skeleton* and is similar to the geometric concept of a *medial axis*.

Voronoi graphs are used in a wide range of disciplines, from the natural sciences and mathematics to computer science [15, §1]. In particular, they are heavily employed in robotics [48], because they describe the topological features of an environment, making them suitable for use in route-planning.

Voronoi graphs are naturally suited for use in computing road centrelines. We produce the Voronoi graphs of the contours describing the edges of the roads and discard the resulting edges which lie outside the roads.

However, because the edges of the roads are not convex, there will be many short edges of the Voronoi graph attached to the main trunk, along each road, giving the graph a rather 'hairy' appearance, as can be seen in Figure 4.5a. These edges do not correspond to real-life roads as they are artefacts resulting from our initial discretisation of GPS fixes into square cells causing the road edges to be stepped rather than smooth. We can remove the edges which are shorter than a threshold representing the minimum permitted absolute road length, leaving just the main backbone edges running the lengths of the roads. The result of removing these edges can be seen in Figure 4.5b. Figure 4.1e depicts the Voronoi graph generated from the contours in the running example, with the short edges which are discarded indicated in red.


(a) A 'hairy' Voronoi graph

(b) Voronoi graph with artefacts removed

Figure 4.5: Removal of artefacts.

4.1.2.4 Computing road direction

The Voronoi graph is an undirected graph of the road network. Edges of the graph correspond to road centrelines and nodes correspond to junctions.

The final stage of the algorithm is to deduce which edges of the undirected Voronoi graph represent unidirectional roads and which represent bidirectional roads. Since the original GPS data are available in temporally-ordered form, we can produce a further data structure which will help us determine this. Having again split space into cells, in each cell we now keep track of the directions vehicles travel in as they pass through the cell. This is done by counting the number of journeys embodied in the GPS traces which pass through the cell in each of eight directions of the compass. This is generated by quantising the bearing of the displacement vector between each successive pair of fixes and incrementing the count associated with that direction.

Using this data structure, we can now determine whether the journeys along the roads in the undirected graph are uni- or bidirectional. Roads containing cells in which vehicles travel in opposing directions (e.g. both north and south) are deemed to be bidirectional. Figure 4.6 exemplifies this approach: at two places on the portion of road shown, we examine the cells through which a line perpendicular to the axis of the road passes. If, on the majority of such lines, the cells contain opposing directions, the road is deemed to be bidirectional. Figure 4.1f shows which edges were deduced as uni- and bidirectional in the running example.

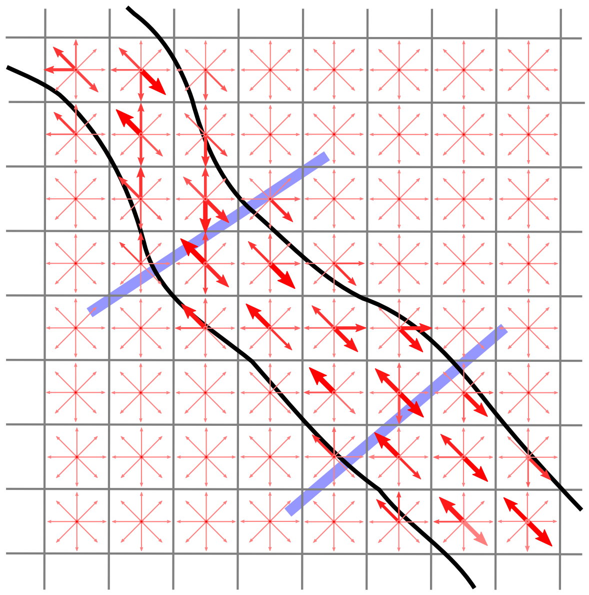


Figure 4.6: Determining whether a road is uni- or bidirectional. The sizes and shades of the arrows in each cell relate to the number of journeys travelling in those directions through it.

4.1.3 Complexity

The overall time complexity of this algorithm is $O(n + m \log m)$, where n is the number of GPS readings and m is the number of cells in the histogram. Individually, the stages involving the processing of GPS readings are $O(n)$. To blur or threshold a histogram, one operation is required for each cell and hence they are $O(m)$ operations. To find road edges, a contour follower also performs a raster scan across the cells; again $O(m)$. The resulting polygons will be bounded by $O(m)$ in size. Producing the Voronoi graph of these polygons can be achieved in $O(m \log m)$ time, and removing the resulting artefacts in $O(m)$.

However, if we are able to store the latest versions of the histogram and direction data, the time complexity of generating a new map which incorporates an additional set of GPS readings of size δn is merely $O(\delta n + m \log m)$.

4.1.4 Evaluation

In this section we evaluate the fundamental performance limitations and performance characteristics of the algorithm and then describe a practical experiment to investigate the algorithm's efficacy with real-world data collected from the sensor platform described in Chapter 3.

4.1.4.1 Fundamental limitations and performance characteristics

Errors in GPS readings are typically modelled by a bivariate normal distribution. The standard deviation, σ, in the error of the position estimate for a modern GPS receiver has

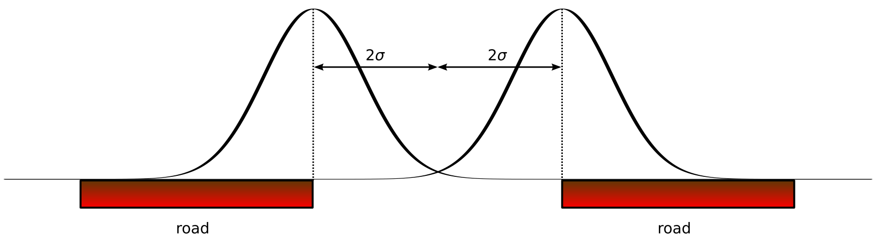


Figure 4.7: Cross-sectional view of two roads separated by 4σ, showing the normal distributions of position fixes of vehicles travelling along the edges of the roads.

recently been estimated at 4.25 m, giving a 95% confidence interval of 8.5 m [202]. Others have estimated the value to be 3.5 m [172]. We believe that a reasonable minimum distance between the centrelines of two adjacent parallel roads should be 4σ, so that no more than approximately 2.5% of the position estimates of vehicles at the edge of one road could overlap with no more than approximately 2.5% of those of vehicles at the nearest edge of the other road. This condition is depicted in Figure 4.7. If this limit is not observed then the region between the roads may be filled with stray GPS fixes and thus there may be no discernable gap between the roads after thresholding. For example, with $\sigma = 4$ m, the minimum tolerated road spacing is approximately 16 m.

However, in practice, this model of errors is over-simplistic. In certain areas—especially urban environments with tall buildings—multi-path effects are common, causing systematic rather than random errors as a position is consistently reported incorrectly. We have experienced occasions where a GPS unit has unrepeatably reported a series of positions consistently offset from the true path by a significant distance. If line segments generated from these traces were to be incorporated in the resulting map, we would be led to believe that there were roads where none exist. However, since these occurrences are rare, they can be excluded from the map by setting a sufficiently high binarisation threshold.

Many modern GPS units incorporate some form of additional positioning assistance. Augmenting technologies such as the Wide Area Augmentation System (WAAS) and the European Geostationary Navigation Overlay Service (EGNOS) provide additional radio signals which report the accuracy of the GPS signals; these reduce the value of σ and thus permit the cell size to be shrunk whilst maintaining the same level of confidence in each cell's status. The Galileo system will reduce the errors further when it is introduced. Dead reckoning units are common in vehicular navigation units and make use of a set of perpendicular accelerometers, integrating the outputs with respect to time to give accurate velocity and hence displacement measurements. As well as reducing σ, this form of assistance allows the unit to continue to deduce position fixes even when no satellites are in view. This means that maps generated from data using dead reckoning units will include tunnels and tree-lined avenues where a view of the sky is limited.

The distribution of GPS readings for vehicles travelling in all lanes of a road is unlikely to be normal but instead may be approximated by a multi-modal distribution made up of one normal distribution per lane. An example of a bimodal distribution for a two-lane road is shown in Figure 4.8. However, if we assume that the underlying mean of this distribution is on the centreline of the road (roughly that the volume of traffic is evenly distributed

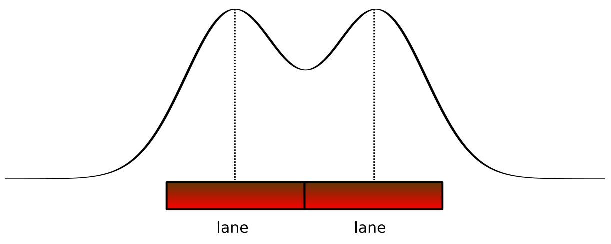


Figure 4.8: Cross-sectional view of a road with two lanes, showing a bimodal distribution approximating the distribution of location fixes for vehicles travelling on it.

about the centreline) then, by the Central Limit Theorem, the error of the mean of the GPS readings, when compared with the real centreline of the road, will be normally distributed and its standard deviation will decrease at a rate of $1/\sqrt{n}$ for an n-fold increase in the number of samples. Therefore, with sufficient samples, GPS data becomes an accurate predictor of the real road centreline. More specifically, the distribution of GPS samples collected from a two-lane road can be modelled as a distribution $(N(\mu_1, \sigma^2)+N(\mu_2, \sigma^2))/2$, where μ_1 and μ_2 are the positions of the centrelines of the lanes, and where the mean is the centreline of the road. By the Central Limit Theorem, with 73 samples, the estimate of the centreline of a road with the centres of the lanes 3 m apart will be within 1 m of the true position, 95% of the time.

High GPS sampling rates are desirable, and ideally the sampling rate should be such that when there is an abrupt change of direction, consecutive position fixes are no more than one cell width apart. This corresponds to a frequency of v/w for maximum cornering speed v and cell width w. For a cell width of a few metres, a sampling rate of 1 Hz is typically sufficient for adequate performance. With lower frequencies, if the samples are linearly interpolated, the change of direction would not be as sharp as in reality. When a vehicle is travelling rapidly, the distance between consecutive fixes will be larger, but abrupt changes of direction are not possible due to physical limits on lateral acceleration. In order to decrease the volume of GPS data collected by a vehicle, a strategy could be adopted such as only recording points when there is a substantial change of direction.

Roads which feature little in the journeys in the GPS traces will not appear in the generated map if they fall under the threshold and thus will not be included in the generated map, as they are not distinguishable from erroneous road segments.

With each additional GPS trace processed, the degree to which it affects the generated map is variable. If it makes use of roads which had not been visited before, the map is not likely to be any different than if it were not included since the contributions to cells in the histogram will not rise above the threshold. On the other hand, if it makes use of roads which have been heavily visited, the map will also be minimally different because they will have little impact on the position of the centreline. Between these two extremes, the trace will have a more significant effect.

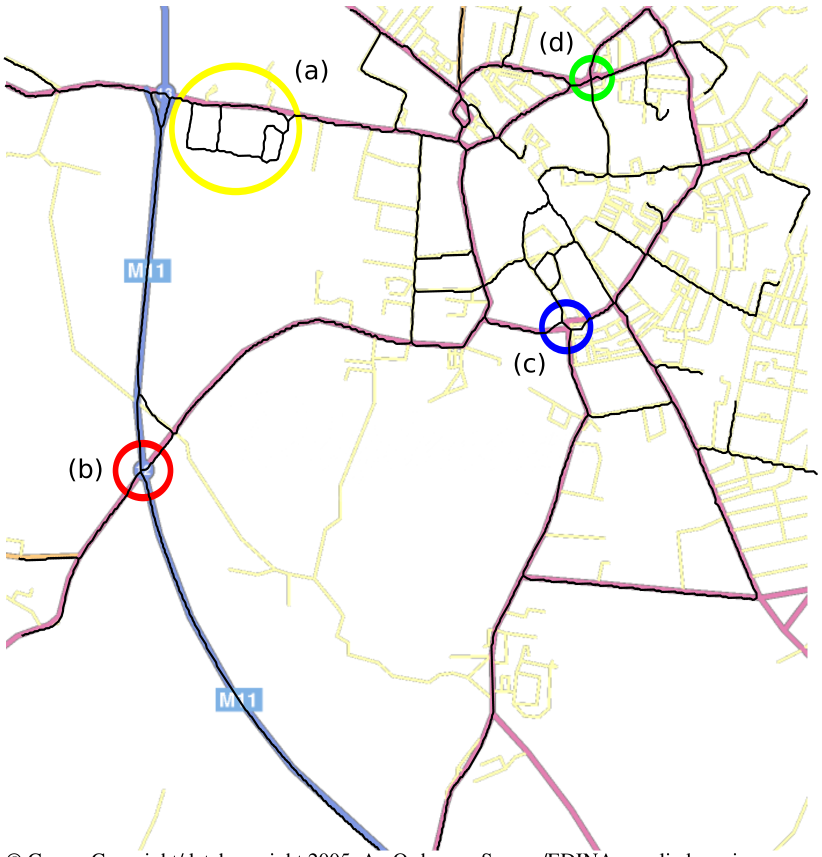


Figure 4.9: An Ordnance Survey map of the area with the generated road map overlaid in black. (a) The yellow circle highlights new roads; (b) the red circle highlights a bridge misinterpreted as a junction; (c) the blue circle highlights a junction which is misaligned; (d) the green circle highlights two junctions which have merged into one.

4.1.4.2 Practical analysis

The images in Figures 4.1 were generated from GPS traces constituting nearly one million position readings collected by the vehicle described in Chapter 3 driven in Cambridge. Road maps have also been generated from other sources of GPS data.

To evaluate the efficacy of the application, we have compared its output with a map of the same region created by the Ordnance Survey (OS): see Figure 4.9. Our non-optimised proof-of-concept implementation of the algorithm takes less than one minute to run to completion on a standard desktop workstation, for a 70 km^2 region comprising around a sixth of a million cells, with over five million GPS readings. In the histogram, 90% of the cells were empty, with 11% of the remainder falling below the empirically-determined threshold.

On the roads which have a sufficient density of GPS readings, the generated road segments align well with the OS road segments. However, from the roads which our GPS traces cover, a number of differences are evident:

- Some road segments exist in the generated map but not in the OS map. These correspond to newly-constructed roads that are not yet reflected in the OS data, confirmed by visiting the area, suggesting that the algorithm presented can be used to draw attention to the creation of new roads. An example of this is highlighted in yellow on Figure 4.9.
- The generated segments are jagged in shape whereas the OS segments are much smoother. This is a result of the segments being extracted from a Voronoi graph of stepped road boundaries. A topic for further work is to explore whether smoother segments could be produced.
- Road bridges are interpreted as crossroads in the generated map. An example of this is highlighted in red on Figure 4.9, where, although real junctions exist in the vicinity, the cross-roads which the generated map shows does not exist. This is a result of discarding altitude data from the GPS data when initially forming the two-dimensional histogram. There are two potential solutions to this problem. Firstly, using a three-dimensional histogram, extracting the three-dimensional surface (using an algorithm such as Marching Cubes [161]), and producing a three-dimensional Voronoi graph, we would find that the two roads no longer intersect. Alternatively, we could analyse each generated junction and determine the turns which are permissible for vehicles to make. Bridges would then be interpreted as crossroads in which no turns are permissible.
- Some junctions are skewed in the generated map. An example of this is highlighted in blue on Figure 4.9. This effect results from the errors inherent in the initial GPS fixes causing the histogram to fail to accurately represent the true road layout. This problem is hard to fix but may be solved by vehicles using accelerometers and sensor fusion techniques to improve the accuracy of location fixes.
- Some pairs of nearby junctions have merged together into one. An example of this is highlighted in green on Figure 4.9. This results from an excessively gross discretisation resulting from too large a cell size, or too heavy a blur. Hence this problem can

potentially be fixed by adjusting these parameters, at the cost of more expensive computation or of an increased risk of the presence of gaps in the histogram.

Small-scale changes such as lane closures on multi-lane roads are not reflected in the map. Even with a smaller cell size, individual lanes would not be distinguishable unless the location data was provided by a more accurate system than GPS. If such a system were used, individual lanes may be resolvable, but the result of vehicles changing lanes at arbitrary positions mean that lanes would be heavily cross-linked in the directed graph.

In summary, generating road maps in this way does not produce a perfect result, but neither do traditional map-making techniques.

4.2 Maintaining a model of the road network

The algorithm presented above can be used to process the location traces obtained from a single vehicle, and it will yield a map of only those roads that this vehicle has travelled on. In order to produce a complete map of the road network, we would need to use data from a large number of vehicles over a period of time. Thus, not only is it necessary to be able to collect GPS traces from many vehicles, we must also be able to run the algorithm repeatedly, updating the model of the road network with each new piece of information received. We address the latter problem of re-generating the map first and return to the former problem of collecting multiple traces in Section 4.3.

4.2.1 Map regeneration

It is desirable for the map generation application to have the ability to reflect (*i*) the creation of new roads, (*ii*) the closure of old roads, and (*iii*) the change in geometry of existing roads in the digital map. Using the algorithm described above, it is possible to establish that a new road has been opened: when GPS traces of vehicles making use of the road are acquired, the digital map will show the presence of the new road when regenerated. However, the same cannot be said about the other two aims: once we have some GPS traces travelling down a particular road, the digital maps we produce will contain that road in perpetuity.

In order to reflect changes that happen to existing roads over time in the digital map, we must place lower trust in older data than more recent data and thus rely on vehicles continuing to travel down roads regularly in order to maintain our level of trust in their existence. This can be achieved by incrementing the values in the histogram by smaller increments when processing an older GPS trace than when processing a more recent one. An entire histogram representing previous state can be aged by simply multiplying each cell by a scaling fraction. Thus, when a road is closed, we will cease to receive new GPS traces showing the road in use so eventually the value in the cells of the histogram will fall below our binarisation threshold, causing the road to disappear from the map. However, this implies that there will be a certain latency between the road closing and this fact being reflected in the map; this delay can be reduced if we can obtain more GPS traces from more vehicles.

4.2.2 Retaining associated metadata

For the map to be suitable for use in navigation, metadata needs to be associated with each edge in the directed graph. However, this may be a relatively expensive task to perform as it may involve some manual effort. Whilst metadata such as the speed limit could potentially be inferred from the GPS data, other metadata such as the road name could only be determined by visiting each road, or perhaps, in the future, from in-vehicle cameras or active road signs. Since we strive for up-to-date maps to be generated, the algorithm must be executed repeatedly as new GPS traces come to light, but we must be able to avoid the cost of having to reassociate the metadata from scratch every time we generate a fresh version of the map.

Ideally, we would like to be able to transfer the metadata associated with roads in an old map to the corresponding roads in a new map. However, with the benefit of the knowledge obtained from additional GPS traces, roads that existed in the old version may change their shape, and junctions may shift position. Furthermore, roads—and thus junctions—may appear or disappear. This makes it difficult to determine which roads in the old version correspond with which roads in the present version [102, §3.2.4.1].

One technique to estimate which road in the old version corresponds to which road in the present version respectively, and which roads exist in one version but not the other, is to employ a weighted bipartite graph[2]. The two sets of vertices of the bipartite graph represent the roads in the old version of the map and the roads in the present version. The edges in the bipartite graph are associated with a weight relating to the similarity between a pair of roads from those sets. Hence, a low weight edge between two vertices in the bipartite graph means that a road in the old version of the map very closely corresponds to a road in the present version. The weighting relates to the distance that the ends of the road network edges have moved by, and to the similarity in the shape of the edges. Figure 4.10c shows such a bipartite graph constructed from two versions of a road map shown in Figure 4.10a and 4.10b.

A minimum weight maximal matching[3] on this bipartite graph will therefore indicate which edges in old and present versions best correspond, and will contain high weight edges between the remaining vertices. These high weight edges can be ignored: they correspond to pairs of roads which exist in one version of the map but not the other. For the remaining low weight edges, metadata can be transferred between the roads corresponding to the vertices. In Figure 4.10d, the red edge represents a high weight edge which is discarded. From this matching, it is evident that road B has closed and roads 1, 3 and 7 are new.

[2]A bipartite graph is a special type of undirected graph in which vertices are partitioned into two disjoint sets, and which has no edges between two vertices in the same set. A weighted bipartite graph has weights associated with its edges.

[3]A matching, with respect to a graph, is a subset of the graph's edges with no vertices in common. A maximal matching is a matching which employs as many edges as possible. A minimum weight maximal matching is that which minimises the sum of the weights of its edges.

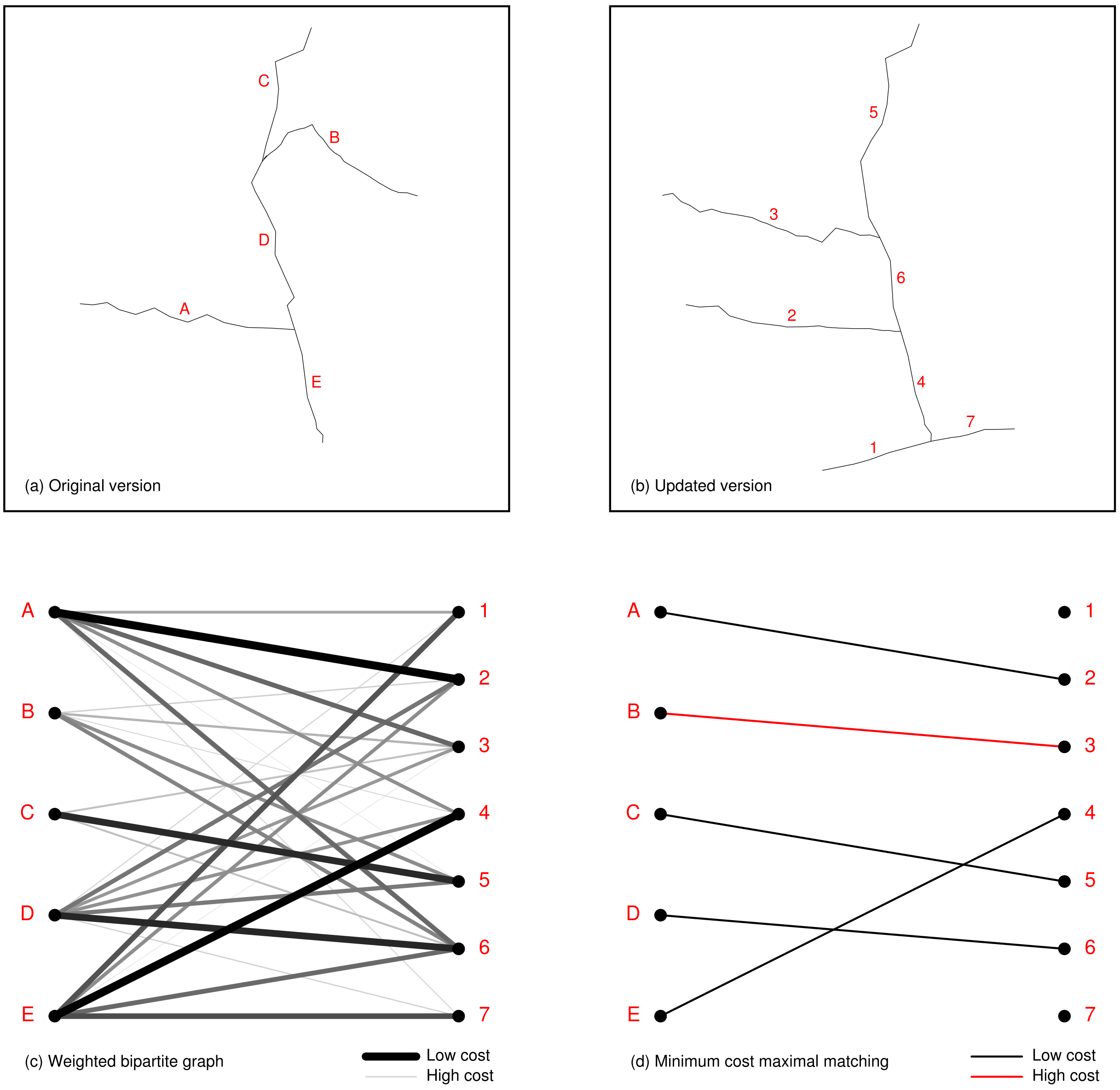


Figure 4.10: (a, b) Two versions of the road map; (c) the bipartite graph; and (d) the minimum cost maximal matching.

4.2.3 Cost of execution

Traffic data quoted in Cambridgeshire County Council's 2006 Traffic Monitoring Report [40] regarding a typical city-centre road indicates that an average of around 1000 vehicles use the road per hour. If we need 73 samples at a particular point along the length of a road to achieve an acceptable level of accuracy, then we should regenerate the map 14 times per hour—once every four minutes—with fresh data. Since the time complexity of the algorithm is quasilinear with area, our proof-of-concept implementation could process an area around 325 km^2 every four minutes. Thus, to be able to process an area the size of the entire United Kingdom, we would need around 750 processing nodes. However, we believe that an optimised implementation could execute at least an order of magnitude more quickly. Furthermore, it is only a minority of roads that see this kind of traffic flow and so require such frequent map regeneration.

Because of the cost of regenerating the map, we could instead choose to only regenerate it when we have sufficient new data to significantly affect the output. We can reduce the cost of executing the algorithm yet further by only regenerating the parts of the map which have received new data rather than the entire region.

4.3 Involving multiple vehicles

In-vehicle computational resources can be used to enable direct participation in the provision of accurate map data. The more vehicles from which GPS traces are obtained, the better the quality of the resulting map. We envisage vehicles on the road network forming a wide-scale mobile sensing and computing platform in order to achieve this.

4.3.1 Scalability

In order to be able to process GPS traces from a vast number of vehicles, covering a large geographical area, this algorithm needs to scale gracefully. Fortunately, the algorithm is highly parallelisable, by dividing up space into tessellating regions or *tiles*.

There are various stages of the algorithm at which the partitioning of data in this way could take place. For example, it could be the collection of GPS traces that is partitioned. For each region, a processor would be responsible for collecting the traces concerning journeys within that region. A central processor would then collate all of this data and compute the map.

Alternatively, each region's processor could get as far as producing a thresholded histogram, and then a central processor would process the combined histogram into a single map.

A third option is to produce a directed graph of the road network within each tile in isolation. Once each tile's graph has been produced, we then need to stitch the results back together into one complete graph. However, unfortunately, we cannot expect that roads spanning the edges of the tiles will necessarily align and thus be contiguous when juxtaposed. This is because we cannot be certain about the results produced near the

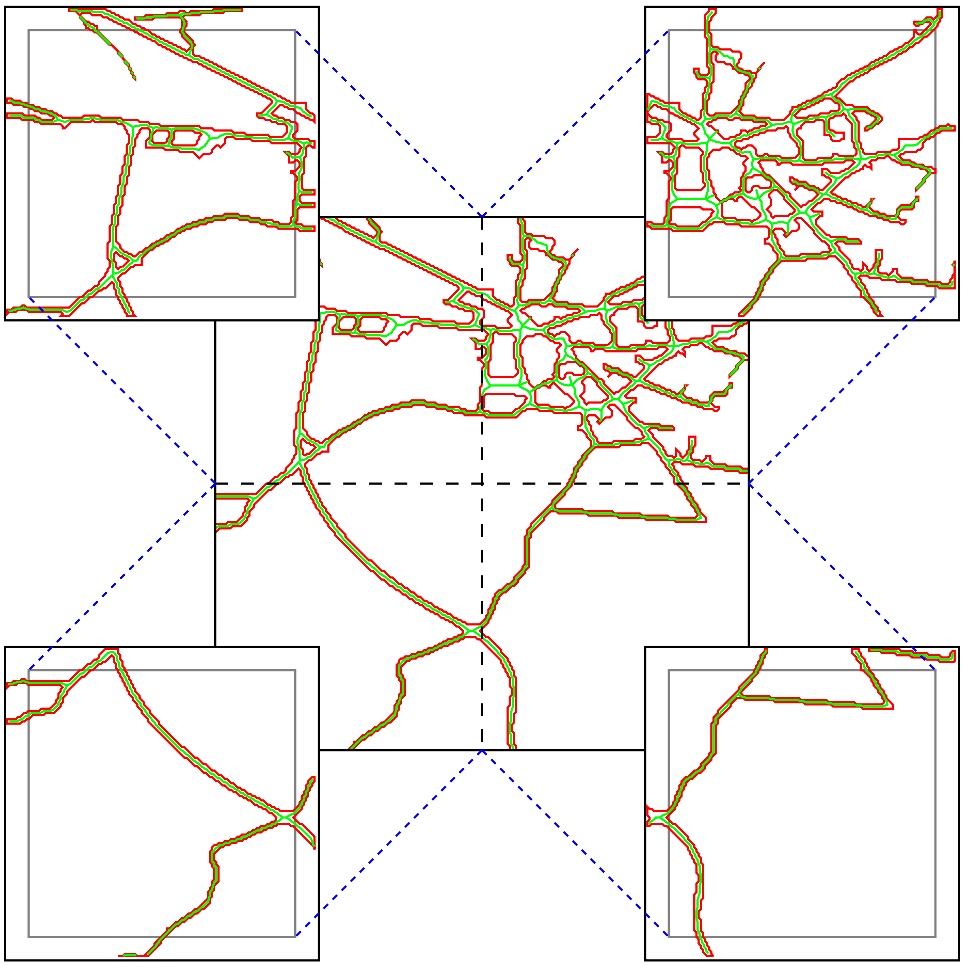


Figure 4.11: Merging together the parts of the directed graph lying within the central region of each tile. (The cell size is deliberately large to exaggerate the relative size of the overlap regions.)

border of a tile when it is processed in isolation. To solve this problem, we instead process tiles which overlap the adjacent tiles by the number of cells corresponding to the region of uncertainty. Then, when the resulting directed graphs from each region are to be stitched together, we simply clip the roads to the tile's central region. It will then be the case that a road traversing two adjacent tiles will be contiguous so we can join graph edges which meet at the seam between their respective tiles into a single edge. The ratio of the area of the overlap region to the area of the entire map corresponds to the overhead that parallelisation introduces.

We have tested this technique successfully by partitioning the data used in Figure 4.1 into four quadrants, mimicking separate processing nodes. Processing each quadrant individually and stitching the tiles together resulted in the same output as processing all the data at once. Figure 4.11 illustrates the technique.

4.3.2 System architecture

By virtue of our desire for a wide range of vehicles' GPS traces so that an accurate and complete road map can be computed, we need a means of collecting and processing the data from vehicles which are inherently geographically distributed.

There is a broad spectrum of architectures which could support an application such as map generation. We can view a network of vehicles along with any infrastructure support as a set of nodes in a loosely-coupled distributed system (see Section 2.2.2). At one end of the spectrum is the fully centralised approach, where vehicles upload their raw GPS readings to a central server which generates new map data. Despite bringing benefits, particularly in the timeliness of data delivery, this suffers from a single point of failure and is likely to be impractical because of the communication bandwidth needed to transmit all the GPS readings to the server.

To spread the communications bandwidth, we can employ regional processing servers, each responsible for processing data concerning its own geographical region. This approach may be preferred by commercial operators seeking to gather and process data cheaply. Further, to avoid requiring a costly backhaul network between regional servers, data gathered from the boundaries of the regions may be distributed by the vehicles themselves. Vehicles used in this way are referred to as 'data mules', which are a high latency but high bandwidth form of communication [39].

It may be possible to execute some of the processing steps on the vehicles themselves, an approach which is becoming increasingly common. For example, in the Vehicle Data Stream Mining (VEDAS) system [134], vehicles perform on-board processing of data from their sensors, uploading the results to a remote central server over a low-bandwidth wireless network.

Another alternative architecture is to provide public data caches which are connected to the Internet and store data but do not contain any processing facilities. In this scenario, the onus is on the vehicles themselves to acquire as many GPS traces as they can from the nearest public data cache (perhaps by IEEE 802.11 communications) and execute the map generation application locally. This approach may be preferable to consumers concerned about location privacy, where a more decentralised scheme can limit the transmission of personally identifiable data.

At the far end of the architecture spectrum is the fully peer-to-peer scenario in which vehicular ad-hoc networks (VANETs) are formed to share GPS and map data. Although this solution will not have any on-going service costs to customers, it is unlikely that sufficient data could be gathered in this way to produce up-to-date and reliable maps.

Selecting the appropriate architecture is difficult. Moreover, it is conceivable that the processing nodes available to execute the application will not be known in advance. This requires the decision about which computational resources to use to be delayed until just before run-time. Hence, a form of automatic task assignment is desirable (see Section 2.2.3). A system which allows programmers to design applications in an architecture-neutral fashion and which automatically selects the most appropriate computational resources will be the subject of Chapter 5.

4.4 Further work

Further research relating to the map generation algorithm could involve the following tasks:

- The potential of making the histogram's cell sizes adaptive could be investigated, so that areas with large numbers of GPS readings can be inspected more accurately.
- An additional stage in the algorithm could be implemented to analyse the direction of approach and departure of vehicles from junctions to determine their nature more precisely.
- The generated road maps tend to be jagged in shape as a result of the use of square cells in the histogram. A different map generation technique could be employed to generate smooth road segments. Alternatively, a line simplification algorithm could be applied to the segments, such as the one proposed by Douglas and Peucker [60].
- It may be possible to make the thresholding of the histogram adaptive rather than global. This would mean that busier roads would have a higher threshold applied to them than other roads so that the accumulation of noise from location errors does not make the road appear wider than it actually is.
- Similarly, the ageing of old data could be made adaptive to detect road closures more efficiently. For a road which is usually busy, an absence of vehicles is a strong suggestion that the road has been closed. For an infrequently visited road, an absence of vehicles may occur naturally even if the road remains open.

It will also be important for future research to consider the social and security implications of applications of this kind, such as the means by which vehicle owners' privacy concerns can be addressed, and how the system can be protected against the actions of malicious users.

4.5 Summary

We have examined how up-to-date digital road maps can be created from sensor data contributed by participating vehicles. An algorithm has been proposed for inferring a directed graph of the road network from sets of vehicle location traces. The algorithm involves a number of stages of processing involving techniques from image processing and graph theory. We will return to describe an implementation of the application in Chapter 7.

Generating maps in this way has some advantages over traditional map-making techniques. For example, the opening of new roads or the closure of existing roads can be reflected in the map in real-time. However, this requires metadata about a generated map to be maintained, which is problematic. A technique for managing this has been suggested that makes use of bipartite graphs.

In order to exploit data collected from large numbers of vehicles, the algorithm needs to operate in a scalable fashion. This is achieved by partitioning the input data into geographic regions that can be processed independently and then combined into a single map.

There is a wide range of potential system architectures that could support this kind of application. A particular architecture impacts on various aspects of the application's execution, especially its performance characteristics. If the computational resources that will be available are not known in advance of the application's execution, the application must be designed without architectural assumptions being made, and a system that performs automatic task assignment is required. The design of programs in this way is the subject of Chapter 5.

CHAPTER 5

Automatic task assignment

We have introduced a wide range of applications, many of which are characterised by the collection, processing and dissemination of sensor data in networks of vehicles. The example of automatic map generation was given in Chapter 4. We noted that this kind of application tends to produce the best results when as large a number of vehicles as possible is involved. We described a spectrum of system architectures which could support such applications, each with different trade-offs. But there is no architecture which is guaranteed to be best in all circumstances.

In this chapter, we broaden the scope of our discussion to general distributed systems, showing how a suitable architecture can be automatically selected. This is achieved through describing the application as a task graph on which transformations can be performed, and using automatic task assignment to map tasks to processors. For example, in the context of the automatic map generation application, the transformations may change a centralised version of the application into one in which the image processing steps are performed in parallel on the vehicles which generate the GPS data.

The concepts introduced in this chapter will form the basis for the language and compiler described in Chapter 6. In particular, we will show the foundations for a variety of optimisations for the compiler.

5.1 The problem of early physical binding

In a distributed computing environment, applications can be distributed across several computers, with separate computers responsible for executing different components of the application. It is often the case that there will be several potential computers that could suitably execute a given component. When designing applications, it is common to identify which of these computers is to be responsible for executing each component. We refer to this approach to systems designing as *early physical binding* as the decision about which node should execute each part of the application is made at design-time.

In many distributed computing environments, the members of the network and its layout are not known at design-time. Perhaps the network contains mobile nodes whose presence and communication characteristics vary over time; vehicular networks are an obvious example. Therefore, the early binding of application components to processors leads to poor resource utilisation because a best-guess approach must be taken. This will typically be less optimal than a post-hoc binding when the characteristics of the network are known. Perhaps slow processors or expensive communication links might be employed, or the application might be configured such that the battery life of low-power mobile nodes is not maximised.

The early physical binding approach also precludes the possibility of writing generalised applications which could execute in a variety of network configurations.

Asssuming that knowledge of the characteristics of the network can be obtained (perhaps using a suitable device discovery scheme) then *late physical binding* is preferable. In this approach, the processor on which an application component is to be executed is selected just before the application is run. Delaying this decision means that the placement of application components can be optimised with respect to the actual network characteristics.

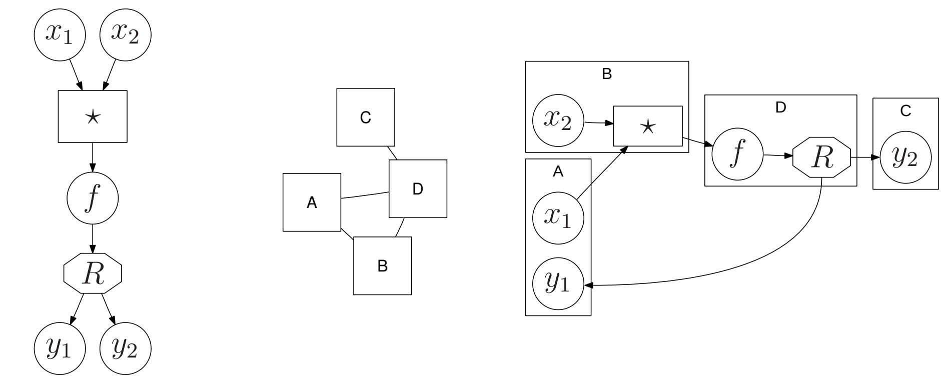
However, because it must happen for every deployment of the application, when the programmer is not necessarily present or able to assist, and must happen just before its execution is commenced, late physical binding cannot in general be performed manually; it must be done automatically. This is the problem of automatic task assignment, introduced in Section 2.2.3.

Moreover, a system which automatically determines the placement of application components also has the advantage of potentially being better at optimising placement than a manual approach, especially when the network is large, since complex trade-offs can be efficiently evaluated.

5.2 Automatic task assignment

Performing automatic assignment of program components to processors demands that the program is specified independently of the processors on which it may be executed. This means that programmers need only to focus on the algorithmic details of the program rather than on the networking aspects. Programmers use a language in which this can be done, which is fed to a compiler which computes an assignment and generates binaries which can be executed on each processor. We assume that the compiler has a global view of the network and can perform a global optimisation to find the best task assignment. (In some systems, this assumption may be somewhat naïve; we will discuss the specific case of large-scale vehicular networks in Section 5.4.1.)

The compiler must be provided with



(a) Task graph (b) Resource graph (c) Tasks assigned to resources

Figure 5.1: Example graphs.

- a description of the program (which can be thought of as the 'source code'),
- a description of the available processors and
- a cost function which provides the optimisation policy.

We will now give formal descriptions to these inputs. A concrete programming language in which these can be specified will be described in Chapter 6.

5.2.1 Task graph

A program is described by a *task graph* whose nodes are tasks which each consist of a set of instructions to be executed sequentially.[1] Formally, the task graph is a directed acyclic graph $G^{\mathrm{t}} = (E^{\mathrm{t}}, V^{\mathrm{t}})$ where the set of vertices, V^{t}, are the tasks and the edges, E^{t}, indicate the direction of data flow between tasks. An edge (v_1, v_2) indicates that task v_2 receives the output of task v_1, and that v_2 cannot commence execution until the execution of v_1 is complete. An example is drawn in Figure 5.1a.

Tasks are implicitly assumed to execute concurrently. The flow of data between tasks is data-driven rather than demand-driven. A data-driven approach involves a task notifying its successor task(s) to consume its output data; a demand-driven implementation would involve a task requesting input from its predecessor task(s). The demand-driven approach is a lazy approach, where the graph's inputs are only required to produce data when there is the demand to produce an output. Instead, we adopt the eager, data-driven approach so that outputs are produced as rapidly as the rate of data input allows.

The edges of a task graph represent data *streams*; each task is a stream processing element. A stream can be thought of as a temporally-ordered sequence of values. The execution of a task to process values from its input streams cannot commence until the processing

[1]According to Stephens' classification of stream processing systems [226, §3] (see Section D.3), task graphs are ANU systems: tasks execute asynchronously and are non-deterministic, and the channels are uni-directional.

of the previous ones has completed. Thus, in order to take advantage of *pipeline parallelism*, where at any moment in time multiple values from the stream are being processed concurrently, tasks must be broken down into shorter stages.

The ability for a task to only process one set of inputs at once means that it must employ queues at each input. As values arrive at a task for processing, they are pushed into a queue; when a task is ready to commence a new wave of execution it pops the front value from each queue. For now, we will treat these queues as being of infinite length to avoid the possibility of an input queue becoming filled.

5.2.2 Resource graph

The network of processors in which the application is to be executed is a graph $G^{\mathrm{n}} = (E^{\mathrm{n}}, V^{\mathrm{n}})$, where the vertices V^{n} model processors and the edges E^{n} model communication links between processors. We refer to this as a *resource graph*; an example is shown in Figure 5.1b. The processing nodes, which have local memory, are *not* assumed to be homogeneous in their processing power or communications capabilities.

The resource graph is usually undirected, implicitly indicating that the communication links are bi-directional, but can be directed if communication is not symmetric.

5.2.3 Assignment function

The output of task assignment is an assignment function $A : V^{\mathrm{t}} \to V^{\mathrm{n}}$ which maps tasks from the task graph to processing nodes from the resource graph. A valid assignment function is such that for each edge $(t_1, t_2) \in E^{\mathrm{t}}$ there exists a path $(n_1, n_2, \ldots, n_k)$ in G^{n} where $n_1 = A(t_1)$, $n_k = A(t_2)$ and $\forall i \in \{1, 2, \ldots, k-1\}.\ (n_i, n_{i+1}) \in E^{\mathrm{n}}$. In other words, the processor on which a task's successor is mapped must be reachable from the processor on which the task itself is mapped. An example of an assignment function is shown pictorially in Figure 5.1c, where the tasks x_1 and y_1 are mapped to processor A; the tasks x_2 and $\star$ are mapped to processor B; the task y_2 is mapped to processor C; and tasks f and R are mapped to processor D.

5.2.4 Cost function

The decision about which nodes to use affects the efficacy of the assignment. The efficacy of the assignment can be described quantitatively by a *cost function* specific to each application. A cost function $C : G^{\mathrm{t}} \times G^{\mathrm{n}} \times (V^{\mathrm{t}} \to V^{\mathrm{n}}) \to \mathbb{R}^+$ is a function of a task graph, a resource graph and an assignment function yielding a positive real number indicating the cost of the assignment. Higher values indicate less desirable assignments.

Applications will use a cost function which embodies the trade-offs they desire between relevant metrics. For example, one application may express in its cost function the policy that any execution time of less than two minutes is acceptable, and that minimising the use of network bandwidth is the next most important concern. Another application may seek to minimise total execution time at the expense of all other metrics. In some applications, the desired cost function may vary over time. For example, an application in a sensor network may desire high fidelity or high data-rate sensor readings for some periods of time (perhaps depending on external factors). In this case, a variety of different assignments may be computed.

Several metrics for evaluating an assignment function are relevant to many applications:

Total execution time. The duration of time elapsed from an item of data being produced to the final result derived from that data being delivered to all outputs. This metric takes into account the CPU time and the communication time whilst also respecting the queueing of incoming data at processors before service.

Network usage. The use of different networks may incur different (monetary) costs. For example, communicating over a 3G network may require a payment to the provider whereas the use of a privately-owned local WiFi network may be free of charge.

Quality of result. The number of items of input data which contribute to the computation of an output will affect the quality of the output. Also, any approximations which take place in the application will influence this.

Privacy. The level to which the privacy of the originators of the input data is respected is important in applications where personally-identifiable data is processed. The value of this metric could relate to an observer's view of the number of individuals who could have produced a particular item of data, known as the anonymity set.

Energy consumption. Another useful metric is the energy consumption caused by the execution of particular tasks. By associating with each processor a value indicating its power consumption and ability to switch to a low-power state, the total energy consumption for a given assignment function can be calculated.

So that a cost function can compute the values of relevant metrics, the graphs G^{t} and G^{n} must be weighted. Nodes of the resource graph (processors) are weighted with values describing their computational characteristics, such as processor speed. Edges of the resource graph (communication links) are weighted with values characterising the links, such as maximum throughput or latency.[2] Nodes of the task graph (tasks) are weighted with values describing their requirements, such as the number of instructions constituting them. Edges of the task graph (data flow) are weighted with values characterising the data, such as the size of the data or its level of confidentiality.

Task assignment is a well-understood area of research [144]. We will discuss task assignment algorithms, and their computational complexities, in Section 6.2.1. The implemented algorithm is not the focus of this dissertation; however, we will also describe it in that section. Unfortunately, task assignment is NP-complete, so it is often infeasible to find an optimal assignment and we must settle for a near-optimal solution.

[2]We assume that these characteristics are not affected by external factors. For example, we ignore the possibility that resources are shared with other applications; this could affect characteristics such as a processor's cache hit rate, a disk's seek time and the time to transmit a message on a given communication link.

5.3 Application design process

The typical process of designing an application using automatic task assignment involves a programmer defining a task graph and the cost function which describes the desired nature of the assignment.

For small applications, or applications in which the network is predictable, the resource graph could be hand-crafted by the programmer. But even in these cases, automatic task assignment is beneficial. However, we are left no choice but to use automatic task assignment when it is impossible to perform it manually, such as when the nature of the network is not known until run-time. For applications in large networks, a process of device discovery must be undertaken in order to ascertain the constituent components of the resource graph.

Once the resource graph is known, it can be combined with a task graph and cost function by a compiler which performs the assignment. The compiler produces executable code for each processor to which at least one task is assigned. As well as defining the tasks to execute, this must also include communication code to receive each task's input and deliver each task's output.

5.4 Applicability

Although motivated by large-scale vehicular sensor networks, this work is more generally applicable to several areas of distributed computing. Other forms of wireless sensor network are ideal candidates because of their characteristic use of nodes that are limited in power and processing capabilities, and have unknown or changing network topology [3], for which automatic task assignment is compelling.

Ubiquitous computing envisions an era when computers "weave themselves into the fabric of everyday life until they are indistinguishable from it" [243, p94]. Such a system requires a network of sensors which monitor the environment and its contents to gather information about the state of the world. This allows ubiquitous computing applications to interact seamlessly with humans in the environment. The use of a vast number of sensors, and the need to process this data into higher-level models of the environment, mean that automatic task assignment is also desirable in this area.

This work is also relevant to the areas of web services [4] and grid computing [78], since both of these technologies utilise data and processing capabilities in many different physical locations and across organisational boundaries. Grid computing enables programmers to implement applications which are distributed over multiple computers and which access repositories of data which are sufficiently large that moving programs onto processors near the data source is much easier than moving the data itself. In terms of data processing, there are some similarities between grid computing and sensor networks. The question of where data integration and processing is done is paramount. A large data store in the grid is infeasible to transfer across the network for processing or summarisation; instead, the processing must be executed close to the store. The OGSA-DAI framework [9] achieves this using a scripting language whose programs are sent over the grid and executed close to the data. Similarly, sensors exist at particular physical locations so processing on their outputs is best done close to them to avoid excessive network traffic.

In the vision of the Semantic Web, Berners-Lee et al. describe an application in which three members of a family wish to book an appointment together to see a suitable doctor [26, p28]. To arrange a suitable time for the appointment, the four parties' diary information needs to be compared. Determining where this comparison should be carried out is not obvious. For example, is it best to transfer the doctor's diary to the patient's computer, or the patient's diary to the doctor's computer? Here a measure of 'confidentiality' might be suitable for use as a metric for optimal placement of computation.

5.4.1 Vehicular networks

If automatic task assignment techniques are to be used in applications for vehicular networks, we must avoid the need to recompute the assignment of tasks from scratch every time the network conditions change because this is too frequent an occurrence. One way of working around this problem could be to produce approximate task assignments which make general statements about a task such as "execute it on every node", "execute it once per unit area" or "execute it centrally". This level of detail may suffice for many applications.

In order to produce assignments of this kind, we can use a variant of the resource graph in task assignment. Rather than the resource graph containing individual processing nodes, it contains *regions*. Each region is a spatial area which may contain many vehicles and is treated by the task assignment algorithm as an indivisible and opaque processing unit.

The intent of the use of regions is as a layer of abstraction that hides the mobile nature of vehicles. By defining a region such that its rate of departure of vehicles is roughly equivalent to its rate of arrival of vehicles, the externally visible characteristics of the region will be largely constant over time. Now, task assignment will assign tasks to regions and it is the responsibility of the members of each region to organise themselves so that the inputs are collected from the neighbouring regions, the tasks are executed and the results are sent to the destination regions. We refer to applications for which such regions can be defined as *quasi-static* applications.

This is a similar approach to that of the principle of a 'virtual ad-hoc server' used as part of the inter-vehicular communication protocol VITP [57]. In this protocol, vehicular sensor data queries are aimed towards a specified spatial region. The response is formed by a group of nodes whose membership may change over time. Similarly, the SpatialViews framework provides a programming language in which spatial regions can be expressed and which executes code on whichever devices are found to be located in that region [189] (see Section 2.3.2.1).

In order to guarantee that tasks are executed, such a protocol must ensure that information is not lost, especially handling the case where a region temporarily empties of vehicles. To prevent this, other regions must be responsible for monitoring adjacent regions. Moreover, within a region, since vehicles could leave the region without giving any advance warning, measures must be taken to ensure that other members of the region are aware of the work that needs to be done; this is likely to involve some redundancy as the work is replicated amongst a number of vehicles.

We leave the implementation of this protocol to further work (see Section 5.8) and now concentrate on the issues of task assignment for static applications whose topology is not predictable or known by the application developer.

5.5 Task graph optimisation

Automatic task assignment brings the ability to find a near-optimal mapping of application tasks to processors for a given task graph. However, some applications can be expressed in a variety of semantically-equivalent task graphs. Therefore we are led to consider whether we can determine an even better fashion in which to execute an application by automatically finding an alternative task graph which is somehow better-suited to the resource graph.

For example, in the automatic map generation application described in Chapter 4, we could imagine expressing the task graph in a centralised fashion, where the vehicles' GPS data is gathered at a central server before being processed. Alternatively, the data processing could be done in parallel so that the image processing steps are performed on each data partition independently before being combined together into a single map.

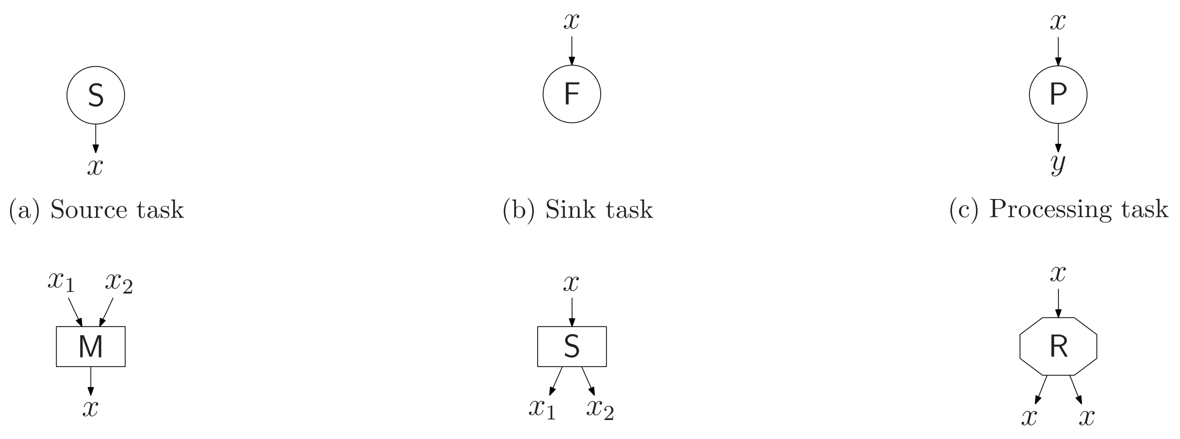
The attributes of the vehicles and of the central server described in the resource graph will dictate which of these alternative task graphs would produce maps at a faster rate. For example, with some resource graphs, this task graph could be naturally mapped to a scenario in which the vehicles each process the data that they generate. Or, if the resource graph indicates that there are fixed servers in each geographical region—characterised by having low-cost communication edges to some vehicles and high-cost edges, or none at all, to others—then these could be used to process subsets of the data, collected from local vehicles, in parallel.

As well as differing in terms of execution time, these alternative task graphs could differ in terms of their treatment of privacy. Since a vehicle's GPS data contains details about where its drivers have taken it, passing this data to untrusted third parties for processing will constitute a loss of location privacy. Hence, processing one's own data locally before communicating it to others may be preferable if the processing reduces the quality or amount of sensitive data contained in it.

In general, for a given application, there will be a spectrum of alternative, equivalent task graphs involving different stages of processing in different orders. We will consider the nature of this equivalence relation between alternative task graphs by examining the transformation space which explores the different versions of the algorithm.

5.5.1 Types of task

Firstly, in order to identify useful task-level program transformations, it is prudent to classify program tasks based on the numbers of inputs and outputs and their behaviour. We identify six kinds of task:



(a) Source task (b) Sink task (c) Processing task

(d) Merge task (e) Split task (f) Replication task

Figure 5.2: The six kinds of task.

Source tasks are where data is produced, drawn in a task graph as circles (Figure 5.2a). A source task has no inputs and one output. Although they only have one output edge, multiple values can be emitted in a sequential fashion. In other words, a source task produces a stream of values. For example, in a sensor network, a thermometer which outputs the temperature once per minute is modelled as a source task.

Sink tasks are where data is consumed, also drawn as circles (Figure 5.2b). A sink task has one input and no outputs. For example, in a sensor network, a sink task could be used to output the results of data processing to a physical display.

Processing tasks are functions which transform data of one type to another type, and are drawn as circles (Figure 5.2c). A processing task has one input and one output.

Merge tasks are functions which combine two items of data of a particular type into a single value of that same type, and are drawn as rectangles (Figure 5.2d). A merge task has two inputs, a single output and is commutative and associative.

Split tasks are functions which decompose a single item of data of a particular type into two values of that same type, and are drawn as rectangles (Figure 5.2e). These values must be constructed such that, when fed into a merge task, the original item of data is yielded. Thus, split tasks can be thought of as the inverse of merge tasks. A good split function will break an item of data into two parts of roughly equivalent size. Split tasks allow large items of data to be *partitioned* into smaller items so that computation can be performed in parallel. This permits a *divide-and-conquer* approach to data processing.

Replication tasks are functions which copy a value into a pair of identical values, and are drawn as octagons (Figure 5.2f). A replication task thus has a single input and two outputs.

The example task graph shown in Figure 5.1a consisted of two source tasks and two sink tasks connected by a merge task performing a $\star$ operation; a processing task performing an f function; and a replication task.

5.5.2 Denotational semantics

In order to formalise the definitions of task graphs and of the tasks, we provide a denotational semantics for task graphs. Not only will this aid the intuitive understanding of the processing performed by task graphs, we will also use the denotational semantics in proofs, particularly in justifying that the transformations which will be implemented in the compiler are correct.

The denotation of a task graph is a predicate in higher-order logic whose arguments are the inputs and outputs. In a complete graph, the inputs and outputs are the sources and sinks, respectively. As is the norm, the denotational semantics is compositional; that is, the denotation of a task graph is built out of the denotation of its components.

In this semantics, we will only model the mathematical relationship between the inputs and outputs in a single wave of execution. We do not consider the temporal or sequential aspects of the data streams flowing through the graph or other aspects of its execution.

The denotations of processing, merge, split and replication tasks are as follows:

$$\begin{aligned} \mathsf{Proc}(f)(x,y) &\triangleq y = f(x), && (5.1)\\ \mathsf{Merge}(\star)(x_1,x_2,x) &\triangleq x = x_1 \star x_2, && (5.2)\\ \mathsf{Split}(\circledast)(x,x_1,x_2) &\triangleq (x_1,x_2) = \circledast(x), && (5.3)\\ \mathsf{Rep}(x,x_1,x_2) &\triangleq x = x_1 \wedge x = x_2. && (5.4) \end{aligned}$$

For merge tasks, the $\star$ operation must be an associative and commutative binary operator. For convenience, we write expressions involving $\star$ using infix notation; $a \star b \triangleq \star(a,b)$. The associativity and commutativity of $\star$ are respectively defined as:

$$\forall a,b,c.\ (a \star b) \star c = a \star (b \star c); \qquad (5.5)$$

$$\forall a,b.\ a \star b = b \star a. \qquad (5.6)$$

The following properties of tasks follow directly from the commutativity of $\star$ and the equality of outputs from replication tasks:

$$\begin{aligned} \mathsf{Merge}(\star)(x_1,x_2,x) &\equiv \mathsf{Merge}(\star)(x_2,x_1,x), && (5.7)\\ \mathsf{Rep}(x,x_1,x_2) &\equiv \mathsf{Rep}(x,x_2,x_1). && (5.8) \end{aligned}$$

The denotation of a task graph is composed from the logical conjunction of these predicates, instantiated with the appropriate inputs and ouputs, with the existential quantifier used to hide 'internal' connections.

The task graph shown in Figure 5.1a has the following denotation. We refer to it by the predicate $\mathsf{Example}$, which takes the graph's two inputs and two outputs as parameters. The two internal connections are referred to by the existentially quantified variables t_1 and t_2.

$$\begin{aligned} \mathsf{Example}(x_1,x_2,y_1,y_2) &\triangleq \exists t_1,t_2.\ \mathsf{Merge}(\star)(x_1,x_2,t_1) \wedge \\ &\qquad\qquad \mathsf{Proc}(f)(t_1,t_2) \wedge \mathsf{Rep}(t_2,y_1,y_2) \\ &\equiv \exists t_1,t_2.\ t_1 = x_1 \star x_2 \wedge \\ &\qquad\qquad t_2 = f(t_1) \wedge y_1 = t_2 \wedge y_2 = t_2 \\ &\equiv y_1 = y_2 = f(x_1 \star x_2). \end{aligned}$$

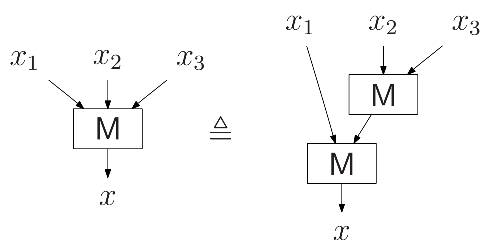


Figure 5.3: A ternary merge task defined in terms of a chain of binary merge tasks.

This simplified expression indicates the mathematical relationship between the inputs and outputs.

5.5.3 Generalising to n-ary tasks, $n > 2$

It is common to want to merge more than two items of data, although the definition of merge tasks above only permits two items to be combined. Merge tasks are particularly important in applications where a large number of input values need to be processed, such as in sensor networks or grid computing. Because of the wealth of input data, it is usually necessary to be able to aggregate data into a significantly smaller amount of information to make their processing and interpretation more manageable.

This notion of combining several items into a single value is common in parallel computing, and is often referred to as 'reduction'. For example, MapReduce is partly named after this principle [56]. Reduction is a fundamental part of the programming abstraction in Abstract Regions [244, §3.1] (see Section 2.3.2.1) and is a primitive operation provided by the Message-Passing Interface (MPI) [242, §3.4.2] (see Section 2.2.2.3).

More than two items of data can be combined into a single value by chaining several merge tasks together in any order. For convenience, we draw a chain of merge tasks combining n items of data as a single, n-ary task. A ternary merge task and an equivalent chain of binary merge tasks are shown in Figure 5.3.

Similarly, it is common to want to replicate a single value more than twice so that it can be distributed amongst a large number of participants, or to want to split a value into more than two parts. As with merge tasks, a chain of replication or split tasks can be constructed in order to generate more than two replicas or partitions of a value respectively, and are drawn as a single, n-ary task.

Formally, the n-ary merge, split and replication tasks, where $n > 2$, are defined inductively by the following rules, where Merge_2, Split_2 and Rep_2 are synonymous with Merge, Split and Rep, respectively.

$$\mathsf{Merge}_n(\star)(x_1,\ldots,x_n,x) \triangleq \exists t.\ \mathsf{Merge}_{n-1}(\star)(x_1,\ldots,x_{n-1},t) \wedge \mathsf{Merge}(\star)(t,x_n,x), \tag{5.9}$$

$$\mathsf{Split}_n(\circledast)(x,x_1,\ldots,x_n) \triangleq \exists t.\ \mathsf{Split}_{n-1}(\circledast)(t,x_1,\ldots,x_{n-1}) \wedge \mathsf{Split}(\circledast)(x,t,x_n), \tag{5.10}$$

$$\mathsf{Rep}_n(x,x_1,\ldots,x_n) \triangleq \exists t.\ \mathsf{Rep}_{n-1}(t,x_1,\ldots,x_{n-1}) \wedge \mathsf{Rep}(x,t,x_n). \tag{5.11}$$

However, this definition of n-ary split tasks is inadequate. A good binary split task will divide its input into two pieces of roughly equal size. Hence, as defined above, a ternary split would divide an input of size x into three pieces, of sizes $\frac{x}{2}$, $\frac{x}{4}$ and $\frac{x}{4}$. But a good ternary split task would preferably divide its input into pieces of sizes $\frac{x}{3}$, $\frac{x}{3}$ and $\frac{x}{3}$. From a theoretical standpoint, we will continue to treat an n-ary split task in terms of a chain of binary tasks, but we will allow programmers to specify n-ary split tasks which produce balanced outputs.

5.5.4 Fault tolerance

In a large-scale system, merge tasks must be employed in order to summarise or aggregate many input values. However, the various inputs may have different data rates. In many applications, if there is a single merge task collating, say, one hundred inputs, and at a point in the execution ninety-nine of them have arrived, it may sometimes be most prudent to execute the merge function on those ninety-nine and ignore the late-comer. In other words, we may choose to trade the fidelity or accuracy of outputs for the frequency at which they are produced. In the extreme case, if one of the inputs to a task fails altogether, then it may be preferable to produce *some* output rather than no output at all, waiting infinitely for data to arrive on an input where an upstream task has failed. This may be particularly important in sensor networks where failures of processors or communication links are frequent.

This problem is avoided by extending the definition of merge tasks to allow them to produce a result based on the available inputs after a specified timeout has expired, so that they are resilient to the failure of a subset of the inputs. If the actual time of arrival of the ith input, relative to the start of execution, is t_i^{a} and the latest permitted arrival time is t_i^{d}, then the semantics of the extended definition of n-ary merge tasks would be as follows:

$$\mathsf{Merge}_n(\star)(x_1, \ldots, x_n, x) \quad \triangleq \quad x = \begin{cases} e & \text{if } I = \varnothing, \\ x_0 & \text{if } I = \{x_0\}, \text{ or} \\ \bigstar_{i \in I} x_i & \text{otherwise} \end{cases}$$

where $I = \{i \mid 1 \leq i \leq n \wedge t_i^{\mathrm{a}} \leq t_i^{\mathrm{d}}\}$, e is a value satisfying $\forall x.\ e \star x = e$, and $\bigstar$ is the cumulative version of the binary $\star$ operator.

5.5.5 Examples: aggregation operators

Continuing the theme of processing large volumes of input data, we consider approaches to implementations of aggregation operators.

The database community has traditionally expressed aggregation of data in terms of processing and merge functions. For example, the designers of FAD describe aggregation in terms of an operator called *pump* which takes a unary operator which is mapped to each element of a set, and a tree of associative and commutative binary functions which aggregate the data [19, p103]. The authors give the example of computing the average salary of a set of employees, which involves the three functions

$$\begin{aligned} f(x) &= (1, x.\text{salary}), \\ g(x,y) &= (x.1 + y.1, x.2 + y.2), \\ q(x,y) &= \frac{y}{x}. \end{aligned}$$

Functions f and g are passed to the pump operator along with a set of employee objects. The value returned from the pump is passed to the function q to yield the desired result.

The TAG framework for wireless sensor networks, described in Section 2.3.1.1, adopts a similar approach to resolve queries for aggregate values of data from sensors in the network [165]. To implement an aggregate they employ three functions which they refer to as a *merging* function, an *initializer* function and an *evaluator* function [165, p134]. The initializer (such as f above) is used to specify how to transform a sensor value into a 'partial record state', a value which can be aggregated (which is a processing function); the merging function (such as g above) takes two partial record states and returns a single partial record state (which we refer to as a merge function); the evaluator (such as q above) takes a partial record state and returns the final value of the aggregate (which is a processing function). The authors state that all the basic SQL aggregates are expressible in this way.

Task graphs of several common aggregation functions which either produce a summary of a set of input values or return an exemplar value can be found in Appendix G. We give the example of finding the median value of a set of input values here.

5.5.5.1 Median

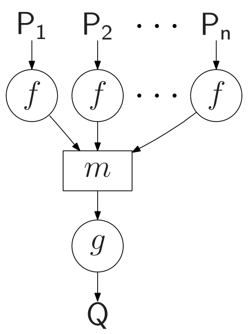


Figure 5.4: Finding the median input value.

The task graph in Figure 5.4 finds the median input value.[3] For n input values, $n + 1$ processing tasks and one n-ary merge task are employed. We use the terms P_1, P_2 and P_n to denote sub-graphs with a single output, and Q to denote a sub-graph with a single input.

[3]There exist algorithms to find the median in a more efficient manner, rather than collecting all the data centrally. We use a simplistic approach to median-finding in this example for clarity.

The processing function f creates a unit list from its input,

$$f(x) \quad = \quad [x].$$

The merge function is that used in merge-sort to produce a single sorted list from two sorted lists, defined as

$$\begin{array}{rcl} m(as, []) & = & as \\ m([], bs) & = & bs \\ m(a +\!\!+ as, b +\!\!+ bs) & = & \begin{cases} a +\!\!+ m(as, b +\!\!+ bs) & \text{if } a < b \\ b +\!\!+ m(a +\!\!+ as, bs) & \text{otherwise,} \end{cases} \end{array}$$

where $+\!\!+$ is the list-cons function such that $a_1 +\!\!+ [a_2, \ldots, a_n] = [a_1, a_2, \ldots, a_n]$. The processing function g selects the median value from the list and is defined as

$$g([x_0, x_1, \ldots, x_n]) \quad = \quad \begin{cases} x_{\frac{n}{2}} & \text{if } n \text{ is even} \\ \frac{1}{2}(x_{\frac{n-1}{2}} + x_{\frac{n+1}{2}}) & \text{otherwise.} \end{cases}$$

5.5.6 Datatypes

Each item of data passed between tasks can be thought to belong to a particular datatype. A datatype is defined in terms of a set of underlying values. For example, in a sensor network, a *temperature* datatype would have underlying set $\mathbb{R}$.

5.5.6.1 Mergeable datatypes

Some datatypes which hold values of type T also have a binary operation $\star : T \times T \rightarrow T$ which is associative and commutative, and have an identity element $i \in T$ such that

$$\forall a \in T.\ i \star a = a. \tag{5.12}$$

The identity element denotes the datatype's 'empty' value.

Theorem 5.1. *The identity element is unique.*

Proof. Assume there are two distinct identities, $i_1, i_2 \in T$, $i_1 \neq i_2$. Then,

$$\begin{array}{rcll} i_1 & = & i_1 \star i_2 & \text{because } i_1 \text{ is an identity and (5.12)} \\ & = & i_2 \star i_1 & \text{by commutativity of } \star \text{ and (5.6)} \\ & = & i_2 & \text{because } i_2 \text{ is an identity and (5.12),} \end{array}$$

which is contradictory. □

These datatypes are of particular interest because their operation $\star$ coincides with the definition of merge tasks above. Using $\star$, several items of data of the same type can be combined into a single item of that type.

Such a datatype is modelled mathematically as a *commutative monoid*[4] $(T, \star, i)$. Some examples of simple commutative monoids are:

[4]A monoid is an abstract algebraic construct which can be thought of as a semigroup [116] with an identity element. A commutative monoid's operation must be commutative and is sometimes referred to as an *abelian monoid* by analogy with an abelian group.

$$\begin{array}{rl} \text{addition} & (\mathbb{R}, +, 0), \\ \text{set union} & (\mathcal{P}(S), \cup, \varnothing), \text{ and} \\ \text{maximisation} & (\mathbb{R} \cup \{-\infty\}, \max, -\infty). \end{array}$$

We refer to these datatypes as being *mergeable*.

Associativity and commutativity reflect the idea of summarising data from a *multiset* of input values: the order does not matter, but duplicates are retained. Note that although ordered lists form a monoid under the append operation with the empty list as the identity element, this is not a commutative monoid since append is not commutative.

Since a split task for a particular datatype is an inverse of its merge task, it follows that mergeable datatypes necessarily support split operations. By analogy with the monoid $(\mathbb{N}, \times, 1)$, where merging is multiplication, splitting is factorisation into a pair of factors. Note that although splitting merely needs to be a right-inverse for merge, i.e. a function $\divideontimes$ satisfying

$$\forall x.\ \star(\divideontimes(x)) = x, \tag{5.13}$$

and therefore many split operations may exist for a given merge operation $\star$, we will nonetheless use the notation $\star^{-1}$.

5.5.6.2 Processing functions

In an application which processes data, it is not always enough to manipulate data within a single type, so functions $f : T_1 \to T_2$, where $T_1 \neq T_2$, are necessary in order to transform data into a new type. We refer to such functions as *processing* functions since they coincide with the definition of processing tasks above.

A processing function f is a *monoid homomorphism* if it transforms data from one monoid $(S, \star, i_1)$ into data from another monoid $(T, \otimes, i_2)$ whilst satisfying the property that for all $a, b \in S$,

$$f(a \star b) = f(a) \otimes f(b). \tag{5.14}$$

An example of a monoid homomorphism is a function $f(x) = e^x$ from monoid $(\mathbb{N}, +, 0)$ to monoid $(\mathbb{R}, \times, 1)$. It is trivial to check that $f(a+b) = f(a)f(b)$ and confirm that $f(0) = 1$.

In practice, the constraints of computation mean that real implementations of datatypes are not necessarily perfect monoids. For example, addition and multiplication are only approximately associative in floating point arithmetic, thus do not faithfully implement the monoid $(\mathbb{R}, +, 0)$. Similarly, certain thresholding operations, e.g. $f(a) = \lfloor a \rfloor$, do not satisfy property (5.14) to be homomorphisms; timeouts on merge tasks further complicate the issue. A formal treatment would involve adding a metric space structure to monoids and adding a continuity requirement for homomorphisms and then to argue that the approximate behaviour is 'close enough' for a given application.

Monoids, and commutative monoids in particular, have found similar application in other fields. The Monoid Homomorphism Language (MHL) is a database query language in which monoids model datatypes and monoid homomorphisms are the sole means of manipulating data [1, §6].

5.5.7 Task graph transformations

We now return to the topic of determining the equivalence relation between alternative task graphs.

Task graph designers are expected to identify which datatypes are appropriate to treat as monoids and which processing functions are appropriate to treat as monoid homomorphisms. An advantage of this is that static analysis can be used to transform the task graph whilst maintaining semantic integrity. We believe that designers will be able to easily identify these in everyday applications. In the worst case, when these are overlooked, this merely results in a smaller range of optimisations being available to the compiler.

We will present a number of task graph transformations that are expressed as bidirectional task graph re-write rules. These transformations will be implemented by the compiler that will be presented in Chapter 6.

We begin by considering two example applications to provide the context in which we will discuss the first transformation.

5.5.7.1 Arithmetic mean

In sensor networks, it is common to want to find the arithmetic mean of a large number of sensor readings. The centralised approach would gather and sum all the readings at the sink node and divide by the number of readings received. Partitioning the problem into smaller subsets of readings means that we can reach an answer using less energy or more quickly as several additions can be executed in parallel. However, the arithmetic means of arbitrary, distinct subsets of readings cannot be readily combined into the overall mean, because the number of readings contributing to each subset's mean is lost. A solution to this problem is to keep a running total of the number of readings in each partition. Adopting this approach, we can express the arithmetic mean of a set of numeric values by employing two processing functions—one a homomorphism, the other not.

A multiset of real numbers is represented by the monoid $(\mathcal{P}(\mathbb{R}), \uplus, \varnothing)$[5] where $\uplus$ is multiset union. We use an intermediate monoid $(\mathbb{R} \times \mathbb{N}, \oplus, (0,0))$, where

$$(a_1, n_1) \oplus (a_2, n_2) \triangleq (a_1 + a_2, n_1 + n_2),$$

to store the numerator and denominator in the calculation of the arithmetic mean. The value of the mean will not be represented by a monoid but will simply be a value drawn from $\mathbb{R}$.

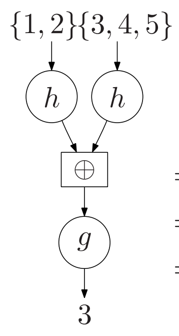
The function to convert a multiset of numbers into the intermediate form is

$$\begin{aligned} h(\varnothing) &= (0,0) && (5.15)\\ h(\{x\} \uplus xs) &= (x,1) \oplus (h(xs)). && (5.16)\end{aligned}$$

[5]Here, we use $\mathcal{P}$ to denote the multiset powerset function,

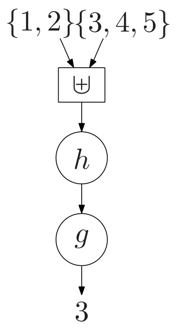
$$\mathcal{P}(S) \triangleq \{x \,|\, x \subseteq S\},$$

where $\subseteq$ denotes the sub-multiset relation.



$$\begin{aligned} & g(h(\{1,2\}) \oplus h(\{3,4,5\})) \\ = \; & g((3,2) \oplus (12,3)) \\ = \; & g(15,5) \\ = \; & 3 \end{aligned}$$

(a) Processing before merging



$$\begin{aligned} & g(h(\{1,2\} \uplus \{3,4,5\})) \\ = \; & g(h(\{1,2,3,4,5\})) \\ = \; & g(15,5) \\ = \; & 3 \end{aligned}$$

(b) Merging before processing

Figure 5.5: Example task graphs for computing the arithmetic mean of two sets of values in a distributed fashion.

Values from the numerator–denominator monoid returned from h can then be transformed into the desired result using a non-homomorphic function g:

$$g(x, n) = \frac{x}{n}.$$

The structure of this program is similar to how aggregate functions are defined in TAG (see Section 5.5.5 above). The function h is akin to an 'initialiser'; the combining of values via $\oplus$ is like a 'merging' function; and the function g is similar to an 'evaluator'.

An example task graph for this application is depicted in Figure 5.5a. The sum and count of two multisets of values are computed by h before being combined by the $\oplus$ merge task. Finally the mean is computed by g.

However, the task graph shown in Figure 5.5a can be equivalently expressed as shown in Figure 5.5b. The former is a conversion to numerator–denominator pairs (*processing*) for both of the sets, followed by the summing function $\oplus$ (*merge*), and finally g. The latter is a multiset-union operation (*merge*) on the two sets, followed by the conversion by h to the numerator–denominator pair (*processing*), and finally g.

The denotations of these two versions of the graph are:

$$\begin{aligned}
\mathsf{Mean}_1(x_1, x_2, y) &\triangleq \exists t_1, t_2, t_3.\ \mathsf{Proc}(h)(x_1, t_1) \wedge \mathsf{Proc}(h)(x_2, t_2) \wedge \\
&\qquad\qquad\quad \mathsf{Merge}(\oplus)(t_1, t_2, t_3) \wedge \\
&\qquad\qquad\quad \mathsf{Proc}(g)(t_3, y) \\
&\equiv \exists t_1, t_2, t_3.\ t_1 = h(x_1) \wedge t_2 = h(x_2) \wedge \\
&\qquad\qquad\quad t_3 = t_1 \oplus t_2 \wedge y = g(t_3) \\
&\equiv y = g(h(x_1) \oplus h(x_2));
\end{aligned}$$

$$\begin{aligned}
\mathsf{Mean}_2(x_1, x_2, y) &\triangleq \exists t_1, t_2.\ \mathsf{Merge}(\uplus)(x_1, x_2, t_1) \wedge \\
&\qquad\qquad \mathsf{Proc}(h)(t_1, t_2) \wedge \mathsf{Proc}(g)(t_2, y) \\
&\equiv \exists t_1, t_2.\ t_1 = x_1 \uplus x_2 \wedge t_2 = h(t_1) \wedge y = g(t_2) \\
&\equiv y = g(h(x_1 \uplus x_2)).
\end{aligned}$$

We can see that these two expressions are equivalent if h is a homomorphism from $(\mathcal{P}(\mathbb{R}), \uplus, \varnothing)$ to $(\mathbb{R} \times \mathbb{N}, \oplus, (0, 0))$, by property (5.14).

Proposition 5.2. *Function h is a homomorphism.*

Proof. We prove $h(a \uplus b) = h(a) \oplus h(b)$ by induction on the structure of a.

For the base case, we consider $a = \varnothing$:

$$\begin{aligned}
h(\varnothing \uplus b) &= h(b) \\
&= (0, 0) \oplus h(b) && \text{because } (0, 0) \text{ is an identity and (5.12)} \\
&= h(\varnothing) \oplus h(b) && \text{by (5.15).}
\end{aligned}$$

We now assume the inductive hypothesis $h(a \uplus b) = h(a) \oplus h(b)$ and show that $h((\{x\} \uplus a) \uplus b) = h(\{x\} \uplus a) \oplus h(b)$:

$$\begin{aligned}
h((\{x\} \uplus a) \uplus b) &= h(\{x\} \uplus (a \uplus b)) && \text{by assoc. of } \uplus \text{ and (5.5)} \\
&= (x, 1) \oplus h(a \uplus b) && \text{by (5.16)} \\
&= (x, 1) \oplus (h(a) \oplus h(b)) && \text{by inductive hypothesis} \\
&= ((x, 1) \oplus h(a)) \oplus h(b) && \text{by assoc. of } \oplus \text{ and (5.5)} \\
&= h(\{x\} \uplus a) \oplus h(b) && \text{by (5.16).}
\end{aligned}$$

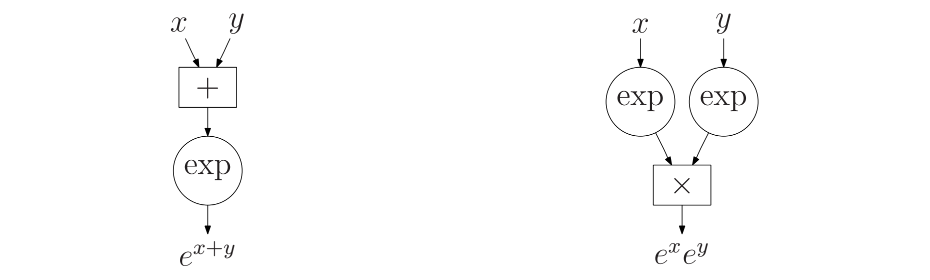
Thus h satisfies (5.14). □

Since h is a homomorphism, property (5.14) implies that $h(x_1) \oplus h(x_2) = h(x_1 \uplus x_2)$, so

$$\mathsf{Mean}_1(x_1, x_2, y) \equiv \mathsf{Mean}_2(x_1, x_2, y).$$

5.5.7.2 Exponentiation

Similarly, an algorithm to compute the exponential of the sum of two natural numbers could be equivalently expressed as the product of exponentials: $e^{x+y} = e^x e^y$. Exponentiation can be thought of as a function from monoid $(\mathbb{N}, +, 0)$ to monoid $(\mathbb{R}, \times, 1)$. The



(a) Addition before exponentiation (b) Exponentiation before multiplication

Figure 5.6: Exponentiation of two natural numbers.

formula e^{x+y} is an additive *merge* of the two natural numbers x and y followed by an exponentiation *processing* task (depicted in Figure 5.6a); a transformation of the formula gives $e^x e^y$ which is two exponentiation *processing* tasks followed by a multiplicative *merge* of the resulting numbers (depicted in Figure 5.6b).

The denotations of the two graphs are as follows:

$$\begin{aligned}
\mathsf{Exp}_1(x_1, x_2, y) &\triangleq \exists t.\ \mathsf{Merge}(+)(x_1, x_2, t) \wedge \mathsf{Proc}(\exp)(t, y) \\
&\equiv \exists t.\ t = x_1 + x_2 \wedge y = e^t \\
&\equiv y = e^{x_1 + x_2};
\end{aligned}$$

$$\begin{aligned}
\mathsf{Exp}_2(x_1, x_2, y) &\triangleq \exists t_1, t_2.\ \mathsf{Proc}(\exp)(x_1, t_1) \wedge \mathsf{Proc}(\exp)(x_2, t_2) \wedge \\
&\qquad\qquad \mathsf{Merge}(\times)(t_1, t_2, y) \\
&\equiv \exists t_1, t_2.\ t_1 = e^{x_1} \wedge t_2 = e^{x_2} \wedge y = t_1 t_2 \\
&\equiv y = e^{x_1} e^{x_2}.
\end{aligned}$$

Again, we can see that these two expressions are equivalent if exponentiation is a homomorphism from $(\mathbb{N}, +, 0)$ to $(\mathbb{R}, \times, 1)$, by property (5.14).

Proposition 5.3. *Exponentiation is a homomorphism.*

Proof. Follows directly from the identity $a^{m+n} = a^m a^n$. □

Since exponentiation is a homomorphism, property (5.14) implies that $e^{x_1+x_2} = e^{x_1} e^{x_2}$, so

$$\mathsf{Exp}_1(x_1, x_2, y) \equiv \mathsf{Exp}_2(x_1, x_2, y).$$

5.5.7.3 Merge–Processing transformation

In both the averaging and the exponentiation examples, we have shown pairs of equivalent task graphs. These are instances of a general bidirectional task graph transformation depicted graphically in Figure 5.7. We refer to this transformation as Merge–Processing. In the figure, P_1 and P_2 denote sub-graphs with single egress edges and Q denotes a sub-graph with a single ingress edge. Implicitly, the functions f, $\star$ and $\otimes$ are universally quantified.

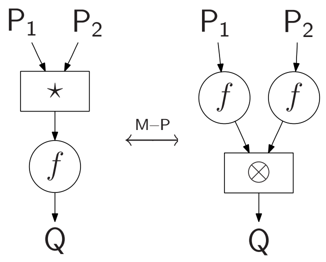


Figure 5.7: The Merge–Processing transformation.

In general, property (5.14) implies that merging before processing will yield the same result as processing before merging *if and only if the processing function is a monoid homomorphism*. Therefore this transformation is only permitted if f is a homomorphism from a monoid with operation $\star$ to a monoid with operation $\otimes$. This means that it is useful for a programmer to be able to express to a compiler when a processing function is a homomorphism, so the compiler knows when the transformation can be applied and is guaranteed not to affect the semantics of the program.

In the cases where a processing function is not a homomorphism, information would be lost if this transformation were to be effected, and this would mean that performing merging before processing would not yield the same result as processing before merging. As noted earlier, functions merely approximating homomorphisms will not in general give bit-identical values, but are sufficient for purpose if marked as homomorphisms when the programmer is satisfied that the transformations involving homomorphisms can be safely applied.

There are several consequences of employing the Merge–Processing task graph transformation. In the rightward direction, the transformation introduces an extra instance of the processing function, with each working on a different 'half' of the input data rather than working on the combination of both halves. This introduces the possibility for parallelism: processing each input independently.

The total amount of work performed will be roughly the same before and after the transformation. But depending on the relative sizes of the pre- and post-processed datatypes, if the two processing tasks are assigned to be executed on (or near) the respective processors which generated the inputs, the total amount of network traffic may be smaller. The computational complexities of the two merge functions may also cause the total execution time to differ.

To exemplify these consequences, consider an application which processes video data and extracts the number of vehicles seen in the images. A typical scenario might be as follows: there are ten video cameras, each connected directly to battery-powered local computers; there is also a central computer powered from the electricity grid; three of the cameras are connected by Ethernet to the central computer; the remaining cameras communicate using a wireless GPRS connection.

One distributed version of the application could involve appending the videos from all the cameras (*merging*) before the vehicle-recognition algorithm is run (*processing*). The compiler could assign the merge and processing tasks to the central computer, causing all of the video data to be transmitted over the network.

Applying the rightward transformation several times yields an alternative expression of the application in which the vehicle-recognition algorithm (*processing*) is run on each individual video, and then the number of vehicles is centrally summed to a single value (*merging*). The compiler could assign these several processing tasks to the computers attached to the cameras, and the merge task to the central computer. In this way, there is less network traffic generated since integer data is significantly smaller than video data. However, more battery energy would be consumed as the battery-powered computers now each perform a computationally-expensive task.

Suppose that the cost function indicates that energy consumption from batteries may be sacrificed for lower volumes of network traffic, but that execution time is important, and that the central computer's processor was five times faster than the battery-powered processors. Then, the optimal configuration would be for the videos produced by the cameras whose processors are connected over GPRS to be processed locally and for the other videos to be streamed over the network, merged and processed on the central computer.

Proposition 5.4. *The Merge–Processing transformation is sound.*

Proof. In order to formally prove the correctness of the transformation, we compare the denotations of the task graphs on the left and the right of the transformation. Function f is assumed to be a monoid homomorphism from $(S, \star, i_1)$ to $(T, \otimes, i_2)$.

$$\begin{aligned}
\mathsf{Left}^{\mathsf{M\text{–}P}}(x_1, x_2, y) &\triangleq \exists t.\ \mathsf{Merge}(\star)(x_1, x_2, t) \wedge \mathsf{Proc}(f)(t, y)\\
&\equiv \exists t.\ t = x_1 \star x_2 \wedge y = f(t)\\
&\equiv y = f(x_1 \star x_2);
\end{aligned}$$

$$\begin{aligned}
\mathsf{Right}^{\mathsf{M\text{–}P}}(x_1, x_2, x) &\triangleq \exists t_1, t_2.\ \mathsf{Proc}(f)(x_1, t_1) \wedge \mathsf{Proc}(f)(x_2, t_2) \wedge\\
&\qquad\qquad \mathsf{Merge}(\otimes)(t_1, t_2, y)\\
&\equiv \exists t_1, t_2.\ t_1 = f(x_1) \wedge t_2 = f(x_2) \wedge y = t_1 \otimes t_2\\
&\equiv y = f(x_1) \otimes f(x_2)\\
&\equiv y = f(x_1 \star x_2) \qquad \text{by (5.14)}.
\end{aligned}$$

Hence,

$$\mathsf{Left}^{\mathsf{M\text{–}P}}(x_1, x_2, y) \equiv \mathsf{Right}^{\mathsf{M\text{–}P}}(x_1, x_2, y). \tag{5.17}$$

□

5.5.7.4 Farm transformation

The symmetry between split tasks and merge tasks gives rise to a second transformation called Farm[6], depicted in Figure 5.8. A single processing task can be replaced by an array of processing tasks which each tackle a part of the input data. As with Merge–Processing, the processing function f must be a monoid homomorphism for the transformation to be valid.

[6]This transformation is called Farm by analogy to a similar construct termed 'farm' by Afshar [1, §5.1.2].

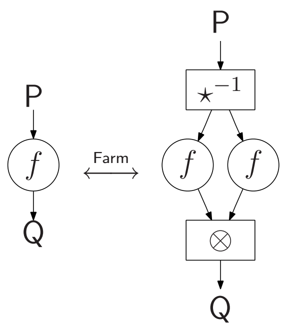


Figure 5.8: The Farm transformation.

This transformation facilitates the parallelisation of data processing by *partitioning* the data, so is particularly applicable to applications in grid computing where a large problem is commonly divided into a number of smaller problems processed in parallel. This paradigm is familiar from popular distributed computing applications such as SETI@home (see Section 2.2.2.3). The repeated use of this transformation allows a very large task graph to be generated from a small, easily designed and expressed task graph.

Proposition 5.5. *The Farm transformation is sound.*

Proof. We compare the denotations of the task graphs on the left and the right of the transformation. Function f is assumed to be a monoid homomorphism from $(S, \star, i_1)$ to $(T, \otimes, i_2)$.

$$\begin{aligned}
\mathsf{Left}^{\mathsf{Farm}}(x,y) &\triangleq \mathsf{Proc}(f)(x,y)\\
&\equiv y = f(x);
\end{aligned}$$

$$\begin{aligned}
\mathsf{Right}^{\mathsf{Farm}}(x,y) &\triangleq \exists x_1, x_2, y_1, y_2.\ \mathsf{Split}(\star^{-1})(x, x_1, x_2) \wedge\\
&\qquad \mathsf{Proc}(f)(x_1, y_1) \wedge \mathsf{Proc}(f)(x_2, y_2) \wedge\\
&\qquad \mathsf{Merge}(\otimes)(y_1, y_2, y)\\
&\equiv \exists x_1, x_2, y_1, y_2.\ (x_1, x_2) = \star^{-1}(x) \wedge\\
&\qquad y_1 = f(x_1) \wedge y_2 = f(x_2) \wedge\\
&\qquad y = y_1 \otimes y_2\\
&\equiv \exists x_1, x_2.\ (x_1, x_2) = \star^{-1}(x) \wedge y = f(x_1) \otimes f(x_2)\\
&\equiv \exists x_1, x_2.\ (x_1, x_2) = \star^{-1}(x) \wedge y = f(x_1 \star x_2) \qquad \text{by (5.14)}\\
&\equiv y = f(\star(\star^{-1}(x))\\
&\equiv y = f(x) \qquad \text{by (5.13)}.
\end{aligned}$$

Hence,

$$\mathsf{Left}^{\mathsf{Farm}}(x,y) \equiv \mathsf{Right}^{\mathsf{Farm}}(x,y). \tag{5.18}$$

□

5.5.7.5 Processing–Replication transformation

A third transformation, Processing–Replication, is similar to Merge–Processing described above, but involves swapping the order of processing and replication tasks rather than

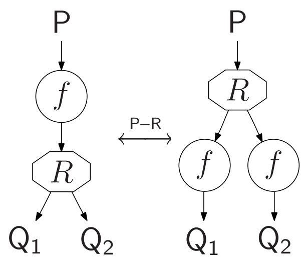


Figure 5.9: The Processing–Replication transformation.

processing and merge tasks. Rather than performing some processing and then replicating the result, we can replicate the input and process each replica individually. This transformation is depicted in Figure 5.9.

On the right of the transformation, the total amount of work is doubled. However, performing the rightward transformation may be desirable in minimising total execution time if the processors to which the replicas are sent are more powerful than that which generates the input value. The leftward transformation may be desirable in maximising the privacy of the originator of the data if processing it reduces its sensitivity, as the data can be processed locally before being replicated to untrusted parties.

Proposition 5.6. *The Processing–Replication transformation is sound.*

Proof. We compare the denotations of the task graphs on the left and the right of the transformation.

$$\begin{aligned}
\mathsf{Left}^{\mathsf{P\text{–}R}}(x, y_1, y_2) &\triangleq \exists y.\ \mathsf{Proc}(f)(x, y) \wedge \mathsf{Rep}(y, y_1, y_2) \\
&\equiv \exists y.\ y = f(x) \wedge y = y_1 \wedge y = y_2 \\
&\equiv y_1 = f(x) \wedge y_2 = f(x);
\end{aligned}$$

$$\begin{aligned}
\mathsf{Right}^{\mathsf{P\text{–}R}}(x, y_1, y_2) &\triangleq \exists x_1, x_2.\ \mathsf{Rep}(x, x_1, x_2) \wedge \\
&\qquad\qquad \mathsf{Proc}(f)(x_1, y_1) \wedge \mathsf{Proc}(f)(x_2, y_2) \\
&\equiv \exists x_1, x_2.\ x = x_1 \wedge x = x_2 \wedge y_1 = f(x_1) \wedge y_2 = f(x_2) \\
&\equiv y_1 = f(x) \wedge y_2 = f(x).
\end{aligned}$$

Hence,

$$\mathsf{Left}^{\mathsf{P\text{–}R}}(x, y_1, y_2) \equiv \mathsf{Right}^{\mathsf{P\text{–}R}}(x, y_1, y_2). \tag{5.19}$$

□

5.5.7.6 Split–Replication transformation

Transformation Split–Replication, shown in Figure 5.10, involves the exchange of replication and split tasks. Much like Processing–Replication, the total amount of work is doubled in the rightward transformation but may bring similar potential benefits if the resulting split tasks can be assigned to faster processors than the original split task could be.

Proposition 5.7. *The Split–Replication transformation is sound.*

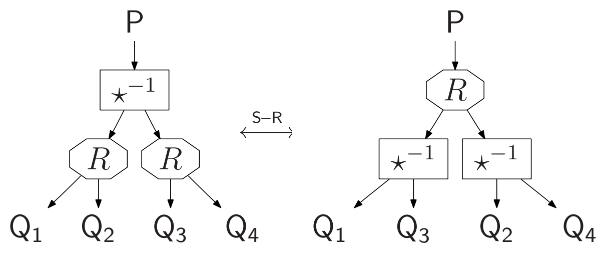


Figure 5.10: The Split–Replication transformation.

Proof. We compare the denotations of the task graphs on the left and the right of the transformation.

$$\begin{aligned}
\mathsf{Left}^{\mathsf{S\text{–}R}}(x, x_1, x_2, x_3, x_4) &\triangleq \exists t_1, t_2.\ \mathsf{Split}(\star^{-1})(x, t_1, t_2) \land \\
&\qquad\qquad \mathsf{Rep}(t_1, x_1, x_2) \land \mathsf{Rep}(t_2, x_3, x_4) \\
&\equiv \exists t_1, t_2.\ (t_1, t_2) = \star^{-1}(x) \land \\
&\qquad\qquad t_1 = x_1 \land t_1 = x_2 \land t_2 = x_3 \land t_2 = x_4 \\
&\equiv (x_1, x_3) = \star^{-1}(x) \land (x_2, x_4) = \star^{-1}(x);
\end{aligned}$$

$$\begin{aligned}
\mathsf{Right}^{\mathsf{S\text{–}R}}(x, x_1, x_2, x_3, x_4) &\triangleq \exists t_1, t_2.\ \mathsf{Rep}(x, t_1, t_2) \land \\
&\qquad\qquad \mathsf{Split}(\star^{-1})(t_1, x_1, x_3) \land \\
&\qquad\qquad \mathsf{Split}(\star^{-1})(t_2, x_2, x_4) \\
&\equiv \exists t_1, t_2.\ x = t_1 \land x = t_2 \land \\
&\qquad\qquad (x_1, x_3) = \star^{-1}(t_1) \land \\
&\qquad\qquad (x_2, x_4) = \star^{-1}(t_2) \\
&\equiv (x_1, x_3) = \star^{-1}(x) \land (x_2, x_4) = \star^{-1}(x).
\end{aligned}$$

Hence,

$$\mathsf{Left}^{\mathsf{S\text{–}R}}(x, x_1, x_2, x_3, x_4) \equiv \mathsf{Right}^{\mathsf{S\text{–}R}}(x, x_1, x_2, x_3, x_4). \tag{5.20}$$

□

5.5.7.7 Merge–Replication transformation

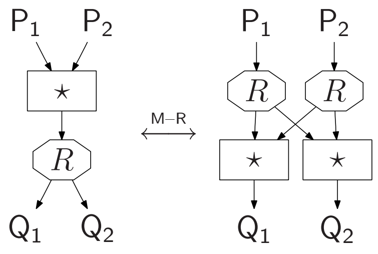


Figure 5.11: The Merge–Replication transformation.

Transformation Merge–Replication, shown in Figure 5.11, switches the order of merging and replication.

The rightward transformation doubles the number of tasks and the amount of work whilst introducing several new dataflow edges which could cause an increase in network traffic. The rightward transformation is likely to be beneficial when the merge operation is best carried out on the processors to which the recipients of the two outputs are assigned, provided the increase in network traffic is acceptable.

Proposition 5.8. *The* Merge–Replication *transformation is sound.*

Proof. We compare the denotations of the task graphs on the left and the right of the transformation.

$$\begin{aligned}
\mathsf{Left}^{\mathsf{M\text{–}R}}(x_1, x_2, y_1, y_2) &\triangleq \exists t.\ \mathsf{Merge}(\star)(x_1, x_2, t) \wedge \mathsf{Rep}(t, y_1, y_2) \\
&\equiv \exists t.\ t = x_1 \star x_2 \wedge t = y_1 \wedge t = y_2 \\
&\equiv y_1 = x_1 \star x_2 \wedge y_2 = x_1 \star x_2;
\end{aligned}$$

$$\begin{aligned}
\mathsf{Right}^{\mathsf{M\text{–}R}}(x_1, x_2, y_1, y_2) &\triangleq \exists t_1, t_2, t_3, t_4.\ \mathsf{Rep}(x_1, t_1, t_2) \wedge \\
&\qquad\qquad \mathsf{Rep}(x_2, t_3, t_4) \wedge \\
&\qquad\qquad \mathsf{Merge}(\star)(t_1, t_3, y_1) \wedge \\
&\qquad\qquad \mathsf{Merge}(\star)(t_2, t_4, y_2) \\
&\equiv \exists t_1, t_2, t_3, t_4.\ x_1 = t_1 \wedge x_1 = t_2 \wedge \\
&\qquad\qquad x_2 = t_3 \wedge x_2 = t_4 \wedge \\
&\qquad\qquad y_1 = t_1 \star t_3 \wedge \\
&\qquad\qquad y_2 = t_2 \star t_4 \\
&\equiv y_1 = x_1 \star x_2 \wedge y_2 = x_1 \star x_2.
\end{aligned}$$

Hence,

$$\mathsf{Left}^{\mathsf{M\text{–}R}}(x, x_1, x_2, x_3, x_4) \equiv \mathsf{Right}^{\mathsf{M\text{–}R}}(x, x_1, x_2, x_3, x_4). \tag{5.21}$$

$\square$

5.5.7.8 Merge-Reorder transformation

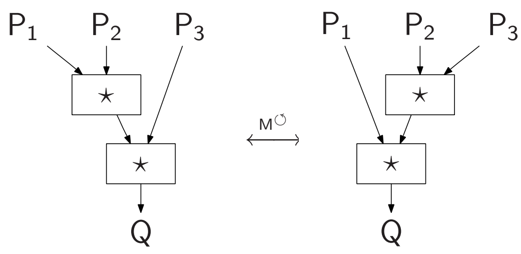


Figure 5.12: The Merge-Reorder transformation.

Transformation Merge-Reorder, shown in Figure 5.12, follows directly from the associativity of the functions embodied by merge tasks. This transformation is useful to alter how the merging of a large number of values takes place in a distributed manner.

It was noted in Section 5.5.3 that a chain of merge tasks combining n items of data can be drawn as a single n-ary merge task for convenience. This transformation indicates that

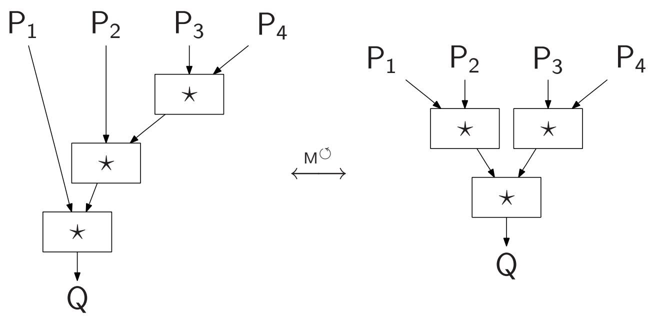


Figure 5.13: The equivalence of two merge trees combining four values.

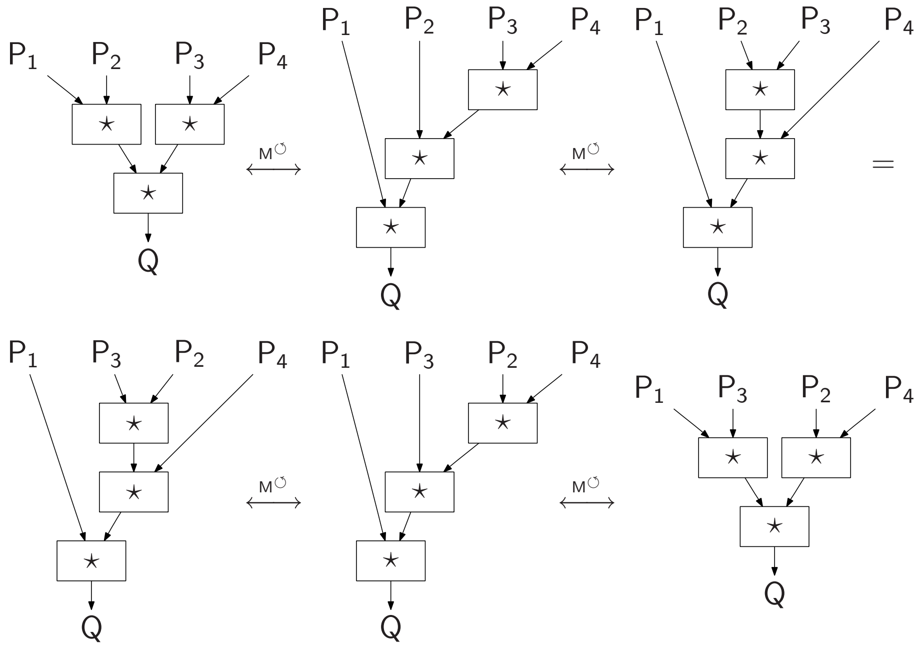


Figure 5.14: Re-structuring a tree, which merges four values, using repeated applications of Merge-Reorder.

the implicit structure of the chain of tasks is unimportant if all the tasks in the chain are assigned to the same processor. If assigned to different processors, this transformation can be used to alter the depth of the merge tree, which may impact on the amount of communication required and the total time required to perform the merging. For example, Figure 5.13 shows how a chain of three merge tasks can be transformed into a two-layer tree of merge tasks using a single instance of the Merge-Reorder transformation. Due to the introduction of parallelism, only $\log_2(n)$ merge stages are required rather than $n - 1$. Note, however, that a balanced tree of merge tasks will not necessarily be quicker to execute than a linear chain since the amount of work required in each task may differ depending on the sizes of the inputs.

This transformation can also be used to re-arrange the structure of a tree of merges that combines four values. A sequence of four Merge-Reorder steps achieving this is shown in Figure 5.14.

Proposition 5.9. *The* Merge-Reorder *transformation is sound.*

Proof. We compare the denotations of the task graphs on the left and the right of the transformation.

$$\begin{aligned}
\mathsf{Left}^{\mathsf{M}^{\circlearrowleft}}(x_1, x_2, x_3, x) &\triangleq \exists t.\ \mathsf{Merge}(\star)(x_1, x_2, t) \wedge \mathsf{Merge}(\star)(t, x_3, x)\\
&\equiv \exists t.\ t = x_1 \star x_2 \wedge x = t \star x_3\\
&\equiv x = (x_1 \star x_2) \star x_3;
\end{aligned}$$

$$\begin{aligned}
\mathsf{Right}^{\mathsf{M}^{\circlearrowleft}}(x_1, x_2, x_3, x) &\triangleq \exists t.\ \mathsf{Merge}(\star)(x_1, t, x) \wedge \mathsf{Merge}(\star)(x_2, x_3, t)\\
&\equiv \exists t.\ x = x_1 \star t \wedge t = x_2 \star x_3\\
&\equiv x = x_1 \star (x_2 \star x_3)\\
&\equiv x = (x_1 \star x_2) \star x_3 \qquad \text{by (5.5).}
\end{aligned}$$

Hence,

$$\mathsf{Left}^{\mathsf{M}^{\circlearrowleft}}(x_1, x_2, x_3, x) \equiv \mathsf{Right}^{\mathsf{M}^{\circlearrowleft}}(x_1, x_2, x_3, x). \tag{5.22}$$

□

5.5.7.9 Replication-Reorder transformation

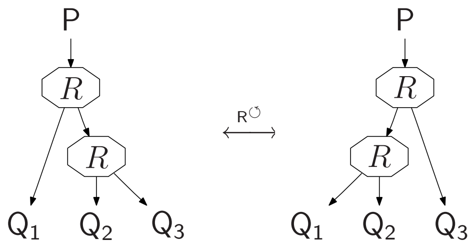


Figure 5.15: The Replication-Reorder transformation.

Transformation Replication-Reorder, shown in Figure 5.15, is analogous to Merge-Reorder above, and follows from the equivalence of the outputs from replication tasks and can be used to alter how the replication of a large number of values takes place.

Proposition 5.10. *The* Replication-Reorder *transformation is sound.*

Proof. We compare the denotations of the task graphs on the left and the right of the transformation.

$$\begin{aligned}
\mathsf{Left}^{\mathsf{R}^{\circlearrowleft}}(x, x_1, x_2, x_3) &\triangleq \exists t.\ \mathsf{Rep}(x, x_1, t) \wedge \mathsf{Rep}(t, x_2, x_3)\\
&\equiv \exists t.\ x = x_1 \wedge x = t \wedge t = x_2 \wedge t = x_3\\
&\equiv x = x_1 \wedge x = x_2 \wedge x = x_3;
\end{aligned}$$

$$\begin{aligned}
\mathsf{Right}^{\mathsf{R}^{\circlearrowleft}}(x, x_1, x_2, x_3) &\triangleq \exists t.\ \mathsf{Rep}(x, t, x_3) \wedge \mathsf{Rep}(t, x_1, x_2)\\
&\equiv \exists t.\ x = t \wedge x = x_3 \wedge t = x_1 \wedge t = x_2\\
&\equiv x = x_1 \wedge x = x_2 \wedge x = x_3.
\end{aligned}$$

Hence,

$$\mathsf{Left}^{\mathsf{R}\circlearrowleft}(x, x_1, x_2, x_3) \equiv \mathsf{Right}^{\mathsf{R}\circlearrowleft}(x, x_1, x_2, x_3). \tag{5.23}$$

□

5.5.7.10 Split–Merge transformation

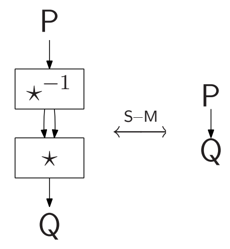


Figure 5.16: The Split–Merge transformation.

A further transformation, Split–Merge, depicted in Figure 5.16, follows from the definition of a split function for a particular datatype as an inverse of the merge function for that datatype.

The rightward transformation can be used to eliminate redundant work; the leftward transformation can be used to introduce a split and merge task that could be manipulated by a further transformation.

Proposition 5.11. *The Split–Merge transformation is sound.*

Proof. We compare the denotations of the task graphs on the left and the right of the transformation.

$$\begin{aligned}
\mathsf{Left}^{\mathsf{S\text{–}M}}(x, y) &\triangleq \exists t_1, t_2.\ \mathsf{Split}(\star^{-1})(x, t_1, t_2) \land \mathsf{Merge}(\star)(t_1, t_2, y)\\
&\equiv \exists t_1, t_2.\ (t_1, t_2) = \star^{-1}(x) \land y = t_1 \star t_2\\
&\equiv y = \star(\star^{-1}(x))\\
&\equiv y = x \qquad \text{by (5.13)};
\end{aligned}$$

$$\mathsf{Right}^{\mathsf{S\text{–}M}}(x, y) \triangleq y = x.$$

Hence,

$$\mathsf{Left}^{\mathsf{S\text{–}M}}(x, y) \equiv \mathsf{Right}^{\mathsf{S\text{–}M}}(x, y). \tag{5.24}$$

□

5.5.7.11 Merge–Split transformation

The complement of Split–Merge is Merge–Split, shown in Figure 5.17. The rightward transformation removes the redundant work involved in splitting the results of a merge operation and leaves the two inputs separate.

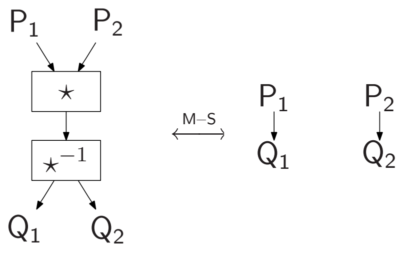


Figure 5.17: The Merge–Split transformation.

The denotations of the task graphs on the left and the right of the transformation are as follows:

$$\begin{aligned}\mathsf{Left}^{\mathsf{M\text{–}S}}(x_1,x_2,y_1,y_2) &\triangleq \exists t.\ \mathsf{Merge}(\star)(x_1,x_2,t) \wedge \mathsf{Split}(\star^{-1})(t,y_1,y_2)\\ &\equiv \exists t.\ t = x_1 \star x_2 \wedge (y_1,y_2) = \star^{-1}(t)\\ &\equiv (y_1,y_2) = \star^{-1}(x_1 \star x_2); \qquad (5.25)\end{aligned}$$

$$\mathsf{Right}^{\mathsf{M\text{–}S}}(x_1,x_2,y_1,y_2) \triangleq y_1 = x_1 \wedge y_2 = x_2. \qquad (5.26)$$

Whilst formula (5.26) implies formula (5.25), the converse is not true in general because $\star^{-1}$ is not necessarily a left-inverse of $\star$. Thus,

$$\mathsf{Left}^{\mathsf{M\text{–}S}}(x_1,x_2,y_1,y_2) \not\equiv \mathsf{Right}^{\mathsf{M\text{–}S}}(x_1,x_2,y_1,y_2),$$

so this transformation is *not sound*.

In words, the split task will not necessarily choose the same output values as were input to the merge task. For example, consider an application which is handling natural numbers, where the merge function is multiplication. Hence the monoid in question is $(\mathbb{N}, \times, 1)$. The merge task would multiply 6 by 4 to give 24, but the split task may factorise 24 into 3 and 8. Thus the value propagated from sub-tree P_1 to Q_1 could be changed from 6 to 3 in the rightward transformation.

The transformation becomes sound in the context of sub-trees Q_1 and Q_2 converging in a merge operation. Both (5.25) and (5.26) imply that

$$y_1 \star y_2 = x_1 \star x_2.$$

Thus, there may be occasions when it is useful to perform this transformation and it does not affect the application's output. This will be the case when the task graph can be transformed (by a sequence of zero or more transformations) into a task graph which has a merge operation that combines the values arriving into Q_1 and Q_2. In such a graph, the rightward Split–Merge transformation can then be applied to remove the adjacent split and merge tasks, as shown in Figure 5.18. Then the inverse of the sequence of transformations can be applied to return to a task graph equivalent to that which would arise by applying the rightward Merge–Split transformation.

An example of this procedure is depicted in Figure 5.19, where the single transformation Merge–Processing is required to transform the initial task graph into a state that admits

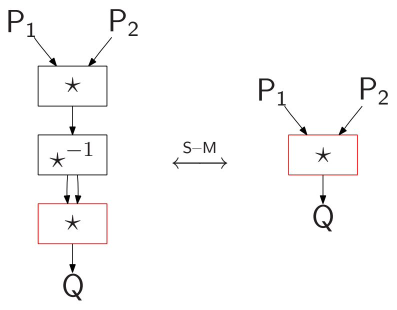


Figure 5.18: Mimicking the effect of the Merge–Split transformation, when the outputs are combined in a merge task (shown in red).

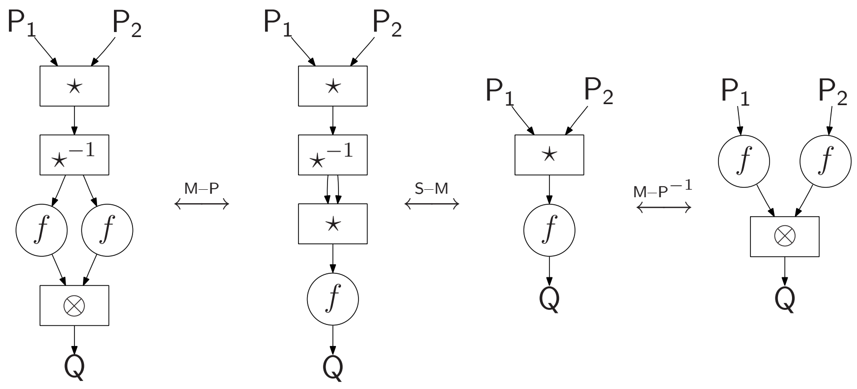


Figure 5.19: An example of achieving the effect of the Merge–Split transformation by performing a transformation, followed by Split–Merge, followed by the inverse of the first transformation.

the Split–Merge transformation. Performing the inverse of Merge–Processing completes a sequence of transformations equivalent to a single instance of Merge–Split. In the example, the function f is assumed to be a monoid homomorphism from $(S, \star, i_1)$ to $(T, \otimes, i_2)$.

Because the effect of performing Merge–Split in the cases where it is sound can be achieved using other transformations, Merge–Split is presented here as a transformation which a compiler may optionally choose to implement if it can confirm the safety of the operation. If it is not implemented then no sound task graph transformations are lost.

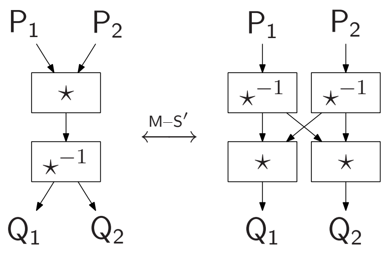


Figure 5.20: The Merge–Split′ transformation: an alternative to Merge–Split.

An alternative to the Merge–Split transformation is presented in Figure 5.20. This transformation is similarly unsound in the general case. We refer to this transformation as

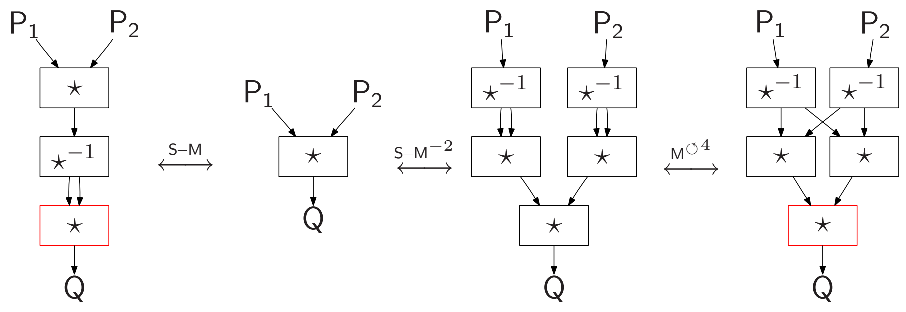


Figure 5.21: Mimicking the effect of the Merge–Split′ transformation, when the outputs are combined in a merge task (shown in red in the initial and final graphs). The second step involves two applications of the inverse of the Split–Merge transformation; the third step involves a sequence of steps akin to those shown in Figure 5.14.

Merge–Split′. The leftward direction of this transformation is particularly useful, especially in conjunction with the rightward direction of Merge–Split, to reduce the complexity of a set of interlinked split and merge tasks.

In this version of the transformation, the left side is as before:

$$\mathsf{Left}^{\mathsf{M\text{–}S}'}(x_1, x_2, y_1, y_2) \equiv \mathsf{Left}^{\mathsf{M\text{–}S}}(x_1, x_2, y_1, y_2).$$

The denotation of the right side is as follows:

$$\begin{aligned}
\mathsf{Right}^{\mathsf{M\text{–}S}'}(x_1, x_2, y_1, y_2) \;\triangleq\;& \exists t_1, t_2, t_3, t_4.\; \mathsf{Split}(\star^{-1})(x_1, t_1, t_2) \wedge \\
& \qquad\qquad\quad \mathsf{Split}(\star^{-1})(x_2, t_3, t_4) \wedge \\
& \qquad\qquad\quad \mathsf{Merge}(\star)(t_1, t_3, y_1) \wedge \\
& \qquad\qquad\quad \mathsf{Merge}(\star)(t_2, t_4, y_2) \\
\equiv\;& \exists t_1, t_2, t_3, t_4.\; (t_1, t_2) = \star^{-1}(x_1) \wedge \\
& \qquad\qquad\quad (t_3, t_4) = \star^{-1}(x_2) \wedge \\
& \qquad\qquad\quad y_1 = t_1 \star t_3 \wedge \\
& \qquad\qquad\quad y_2 = t_2 \star t_4.
\end{aligned} \tag{5.27}$$

As with the original Merge–Split transformation, Merge–Split′ is not sound but (5.27) also implies that

$$\begin{aligned}
y_1 \star y_2 &= (t_1 \star t_3) \star (t_2 \star t_4) \\
&= (t_1 \star t_2) \star (t_3 \star t_4) && \text{by assoc. (5.5) and commut. (5.6)} \\
&= x_1 \star x_2,
\end{aligned}$$

so this transformation also becomes sound in the context of the outputs converging at a merge task. As with Merge–Split, the effect of the transformation can be mimicked by a sequence of sound transformations when the task graph can be transformed into a state where the values arriving at Q_1 and Q_2 are merged. This sequence involves several applications of Split–Merge and Merge-Reorder, and is shown in Figure 5.21.

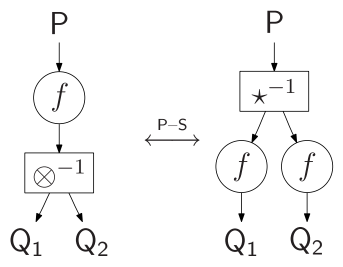


Figure 5.22: The Processing–Split transformation.

5.5.7.12 Processing–Split transformation

Analogous to Processing–Replication but for replication rather than split tasks, Processing–Split is depicted in Figure 5.22. As with Merge–Processing, the processing function f must be a monoid homomorphism from $(S, \star, i_1)$ to $(T, \otimes, i_2)$.

Since the split task $\star^{-1}$ partitions its input value into two parts, the rightward transformation permits earlier parallelism and the two processing functions work on separate data. The total amount of work performed is roughly the same before and after the transformation, as with Merge–Processing, but may similarly be affected by the relative complexities of the pre- and post-processed datatypes and the sizes of the two split functions.

We compare the denotations of the task graphs on the left and the right of the transformation. Function f is assumed to be a monoid homomorphism from $(S, \star, i_1)$ to $(T, \otimes, i_2)$.

$$\begin{aligned}\mathsf{Left}^{\mathsf{P\text{–}S}}(x, y_1, y_2) &\triangleq \exists y.\ \mathsf{Proc}(f)(x, y) \wedge \mathsf{Split}(\otimes^{-1})(y, y_1, y_2)\\ &\equiv \exists y.\ y = f(x) \wedge (y_1, y_2) = \otimes^{-1}(y)\\ &\equiv (y_1, y_2) = \otimes^{-1}(f(x)); \end{aligned} \tag{5.28}$$

$$\begin{aligned}\mathsf{Right}^{\mathsf{P\text{–}S}}(x, y_1, y_2) &\triangleq \exists x_1, x_2.\ \mathsf{Split}(\star^{-1})(x, x_1, x_2) \wedge\\ &\qquad\qquad \mathsf{Proc}(f)(x_1, y_1) \wedge \mathsf{Proc}(f)(x_2, y_2)\\ &\equiv \exists x_1, x_2.\ (x_1, x_2) = \star^{-1}(x) \wedge\\ &\qquad\qquad y_1 = f(x_1) \wedge y_2 = f(x_2). \end{aligned} \tag{5.29}$$

Since $\mathsf{Left}^{\mathsf{P\text{–}S}}(x, y_1, y_2) \not\equiv \mathsf{Right}^{\mathsf{P\text{–}S}}(x, y_1, y_2)$, this transformation is *not sound*.

If, however, y_1 and y_2 are merged by $\otimes$ when sub-trees Q_1 and Q_2 converge, the denotations become equivalent. On the left, (5.28) implies that

$$y_1 \otimes y_2 = f(x).$$

On the right, (5.29) implies that

$$\begin{aligned} y_1 \otimes y_2 &= f(x_1) \otimes f(x_2)\\ &= f(x_1 \star x_2) \quad \text{by (5.14)}\\ &= f(x). \end{aligned}$$

Therefore, if a task graph can be transformed so that the outputs are merged in this way, we can again mimic the effect of Processing–Split with a combination of other, sound transformations. This is shown in Figure 5.23. Hence, Processing–Split is optional for a compiler to implement.

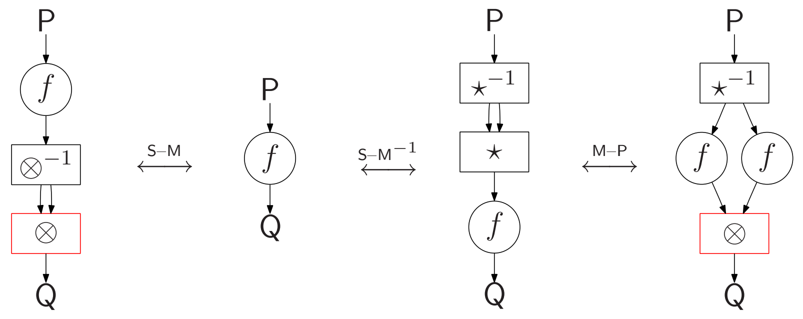


Figure 5.23: Mimicking the effect of the Processing–Split transformation, when the outputs are combined in a merge task (shown in red in the initial and final graphs).

5.5.7.13 Split-Reorder transformation

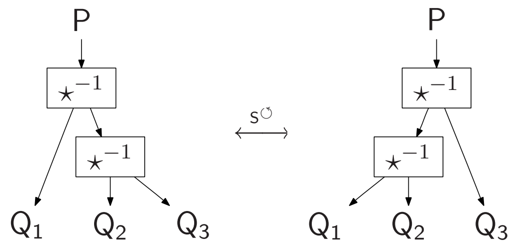


Figure 5.24: The Split-Reorder transformation.

Like Merge-Reorder and Replication-Reorder, transformation Split-Reorder, shown in Figure 5.24, can be used to adjust a chain of split tasks. The values b, c and d propagated to sub-graphs Q_1, Q_2 and Q_3 respectively are such that $a = b \star c \star d$ where a is the value output from sub-graph P.

The denotations of the task graphs on the left and the right of the transformation are as follows:

$$\begin{aligned}\mathsf{Left}^{s^{\circlearrowleft}}(x, x_1, x_2, x_3) &\triangleq \exists t.\ \mathsf{Split}(\star^{-1})(x, x_1, t) \wedge \mathsf{Split}(\star^{-1})(t, x_2, x_3)\\ &\equiv \exists t.\ (x_1, t) = \star^{-1}(x) \wedge (x_2, x_3) = \star^{-1}(t); \end{aligned} \tag{5.30}$$

$$\begin{aligned}\mathsf{Right}^{s^{\circlearrowleft}}(x, x_1, x_2, x_3) &\triangleq \exists t.\ \mathsf{Split}(\star^{-1})(x, t, x_3) \wedge \mathsf{Split}(\star^{-1})(t, x_1, x_2)\\ &\equiv \exists t.\ (t, x_3) = \star^{-1}(x) \wedge (x_1, x_2) = \star^{-1}(t). \end{aligned} \tag{5.31}$$

Since $\mathsf{Left}^{s^{\circlearrowleft}}(x, x_1, x_2, x_3) \not\equiv \mathsf{Right}^{s^{\circlearrowleft}}(x, x_1, x_2, x_3)$, this transformation is *not sound*. The output of the task graph will potentially be different since the transformation changes which of the two parts of the split input is further split by the second split task. For example, if the monoid is $(\mathbb{N}, \times, 1)$ and the value output from sub-graph P is 48, the values propagated to sub-graphs Q_1, Q_2 and Q_3 could be 12, 2 and 2 on the left of the transformation and 3, 4 and 4 on the right.

As with Merge–Split and Processing–Split, this transformation is included because it is sound provided that the sub-graphs Q_1, Q_2 and Q_3 converge in a further merge operation.

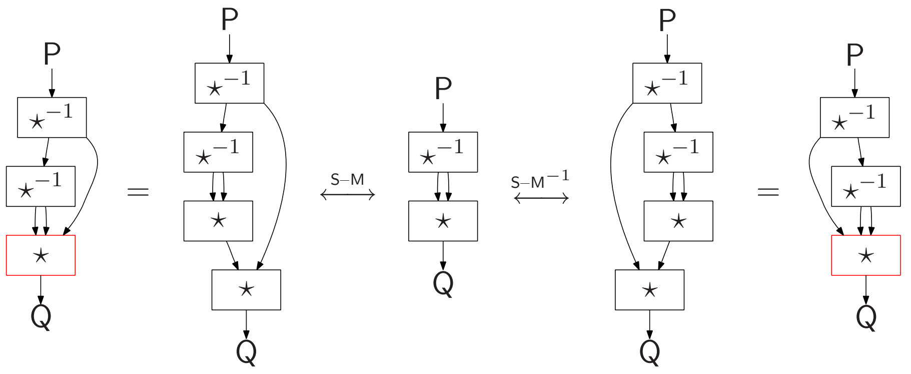


Figure 5.25: Mimicking the effect of the Split-Reorder transformation, when the outputs are combined in a ternary merge task (shown in red in the initial and final graphs).

Both (5.30) and (5.31) imply that

$$x_1 \star x_2 \star x_3 = x.$$

Indeed, as before, when the outputs are merged, the effect of Split-Reorder can be mimicked by a sequence of other, sound transformations. This is shown in Figure 5.25. This transformation may therefore be optionally supported by a compiler if it can confirm the safety of the operation.

We summarise all twelve transformations denotationally in Figure 5.26.

5.5.8 Redundancy

The set of transformations described above includes some redundancy. For example, Farm can be expressed in terms of two other transformations—Split–Merge and Merge–Processing—as shown in Figure 5.27. Whilst Farm could therefore be safely overlooked and cause us to lose no flexibility, it is desirable for a compiler to implement it so that task graphs with matching sub-graphs can be transformed as a unit in a single step.

5.5.9 Optimising transformations for n-ary tasks

The transformations described above all relate to binary merge, replication and split tasks. However, given that applications will commonly involve chains of these tasks to combine, replicate or partition larger numbers of values, it is useful to have n-ary analogues of the transformations to act directly on n-ary tasks. This enables the whole semantic structure of an n-ary task to be considered (and transformed) as a unit rather than needing to break it down into its constituent binary tasks.

Conveniently, it transpires not only that the obvious extensions to the sound binary transformations are also sound, but moreover that they can be expressed in terms of the binary transformations. In Appendix H, we graphically show the ternary versions of the transformations and how they are expressed in terms of the transformations introduced

$$
\begin{aligned}
\mathsf{Left}^{\mathsf{M\text{–}P}}(x_1, x_2, y) &\triangleq \exists t.\ \mathsf{Merge}(\star)(x_1, x_2, t) \wedge \mathsf{Proc}(f)(t, y) \\
\mathsf{Right}^{\mathsf{M\text{–}P}}(x_1, x_2, y) &\triangleq \exists t_1, t_2.\ \mathsf{Proc}(f)(x_1, t_1) \wedge \mathsf{Proc}(f)(x_2, t_2) \wedge \\
&\qquad\qquad \mathsf{Merge}(\otimes)(t_1, t_2, y) \\
\mathsf{Left}^{\mathsf{Farm}}(x, y) &\triangleq \mathsf{Proc}(f)(x, y) \\
\mathsf{Right}^{\mathsf{Farm}}(x, y) &\triangleq \exists x_1, x_2, y_1, y_2.\ \mathsf{Split}(\star^{-1})(x, x_1, x_2) \wedge \\
&\qquad\qquad \mathsf{Proc}(f)(x_1, y_1) \wedge \mathsf{Proc}(f)(x_2, y_2) \wedge \\
&\qquad\qquad \mathsf{Merge}(\otimes)(y_1, y_2, y) \\
\mathsf{Left}^{\mathsf{P\text{–}R}}(x, y_1, y_2) &\triangleq \exists y.\ \mathsf{Proc}(f)(x, y) \wedge \mathsf{Rep}(y, y_1, y_2) \\
\mathsf{Right}^{\mathsf{P\text{–}R}}(x, y_1, y_2) &\triangleq \exists x_1, x_2.\ \mathsf{Rep}(x, x_1, x_2) \wedge \\
&\qquad\qquad \mathsf{Proc}(f)(x_1, y_1) \wedge \mathsf{Proc}(f)(x_2, y_2) \\
\mathsf{Left}^{\mathsf{S\text{–}R}}(x, x_1, x_2, x_3, x_4) &\triangleq \exists t_1, t_2.\ \mathsf{Split}(\star^{-1})(x, t_1, t_2) \wedge \\
&\qquad\qquad \mathsf{Rep}(t_1, x_1, x_2) \wedge \mathsf{Rep}(t_2, x_3, x_4) \\
\mathsf{Right}^{\mathsf{S\text{–}R}}(x, x_1, x_2, x_3, x_4) &\triangleq \exists t_1, t_2.\ \mathsf{Rep}(x, t_1, t_2) \wedge \\
&\qquad\qquad \mathsf{Split}(\star^{-1})(t_1, x_1, x_3) \wedge \mathsf{Split}(\star^{-1})(t_2, x_2, x_4) \\
\mathsf{Left}^{\mathsf{M\text{–}R}}(x_1, x_2, y_1, y_2) &\triangleq \exists t.\ \mathsf{Merge}(\star)(x_1, x_2, t) \wedge \mathsf{Rep}(t, y_1, y_2) \\
\mathsf{Right}^{\mathsf{M\text{–}R}}(x_1, x_2, y_1, y_2) &\triangleq \exists t_1, t_2, t_3, t_4.\ \mathsf{Rep}(x_1, t_1, t_2) \wedge \mathsf{Rep}(x_2, t_3, t_4) \wedge \\
&\qquad\qquad \mathsf{Merge}(\star)(t_1, t_3, y_1) \wedge \\
&\qquad\qquad \mathsf{Merge}(\star)(t_2, t_4, y_2) \\
\mathsf{Left}^{\mathsf{S\text{–}M}}(x, y) &\triangleq \exists t_1, t_2.\ \mathsf{Split}(\star^{-1})(x, t_1, t_2) \wedge \mathsf{Merge}(\star)(t_1, t_2, y) \\
\mathsf{Right}^{\mathsf{S\text{–}M}}(x, y) &\triangleq y = x \\
\mathsf{Left}^{\mathsf{M\text{–}S}}(x_1, x_2, y_1, y_2) &\triangleq \exists t.\ \mathsf{Merge}(\star)(x_1, x_2, t) \wedge \mathsf{Split}(\star^{-1})(t, y_1, y_2) \\
\mathsf{Right}^{\mathsf{M\text{–}S}}(x_1, x_2, y_1, y_2) &\triangleq y_1 = x_1 \wedge y_2 = x_2 \\
\mathsf{Left}^{\mathsf{M\text{–}S}'}(x_1, x_2, y_1, y_2) &\triangleq \exists t.\ \mathsf{Merge}(\star)(x_1, x_2, t) \wedge \mathsf{Split}(\star^{-1})(t, y_1, y_2) \\
\mathsf{Right}^{\mathsf{M\text{–}S}'}(x_1, x_2, y_1, y_2) &\triangleq \exists t_1, t_2, t_3, t_4.\ \mathsf{Split}(\star^{-1})(x_1, t_1, t_2) \wedge \\
&\qquad\qquad \mathsf{Split}(\star^{-1})(x_2, t_3, t_4) \wedge \\
&\qquad\qquad \mathsf{Merge}(\star)(t_1, t_3, y_1) \wedge \mathsf{Merge}(t_2, t_4, y_2) \\
\mathsf{Left}^{\mathsf{P\text{–}S}}(x, y_1, y_2) &\triangleq \exists y.\ \mathsf{Proc}(f)(x, y) \wedge \mathsf{Split}(\otimes^{-1})(y, y_1, y_2) \\
\mathsf{Right}^{\mathsf{P\text{–}S}}(x, y_1, y_2) &\triangleq \exists x_1, x_2.\ \mathsf{Split}(\star^{-1})(x, x_1, x_2) \wedge \\
&\qquad\qquad \mathsf{Proc}(f)(x_1, y_1) \wedge \mathsf{Proc}(f)(x_2, y_2) \\
\mathsf{Left}^{\mathsf{M}^{\circlearrowleft}}(x_1, x_2, x_3, x) &\triangleq \exists t.\ \mathsf{Merge}(\star)(x_1, x_2, t) \wedge \mathsf{Merge}(\star)(t, x_3, x) \\
\mathsf{Right}^{\mathsf{M}^{\circlearrowleft}}(x_1, x_2, x_3, x) &\triangleq \exists t.\ \mathsf{Merge}(\star)(x_2, x_3, t) \wedge \mathsf{Merge}(\star)(x_1, t, x) \\
\mathsf{Left}^{\mathsf{R}^{\circlearrowleft}}(x, x_1, x_2, x_3) &\triangleq \exists t.\ \mathsf{Rep}(x, x_1, t) \wedge \mathsf{Rep}(t, x_2, x_3) \\
\mathsf{Right}^{\mathsf{R}^{\circlearrowleft}}(x, x_1, x_2, x_3) &\triangleq \exists t.\ \mathsf{Rep}(x, t, x_3) \wedge \mathsf{Rep}(t, x_1, x_2) \\
\mathsf{Left}^{\mathsf{S}^{\circlearrowleft}}(x, x_1, x_2, x_3) &\triangleq \exists t.\ \mathsf{Split}(\star^{-1})(x, x_1, t) \wedge \mathsf{Split}(\star^{-1})(t, x_2, x_3) \\
\mathsf{Right}^{\mathsf{S}^{\circlearrowleft}}(x, x_1, x_2, x_3) &\triangleq \exists t.\ \mathsf{Split}(\star^{-1})(x, t, x_3) \wedge \mathsf{Split}(\star^{-1})(t, x_1, x_2)
\end{aligned}
$$

Figure 5.26: Denotations of task graph transformations.

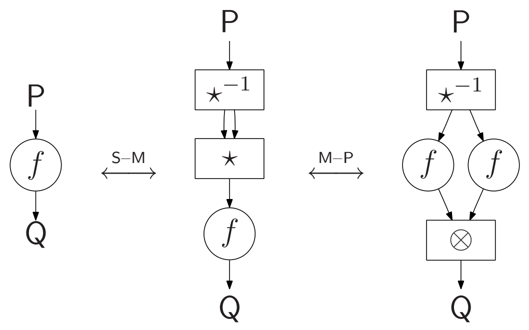


Figure 5.27: The Split–Merge and Merge–Processing transformations can be used in conjunction to achieve the same result as the Farm transformation.

above. The algebraic proofs of soundness for each of the n-ary transformations are given in Appendix I. A summary of the denotations of the transformations is given in Figure 5.28.

5.5.10 Example of using transformations: computing π

A naïve approach to computing the value of π is to measure the area of a circle of a known radius r. An implementation of this technique to compute an approximation to π involves iterating over the pixels in a square and counting the number of pixels (x, y) such that $x^2 + y^2 \leq r^2$.

The datatypes the application will use are as follows:

- A mergeable datatype containing the set of pixels to examine, represented by the commutative monoid $(\mathcal{P}(\mathbb{N} \times \mathbb{N}), \cup, \varnothing)$. The input to the application is drawn from this datatype, and consists of a set containing all pixels (x, y) such that $x, y \in [-r, r]$.
- A mergeable datatype containing the number of pixels found to lie within the bounds of the circle, represented by the commutative monoid $(\mathbb{N}, +, 0)$.
- A mergeable datatype containing the approximation to π, represented by the commutative monoid $(\mathbb{R}, +, 0)$.

A split function, σ, is defined for the monoid $(\mathcal{P}(\mathbb{N} \times \mathbb{N}), \cup, \varnothing)$:

$$\sigma(S) = (S_1, S_2) \quad \text{such that } S_1 \cup S_2 = S \text{ and } S_1 \cap S_2 = \varnothing. \tag{5.32}$$

An implementation of this function should favour balanced splitting, i.e. $|S_1| \approx |S_2|$.

The function f_r to count the pixels lying within the bounds of the circle of radius r is defined as

$$\begin{aligned} f_r(\varnothing) &= 0 & (5.33)\\ f_r(\{(x, y)\} \cup ps) &= \delta_r^{x,y} + f_r(ps), & (5.34) \end{aligned}$$

where

$$\delta_r^{x,y} \triangleq \begin{cases} 1 & \text{if } x^2 + y^2 \leq r^2, \\ 0 & \text{otherwise.} \end{cases}$$

$$
\begin{aligned}
\mathsf{Left}^{\mathsf{M\text{–}P}}_n(x_1,\ldots,x_n,y) &\triangleq \exists t.\ \mathsf{Merge}_n(\star)(x_1,\ldots,x_n,t) \wedge \mathsf{Proc}(f)(t,y) \\
\mathsf{Right}^{\mathsf{M\text{–}P}}_n(x_1,\ldots,x_n,y) &\triangleq \exists t_1,\ldots,t_n.\ \mathsf{Merge}_n(\otimes)(t_1,\ldots,t_n,y) \wedge \\
&\qquad\qquad\quad \textstyle\bigwedge_{i=1}^{n} \mathsf{Proc}(f)(x_i,t_i) \\
\mathsf{Left}^{\mathsf{Farm}}_n(x,y) &\triangleq \mathsf{Proc}(f)(x,y) \\
\mathsf{Right}^{\mathsf{Farm}}_n(x,y) &\triangleq \exists x_1,\ldots,x_n,y_1,\ldots,y_n.\ \mathsf{Split}_n(\star^{-1})(x,x_1,\ldots,x_n) \wedge \\
&\qquad\qquad\qquad\qquad\qquad \textstyle\bigwedge_{i=1}^{n} \mathsf{Proc}(f)(x_i,y_i) \wedge \\
&\qquad\qquad\qquad\qquad\qquad \mathsf{Merge}(\otimes)(y_1,\ldots,y_n,y) \\
\mathsf{Left}^{\mathsf{P\text{–}R}}_n(x,y_1,\ldots,y_n) &\triangleq \exists t.\ \mathsf{Proc}(f)(x,t) \wedge \mathsf{Rep}_n(t,y_1,\ldots,y_n) \\
\mathsf{Right}^{\mathsf{P\text{–}R}}_n(x,y_1,\ldots,y_n) &\triangleq \exists t_1,\ldots,t_n.\ \mathsf{Rep}_n(x,t_1,\ldots,t_n) \wedge \\
&\qquad\qquad\quad \textstyle\bigwedge_{i=1}^{n} \mathsf{Proc}(f)(t_i,y_i) \\
\mathsf{Left}^{\mathsf{S\text{–}M}}_n(x,y) &\triangleq \exists t_1,\ldots,t_n.\ \mathsf{Split}_n(\star^{-1})(x,t_1,\ldots,t_n) \wedge \\
&\qquad\qquad\quad \mathsf{Merge}(\star)(t_1,\ldots,t_n,y) \\
\mathsf{Right}^{\mathsf{S\text{–}M}}_n(x,y) &\triangleq y = x \\
\mathsf{Left}^{\mathsf{M\text{–}S}}_n(x_1,\ldots,x_n,y_1,\ldots,y_n) &\triangleq \exists t.\ \mathsf{Merge}_n(\star)(x_1,\ldots,x_n,t) \wedge \\
&\qquad\ \ \mathsf{Split}_n(\star^{-1})(t,y_1,\ldots,y_n) \\
\mathsf{Right}^{\mathsf{M\text{–}S}}_n(x_1,\ldots,x_n,y_1,\ldots,y_n) &\triangleq \bigwedge_{i=1}^{n} y_i = x_i \\
\mathsf{Left}^{\mathsf{P\text{–}S}}_n(x,y_1,\ldots,y_n) &\triangleq \exists t.\ \mathsf{Proc}(f)(x,t) \wedge \mathsf{Split}_n(\otimes^{-1})(t,y_1,\ldots,y_n) \\
\mathsf{Right}^{\mathsf{P\text{–}S}}_n(x,y_1,\ldots,y_n) &\triangleq \exists t_1,\ldots,t_n.\ \mathsf{Split}_n(\star^{-1})(x,t_1,\ldots,t_n) \wedge \\
&\qquad\qquad\quad \textstyle\bigwedge_{i=1}^{n} \mathsf{Proc}(f)(t_i,y_i) \\
\mathsf{Left}^{\mathsf{S\text{–}R}}_{m,n}(x,x_{1,1},\ldots,x_{m,n}) &\triangleq \exists t_1,\ldots,t_m.\ \mathsf{Split}_m(\star^{-1})(x,t_1,\ldots,t_m) \wedge \\
&\qquad\qquad\quad \textstyle\bigwedge_{i=1}^{m} \mathsf{Rep}_n(t_i,x_{i,1},\ldots,x_{i,n}) \\
\mathsf{Right}^{\mathsf{S\text{–}R}}_{m,n}(x,x_{1,1},\ldots,x_{m,n}) &\triangleq \exists t_1,\ldots,t_n.\ \mathsf{Rep}_n(x,t_1,\ldots,t_n) \wedge \\
&\qquad\qquad\quad \textstyle\bigwedge_{i=1}^{n} \mathsf{Split}_m(\star^{-1})(t_i,x_{1,i},\ldots,x_{m,i}) \\
\mathsf{Left}^{\mathsf{M\text{–}R}}_{m,n}(x_1,\ldots,x_m,y_1,\ldots,y_n) &\triangleq \exists t.\ \mathsf{Merge}_m(\star)(x_1,\ldots,x_m,t) \wedge \mathsf{Rep}_n(t,y_1,\ldots,y_n) \\
\mathsf{Right}^{\mathsf{M\text{–}R}}_{m,n}(x_1,\ldots,x_m,y_1,\ldots,y_n) &\triangleq \exists t_{1,1}\ldots,t_{m,n}.\ \textstyle\bigwedge_{i=1}^{m} \mathsf{Rep}_n(x_i,t_{i,1},\ldots,t_{i,n}) \wedge \\
&\qquad\qquad\qquad\quad \textstyle\bigwedge_{i=1}^{n} \mathsf{Merge}_m(t_{1,i},\ldots,t_{m,i},y_i)
\end{aligned}
$$

Figure 5.28: Denotations of n-ary transformations.

Proposition 5.12. *Function f_r is a monoid homomorphism from $(\mathcal{P}(\mathbb{N} \times \mathbb{N}), \cup, \varnothing)$ to $(\mathbb{N}, +, 0)$.*

Proof. In a similar manner to the proof of Proposition 5.2, we prove $f_r(a \cup b) = f_r(a) + f_r(b)$ by induction on the structure of a, which is a set of ordered pairs of natural numbers.

Consider the base case, where $a = \varnothing$:

$$\begin{array}{rcll} f_r(\varnothing \cup b) & = & f_r(b) & \\ & = & 0 + f_r(b) & \text{because 0 is an identity and (5.12)} \\ & = & f_r(\varnothing) + f_r(b) & \text{by definition of } f_r\text{, (5.33).} \end{array}$$

For the inductive step, we assume $f_r(a \cup b) = f_r(a) + f_r(b)$. Then,

$$\begin{array}{rcll} f_r((\{(x,y)\} \cup a) \cup b) & = & f_r(\{(x,y)\} \cup (a \cup b)) & \text{by assoc. of } \cup \text{ and (5.5)} \\ & = & \delta_r^{x,y} + f_r(a \cup b) & \text{by defn. of } f_r\text{, (5.34)} \\ & = & \delta_r^{x,y} + (f_r(a) + f_r(b)) & \text{by induction hypothesis} \\ & = & (\delta_r^{x,y} + f_r(a)) + f_r(b) & \text{by assoc. of } + \text{ and (5.5)} \\ & = & f_r(\{(x,y)\} \cup a) + f_r(b) & \text{by defn. of } f_r\text{, (5.34).} \end{array}$$

□

The function g_r to convert a pixel-count into an approximation to π is defined as

$$g_r(x) = \frac{x}{r^2}. \tag{5.35}$$

Proposition 5.13. *Function g_r is a monoid homomorphism from $(\mathbb{N}, +, 0)$ to $(\mathbb{R}, +, 0)$.*

Proof.

$$\begin{array}{rcll} g_r(x_1 + x_2) & = & \frac{x_1 + x_2}{r^2} & \text{by definition of } g_r\text{, (5.35)} \\ & = & \frac{x_1}{r^2} + \frac{x_2}{r^2} & \\ & = & g_r(x_1) + g_r(x_2) & \text{by definition of } g_r\text{, (5.35).} \end{array}$$

□

An initial task graph that the application's designer might create is shown in Figure 5.29. It consists of a source and a sink, connected via processing tasks embodying the functions f_r and g_r. The source outputs a single value, the set V which contains all pairs (x, y) such that $x, y \in [-r, r]$, which has cardinality $(2r + 1)^2$.

To this initial task graph, a compiler may apply various optimisations. For example, if supplied with a resource graph containing three fast processors, and a cost function which minimises execution time, the compiler may choose to perform the ternary Farm transformation to partition the problem and parallelise the execution of the f_r processing task into three such tasks. Subsequently, if g_r is defined such that its execution time is proportional to the magnitude of its input, the compiler could perform the ternary Merge–Processing transformation to execute g_r individually on each of the outputs from f_r.

This chain of transformations is shown in Figure 5.30. Overall, the transformations have the effect of parallelising the computation of π so that a result is returned in minimal time.

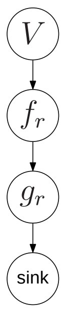


Figure 5.29: Initial task graph for application to compute π.

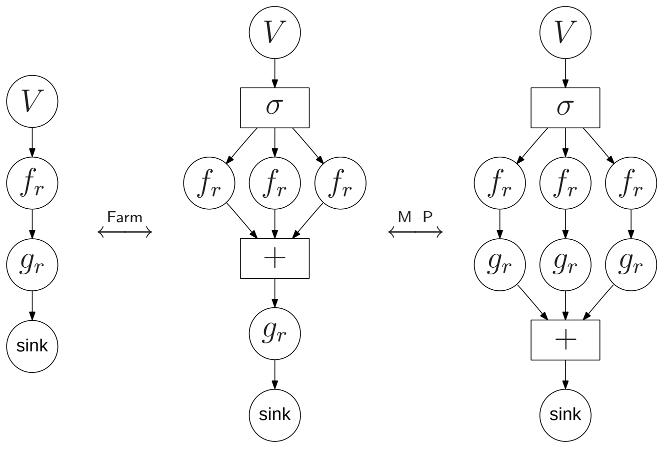


Figure 5.30: A sequence of two transformations applied to the task graph for the π application.

Rather than containing three processors, if the resource graph were able to contain an unbounded number of processors, we would still expect only a finite number of them to be assigned tasks, assuming a realistic model of communication costs. This is because the cost of communicating to and from a distant processor would be outweighed by the cost of having fewer partitions. However, in practice resource graphs are finite; this too provides an upper bound on the number of partitions the set of pixels is split into.

5.6 Expressiveness of task graphs

Proposition 5.14. *A task graph composed of source, sink, processing, merge, split and replication tasks is sufficient to express any algorithm.*

In order to prove this proposition, we need to consider whether the general function of type $\alpha_1 \times \alpha_2 \times \ldots \times \alpha_n \rightarrow \beta_1 \times \beta_2 \times \ldots \times \beta_m$, where $n, m \geq 0$, can be expressed by a graph of tasks, since an algorithm could involve the use of arbitrary functions. We must also consider whether tasks can be connected by edges in an arbitrary manner. We will start by addressing the first issue and return to consider the second in Section 5.6.2.

When $n = 0$ and $m = 1$, the function can be represented by a single source task. When $n = 1$ and $m = 0$, the function can be represented by a single sink task. When $n = 1$ and

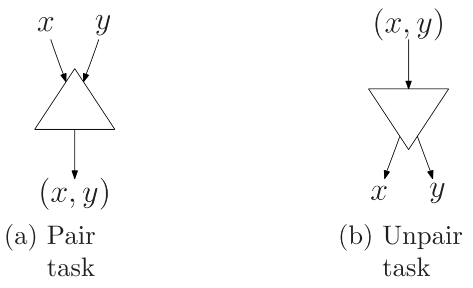


Figure 5.31: Two new kinds of task.

$m = 1$, the function can be represented by a single processing task. But when $n > 1$ or $m > 1$ it is not so straightforward because the definitions of processing tasks only permit a single input and a single output.

One approach to addressing this issue could be to extend the definition of processing tasks to allow *multiple* inputs of different types. However, this would complicate the theory of processing tasks as monoid homomorphisms presented above. It would also not be clear how to amend the transformations appropriately.

Therefore, instead we consider adding two new kinds of task graph element, *pair* and *unpair* tasks.

5.6.1 Pair and unpair tasks

The proposed pair and unpair tasks are depicted in Figure 5.31. A pair task is a function which takes values of types α and β and pairs them up into a single value of type $\alpha \times \beta$. Conversely, an unpair task is a function which takes a value of type $\alpha \times \beta$ and decomposes it into separate values of types α and β.

The denotations of pair and unpair tasks are as follows:

$$\mathsf{Pair}(x_1, x_2, y) \triangleq y = (x_1, x_2), \tag{5.36}$$
$$\mathsf{Unpair}(x, y_1, y_2) \triangleq (y_1, y_2) = x. \tag{5.37}$$

We showed in Section 5.5.3 how the primitive kinds of task could be generalised to n-ary versions, and how these could be expressed in terms of the binary versions. Similarly, we can do the same for pair and unpair tasks.

In general, an n-tuple $(x_1, \ldots, x_n)$ can be thought of as a series of nested pairs,

$$(x_1, x_2, x_3, \ldots, x_n) = (\ldots((x_1, x_2), x_3)\ldots, x_n).$$

This leads us to define an n-ary pair task[7] in terms of a chain of binary pair tasks. We show the ternary instance in Figure 5.32. Similarly, an n-ary unpair task can be defined in terms of a chain of binary unpair tasks.

[7]Perhaps better referred to as an n-tuple constructor task, but we will stick with the term 'n-ary pair task' to make the relationship between the binary and n-ary versions clear.

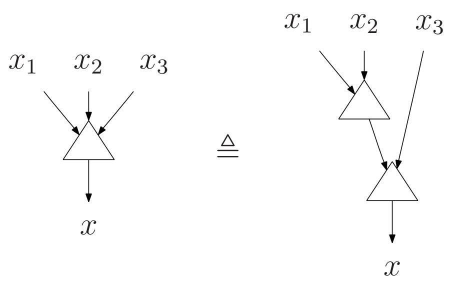


Figure 5.32: A ternary pair task defined in terms of a chain of binary pair tasks.

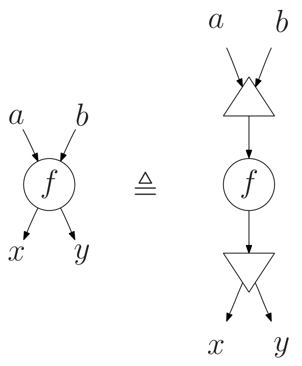


Figure 5.33: A processing task taking two inputs and two outputs can be expressed in terms of a pair task, a processing task and an unpair task.

Formally, the n-ary pair and unpair tasks, where $n > 2$, are defined inductively by the following rules, where Pair_2 and Unpair_2 are synonymous with Pair and Unpair, respectively.

$$\begin{aligned}\mathsf{Pair}_n(\star)(x_1,\ldots,x_n,x) &\triangleq \exists t.\ \mathsf{Pair}_{n-1}(\star)(x_1,\ldots,x_{n-1},t) \wedge \\ &\qquad \mathsf{Pair}(\star)(t,x_n,x), \\ \mathsf{Unpair}_n(s)(x,x_1,\ldots,x_n) &\triangleq \exists t.\ \mathsf{Unpair}_{n-1}(s)(t,x_1,\ldots,x_{n-1}) \wedge \\ &\qquad \mathsf{Unpair}(s)(x,t,x_n).\end{aligned}$$

Using pair and unpair tasks allows us the expressivity to describe processing tasks taking multiple inputs and giving multiple outputs. A processing task taking multiple inputs and giving multiple outputs can be thought of as a unary processing task sandwiched between pair and unpair tasks. The binary case is shown in Figure 5.33, where the function f is of type $\alpha \times \beta \to \gamma \times \delta$.

Thus we recognise that the processing task's single input or output can be a tuple whose type is the Cartesian product of several other types. For example, a task representing a function of type $\alpha \times \beta \to \gamma$ would not be considered as a function of two arguments but a function of a single value of type $\alpha \times \beta$, producing output values of type γ. Indeed, we have already been thinking in this way in the arithmetic mean example (see Section 5.5.7.1).

A benefit of this approach, rather than extending the definition of processing tasks, is that pairing and unpairing are explicit in the task graph so can be allocated to processors in the network. Also, it means that we can deal with all the elements of a tuple together throughout the task graph rather than treating them all independently until required by a processing task, reducing the complexity of the task graph and rendering it simpler to understand.

5.6.1.1 Example: standard deviation

It is common in a sensor network to compute the standard deviation of some sensor values as a measure of the magnitude of the spread of the values. The standard deviation, σ, can be computed using one of the following equivalent formulae:

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2} \tag{5.38}$$

$$= \sqrt{\left(\frac{1}{n}\sum_{i=1}^{n}{x_i}^2\right) - \left(\frac{1}{n}\sum_{i=1}^{n}x_i\right)^2}, \tag{5.39}$$

where n is the number of values and $\overline{x}$ is the mean. In (5.39), the mean is calculated explicitly.

In order to implement these two methods of computing standard deviation as task graphs in as parallelisable fashion as possible, we define the following processing functions:

$$\begin{aligned} f_1(x) &= (x, 1) \\ f_2(x, n) &= \frac{x}{n} \\ g_1(x) &= x^2 \\ g_2(x, y) &= x - y \\ g_3(x) &= \sqrt{x}. \end{aligned}$$

We also recall the merge function defined earlier:

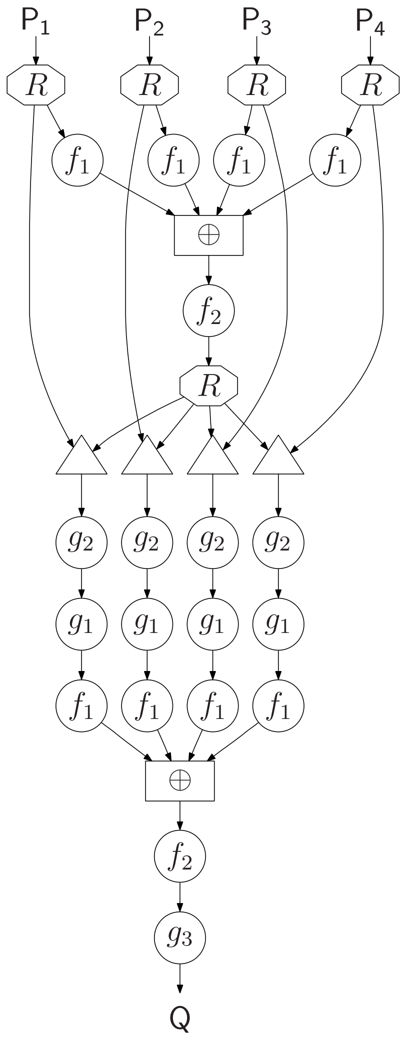
$$(x_1, n_1) \oplus (x_2, n_2) = (x_1 + x_2, n_1 + n_2).$$

Both formulae for computing σ require the use of pair tasks to construct the input to g_2 which subtracts two values.

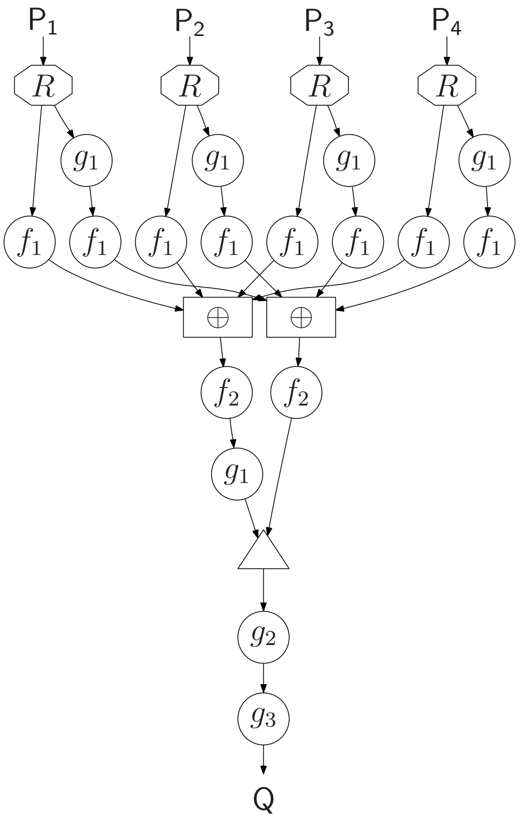
The task graphs for computing the standard deviation of four values, using both approaches, are shown in Figure 5.34. In approach (a), implementing the formula (5.38), the mean is calculated first and then subtracted from each input value. In approach (b), implementing the formula (5.39), the mean of the values is computed in parallel with the computation of the mean of the squares. This parallelisation suggests that approach (b) is likely to compute an output in a smaller time than approach (a), assuming an appropriately endowed resource graph.

5.6.1.2 Transformations involving pair and unpair tasks

Now that task graphs may contain extra kinds of task as well as those originally considered, the set of task graph transformations needs to be reconsidered. There are eleven new



(a) As in formula (5.38)



(b) As in formula (5.39)

Figure 5.34: Two task graphs to compute the standard deviation of four values.

transformations—three which involve both pair and unpair tasks; four which involve pair tasks with the other kinds of task; and four which involve unpair tasks with the other kinds of task. These are listed and proven to be sound in Appendix J (Sections J.1–J.3). A comment on the presence of redundancy in these trasformations is given in Section J.4.

5.6.1.3 Expressing pair and unpair in terms of other task kinds

Pair and unpair tasks appear to be useful in order to deal with tuples. However, adding these two extra primitives to the set of task kinds, along with the transformations shown above, adds complexity. In the spirit of keeping the set of primitives as small as possible, we consider whether pair and unpair tasks can be expressed in terms of the original, primitive task kinds.

We can use processing functions to convert values of type α and values of type β into values of type $\alpha \times \beta$. Where we have values from monoids $(\alpha, \star_\alpha, 0_\alpha)$ and $(\beta, \star_\beta, 0_\beta)$, we use processing functions $p_\alpha : \alpha \to \alpha \times \beta$ and $p_\beta : \beta \to \alpha \times \beta$ to return values from the monoid $(\alpha \times \beta, \star_{\alpha\times\beta}, (0_\alpha, 0_\beta))$, where the merge function $\star_{\alpha\times\beta}$ is defined as

$$(a_1, b_1) \star_{\alpha\times\beta} (a_2, b_2) \triangleq (a_1 \star_\alpha a_2, b_1 \star_\beta b_2). \tag{5.40}$$

If functions p_α and p_β are defined as

$$\begin{aligned} p_\alpha(a) &\triangleq (a, 0_\beta) \quad \text{and} && (5.41)\\ p_\beta(b) &\triangleq (0_\alpha, b), && (5.42) \end{aligned}$$

then values from α are paired with the identity element of β and values from β are paired with the identity element of α which allows a merge task on $\alpha \times \beta$ to yield the desired result.

Proposition 5.15. *A graph consisting of a merge task combining the values returned from $p_\alpha(a)$ and $p_\beta(b)$ acts like a pair task, outputting the pair (a, b).*

Proof. We examine the denotation of this graph:

$$\begin{aligned} \mathsf{G_{pair}}(x_1, x_2, y) &\triangleq \exists t_1, t_2.\ \mathsf{Proc}(p_\alpha)(x_1, t_1) \wedge \mathsf{Proc}(p_\beta)(x_2, t_2) \wedge \\ &\qquad\qquad \mathsf{Merge}(\star_{\alpha\times\beta})(t_1, t_2, y) \\ &\equiv \exists t_1, t_2.\ t_1 = p_\alpha(x_1) \wedge t_2 = p_\alpha(x_2) \wedge y = t_1 \star_{\alpha\times\beta} t_2 \\ &\equiv y = p_\alpha(x_1) \star_{\alpha\times\beta} p_\beta(x_2) \\ &\equiv y = (x_1, 0_\beta) \star_{\alpha\times\beta} (0_\alpha, x_2) \qquad \text{by (5.41) and (5.42)} \\ &\equiv y = (x_1 \star_\alpha 0_\alpha, 0_\beta \star_\beta x_2) \quad \text{by definition of } \star_{\alpha\times\beta},\ (5.40) \\ &\equiv y = (x_1, x_2) \qquad \text{by (5.12)} \\ &\equiv \mathsf{Pair}(x_1, x_2, y). \end{aligned}$$

□

Thus, if the datatypes from which a and b are drawn can be expressed as monoids, we can obtain (a, b) using two processing functions and a merge task. The definition of a pair task in these terms is shown in Figure 5.35.

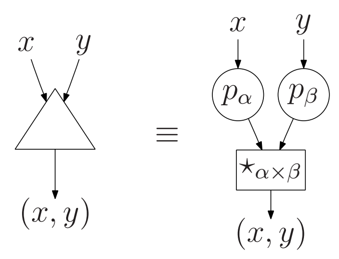


Figure 5.35: A pair task expressed in terms of two processing tasks and a merge task.

Even if a and b are not drawn from mergeable datatypes, we can still adopt this approach by wrapping the datatypes in monoids which each use a special sentinel value for an identity. Each monoid's merge function is then defined for the cases where the sentinel is present in either the first or second argument. Now, $p_\alpha(a) \triangleq (a, \sigma_\alpha)$ and $p_\beta(b) \triangleq (\sigma_\beta, b)$, where $\sigma_\alpha \notin \alpha$ and $\sigma_\beta \notin \beta$ are the sentinel values. Both functions return values from the monoid $((\alpha \cup \{\sigma_\alpha\}) \times (\beta \cup \{\sigma_\beta\}), \star_{\alpha\times\beta}, (\sigma_\alpha, \sigma_\beta))$ where now

$$a_1 \star_\alpha a_2 \triangleq \begin{cases} a_1 & \text{if } a_2 = \sigma_\alpha \\ a_2 & \text{if } a_1 = \sigma_\alpha \\ \perp & \text{otherwise,} \end{cases} \tag{5.43}$$

and $\star_\beta$ is defined similarly, with $\star_{\alpha\times\beta}$ defined in terms of these new functions. In practice, value $\perp$ is never returned from $\star_\alpha$ or $\star_\beta$ since $\star_{\alpha\times\beta}$ is only ever invoked on the return values from p_α and p_β which each guarantee precisely one sentinel value.

Given that an unpair task is the inverse of a pair task, we consider whether unpair tasks can likewise be expressed in terms of the inverses of p_α, p_β and $\star_{\alpha\times\beta}$. The split task we desire is defined as:

$$s_{\alpha\times\beta}(a, b) \triangleq ((a, 0_\beta), (0_\alpha, b)). \tag{5.44}$$

Proposition 5.16. *$s_{\alpha\times\beta}$ is a right-inverse of $\star_{\alpha\times\beta}$.*

Proof. From the definitions of $\star_{\alpha\times\beta}$ and $s_{\alpha\times\beta}$:

$$\begin{aligned} \star_{\alpha\times\beta}(s_{\alpha\times\beta}(a, b)) &= \star_{\alpha\times\beta}((a, 0_\beta), (0_\alpha, b)) && \text{by defn. of } s_{\alpha\times\beta}, (5.44) \\ &= (a \star_\alpha 0_\alpha, 0_\beta \star_\beta b) && \text{by defn. of } \star_{\alpha\times\beta}, (5.40) \\ &= (a, b) && \text{by } (5.12) \end{aligned}$$

□

Hence $s_{\alpha\times\beta}$ is a valid split task for the monoid $(\alpha \times \beta, \star_{\alpha\times\beta}, (0_\alpha, 0_\beta))$.

Once $s_{\alpha\times\beta}$ has split a pair into two components, it remains to select the appropriate elements from each. This can be done by two processing functions which simply discard the other elements:[8]

$$\pi_\alpha(a, b) \triangleq a \tag{5.45}$$
$$\pi_\beta(a, b) \triangleq b. \tag{5.46}$$

[8]The functions are named π_α and π_β by analogy with projection functions in the relational algebra database query language.

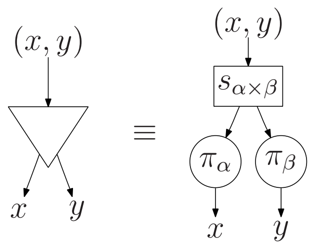


Figure 5.36: An unpair task expressed in terms of a split task and two processing tasks.

If these functions are used on the values returned from $s_{\alpha\times\beta}$, the discarded values will always be the identity values.

Proposition 5.17. *A graph consisting of processing functions π_α and π_β can be used in conjunction with split task $s_{\alpha\times\beta}$ to act like an unpair task, decomposing a pair (a, b) into its elements a and b.*

Proof. We examine the denotation of the graph when passed a pair (x_1, x_2):

$$\begin{aligned}
\mathsf{G}_{\mathsf{unpair}}((x_1, x_2), y_1, y_2) &\triangleq \exists t_1, t_2.\ \mathsf{Split}(s_{\alpha\times\beta})((x_1, x_2), t_1, t_2) \wedge \\
&\qquad\qquad \mathsf{Proc}(\pi_\alpha)(t_1, y_1) \wedge \mathsf{Proc}(\pi_\beta)(t_2, y_2) \\
&\equiv \exists t_1, t_2.\ (t_1, t_2) = s_{\alpha\times\beta}(x_1, x_2) \wedge \\
&\qquad\qquad y_1 = \pi_\alpha(t_1) \wedge y_2 = \pi_\beta(t_2) \\
&\equiv \exists t_1, t_2.\ (t_1, t_2) = ((x_1, 0_\beta), (0_\alpha, x_2)) \wedge \quad \text{by (5.44)} \\
&\qquad\qquad y_1 = \pi_\alpha(t_1) \wedge y_2 = \pi_\beta(t_2) \\
&\equiv y_1 = \pi_\alpha(x_1, 0_\beta) \wedge y_2 = \pi_\beta(0_\alpha, x_2) \\
&\equiv y_1 = x_1 \wedge y_2 = x_2 \qquad\qquad \text{by (5.45) and (5.46)} \\
&\equiv (y_1, y_2) = (x_1, x_2) \\
&\equiv \mathsf{Unpair}((x_1, x_2), y_1, y_2).
\end{aligned}$$

□

Therefore an unpair task can be expressed in terms of one split and two processing tasks, as shown in Figure 5.36. As before, if datatypes α or β are not mergeable, we can change the definition of $s_{\alpha\times\beta}$ to use sentinels rather than identity elements.

5.6.1.4 Revisiting the transformations involving pair and unpair

We have shown that pair and unpair tasks can be expressed in terms of the primitive kinds of task. This leads to the question of whether the transformations involving pair and unpair tasks defined above can also be derived purely from the transformations defined for the primitive tasks when pair and unpair are expressed in this way.

It transpires that five of the transformations cannot be expressed in this way. This is because they depend on the mathematical relationship between particular processing tasks mentioned in the transformations, or on π_α (used in expressing an unpair task) being an inverse of p_α (used in expressing a pair task). Full details are provided in Section J.5.

5.6.1.5 Compiler support of pair and unpair tasks

Some degree of language and compiler support for pair and unpair tasks is desirable. If they are not supported, in order to make use of a pair $\alpha \times \beta$, the programmer would have to code up the processing tasks (p_α and p_β) and merge task ($\star_{\alpha\times\beta}$) manually. This would have to be done for each such pair used in the program. This would be tedious given that these functions largely consist of boiler-plate definition, which means that they are simple to generate automatically, and it introduces the potential for mistakes to be made. It is also rather 'heavyweight' to use the monoid $(\alpha \times \beta, \star_{\alpha\times\beta}, (0_\alpha, 0_\beta))$ when its merge task is only ever used to combine values where precisely one element of the pair is an identity.

Because pair tasks can be expressed in terms of tasks of the primitive kinds, it is justifiable for a compiler to support pair and unpair tasks. This can be done either by internally expanding them in terms of the primitive kinds of task or by treating them as primitives themselves.

When pair and unpair tasks are treated as primitives, all the transformations involving pair and unpair tasks should be supported by the compiler. On the other hand, when pair or unpair tasks are treated as shorthand notation for the combination of processing and merge or split tasks, the transformations involving pair and unpair could still be directly supported in order to maximise the flexibility offered. If instead they are left unsupported and the original set of transformations fallen back upon then the five transformations which cannot be expressed solely in terms of the transformations on the primitives would not be available.

5.6.1.6 Generalising to n-ary pair and unpair tasks

The transformations involving pair and unpair tasks can also be generalised from binary to n-ary versions, so that a compiler can implement these directly rather than relying on repeated application of the binary versions of the transformations.[9]

5.6.2 Task graph edges

Now that we have shown that a function of arbitrary type can be represented in a task graph, we turn to the question of whether they can be connected in an arbitrary fashion in determining whether Proposition 5.14 holds.

We defined a task graph as being a directed acyclic graph. The absence of cycles means that there is no possibility for feedback loops or mutually recursive functions. Figure 5.37 shows two such illegal configurations. These are prohibited because cyclicity makes the task graph stateful, where future outputs may depend on the values of past outputs. This makes it hard to reason about large-scale properties of the graph and safely perform task graph transformations. Moreover, cyclicity gives rise to the possibility of non-termination due to deadlock, as a merge task waits forever for a value that depends on its output. A detailed treatment of deadlock in task graphs is given by Wadge [239].

[9]We give the denotations of the graphs involved in the transformations in Figure J.15 but omit the proofs of soundness.

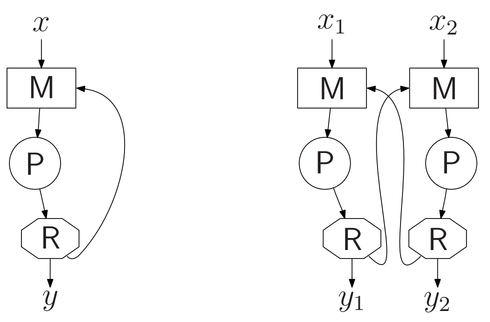


Figure 5.37: Two prohibited task graphs, which involve cycles.

Kahn's theory of dataflow graphs provides a fby operator to allow the present output to depend on the previous input and a next operator to allow the present output to depend on the next input [239, §1]. Conceivably, special fby and next tasks could be introduced to perform these functions in task graphs. However, these operations are particularly low-level. Instead tasks are allowed to store state which allows them to implement this functionality internally.

Therefore, if a program involves any kind of feedback, this must all be enclosed within a single task if it is to be represented as a task graph. However, this is not ideal, as the task would then not be possible to break down into pieces to be assigned to different processors in the network.

5.7 Related work

There has been much research that is related to the content of this chapter. Closely related are the numerous systems which perform automatic task assignment described in Section 2.2.3. Task graphs are related to the approach to modelling parallel systems of using dataflow graphs. This area is briefly described in Section D.3 in Appendix D.

In this section, we begin with a discussion of systems that employ transformations on task graphs, and then perform a comparison to two of the most closely related frameworks, MapReduce and SpatialViews.

5.7.1 Task graph transformations

A compiler offers an opportunity to optimise the execution of a program in some dimension after it has been analysed. In traditional programming languages, optimising compilers will typically apply program transformations to yield a faster executable, an executable that makes more judicious use of memory at run-time, or a smaller executable.

Analogous techniques have also been applied to compilers for graph-based programming paradigms. Usually, the aim is to increase parallelism inherent in the graph, which in turn results in faster execution. Many of the task graph transformations presented in Section 5.5.7 have this effect, although the decision about whether to employ a particular transformation is left to the compiler based on the programmer-supplied cost function, which may embody an optimisation policy with goals other than increases in parallelism.

The Dryad system described in Section 2.2.2.7 performs dynamic graph transformations to modify the shape of an application's task graph in response to real-time observations of the size of data [123, §5.2]. The transformations may have the effect of improving the efficiency of the application's execution.

Gordon et al. describe the application of transformations of stream processing systems (see Section D.3) to increase the degree of task parallelism, data parallelism and pipeline parallelism [90]. This is achieved through 'fission' of stream processing elements into several parallel elements and the 'fusion' of a pipeline of adjacent elements into larger elements. This is based on the StreamIt stream processing programming language, which is described in Section D.3.2.

The field of database query optimisation was briefly introduced in Section 2.3.1. As well as performing a variety of simplification operations, query optimisers may apply transformations to a query's execution plan in order to yield faster computation of results [45]. These transformations may include the elimination of redundant predicates, simplification of expressions, unnesting of sub-queries and re-ordering of operations [140, p426]. Such transformations are particularly important in distributed query processing, where the decision about the placement of execution is crucial since the transportation of data may be costly.

The PacLang language for programming network processors, described in Section F, offers the ability for programmers to express transformations to apply to the source code before the constituent program tasks are mapped to the available processors [66]. Ennals et al. describe a collection of transformations which affect the shape of the task graph which may make it better suited to execution in a given network architecture [67]. Whilst at present the transformations must be hand-picked, it is conceivable that this could be derived by an optimising compiler when given a description of the architecture.

5.7.2 MapReduce

The task graph formalism introduced in this chapter can be used to emulate the MapReduce framework, introduced in Section 2.2.2.7. The implementation of the example in Figure 2.6 as a task graph is shown in Figure 5.38.

A split task is used to divide the input dataset into several chunks. Processing tasks are used to encode the map and reduce functions. The partitioning is implemented by a processing task which takes a set of key–value pairs as input and has r outputs, where r is the number of reduce tasks, defined as:

$$\text{part}(S) \triangleq (\delta_1, \delta_2, \ldots, \delta_r) \quad \text{where } \delta_i = \begin{cases} v_i & \text{if } k_i : v_i \in S \\ 0 & \text{otherwise.} \end{cases}$$

In words, for each key–value pair $k : v$, the part function sends v to the reduce task responsible for key k and 0 to all the others.

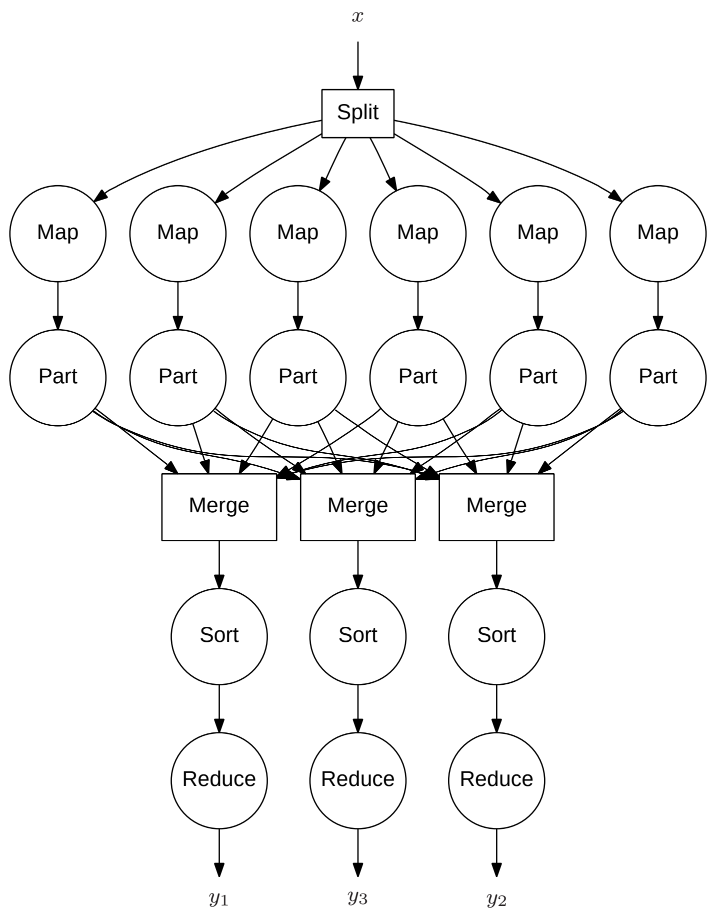


Figure 5.38: Emulating the MapReduce framework with a task graph. We show six map tasks and assume that there are three intermediate keys. The Split, Map and Reduce functions are provided by the programmer; Part, Merge and Sort are application-independent.

The merge function responsible for intermediate key k has m inputs, where m is the number of map tasks, and performs the multiset union of its inputs whilst ignoring zeros:

$$\text{merge}(v_1, v_2, \ldots, v_m) \triangleq \biguplus_{i=0}^{m} v_i \setminus \{0\}.$$

In this way, it gathers all of the values which were paired with intermediate key k into a multiset. This is then passed to the sorting functions, which are also implemented as processing tasks. For some programs, there is no requirement for the intermediate values to be sorted before being passed to the reduce function. In this case, the processing tasks implementing the sorting can be dropped from the task graph.

The dense edges in the task graph between the Part tasks and Merge tasks are necessary because it is not possible to determine statically which intermediate keys will be generated from each map task. However, in general, not all of the mr edges will have key–value pairs passed down them. Instead, many of the merge tasks will be passed zeros. An extension to the compiler could optimise this arrangement to not send zeros from Part functions since they are ignored by Merge.

It is also not possible to determine statically how many reduce tasks are necessary; we must instantiate one for every possible intermediate key. Again, this may give rise to significant complexity, causing the task assignment routine to assign to processors merge, sort and reduce tasks for keys which never arise at run-time.

5.7.2.1 Examples of emulating the MapReduce framework

We show how two examples described by Dean and Ghemawat [56] can be emulated using a task graph. In both cases, the map and reduce functions that are described for use with the MapReduce framework are used unchanged in processing tasks.

Count of URL access frequency. This application processes logs of web page requests and outputs the number of hits every URL has received. We assume that the input is a multiset of URLs.

The split task which divides this multiset into separate chunks to be independently processed is the inverse of the multiset union operator. For efficiency, this should return a balanced set of multisets, such that each multiset is of approximately similar cardinality. The processing function for the 'map' stage is a function m which pairs each URL in the multiset with the integer 1:

$$\begin{aligned} m(\varnothing) &= \varnothing \\ m(\{x\} \uplus xs) &= (x, 1) \uplus m(xs). \end{aligned}$$

The processing function for the 'reduce' stage is a function r which sums each value in the list passed to it by the merge stage for a particular URL:

$$\begin{aligned} r([]) &= 0 \\ r(x \mathbin{+\!\!+} xs) &= x + r(xs). \end{aligned}$$

Reverse web-link graph. This application processes web pages and returns, for each page, a list of the URLs of the pages which link to it. We assume that the input is a multiset of (URL, page contents) pairs.

As above, the split task is the inverse of the multiset union operator, and should return balanced outputs for efficiency. The processing function for the 'map' stage is a function which processes the web pages, extracting the targets of links and returning a set of (destination URL, source URL) pairs. The processing function for the 'reduce' stage is the identity function. No work needs to be done since the Merge stage will produce lists of source URLs for a given destination page URL.

In both examples there is no need for a sort stage.

5.7.3 SpatialViews

The SpatialViews framework [189], described in Section 2.3.2.1, shares the goal of enabling programmers to write architecture-independent programs. SpatialViews programs describe operations which are executed on processors which are not determined until runtime. The kinds of processors on which to execute are prescribed by the programmer via a type system. This facility is not available in the task graph paradigm; instead, the processors on which tasks are executed is determined by a cost function. The approach taken by SpatialViews is more suited to environments in which processors have different capabilities, such as where processors are usually individual sensors rather than general-purpose computing resources.

To exemplify the differences in the programming models, consider the program in Listing 5.1 which computes average light levels within a region.[10] The computation performed by this program is similar in spirit to that described by the task graph of Figure 5.5a.

Listing 5.1: SpatialViews program to compute average light levels

```
 1 public class AverageLighting {
 2   public static void main(String[] args) {
 3     sumreduction float s=0;
 4     sumreduction int n=0;
 5     spatialview sv=LightSensor @ SpaceDefs.CampusB % 320;
 6     visiteach x :  sv
 7       { s += x.read(); n++; }
 8     if (n>0)
 9       System.out.println(Float.toString(s/n));
10   }
11 }
```

The SpatialViews programmer defines a 'spatial view' embodying a set of light sensors (line 5) and an operation to perform on each processor in the set (line 7). Source tasks in the task graph correspond to the light sensors. The computation performed in parallel on each processor is akin to the processing task h. The merging operation, $\oplus$, encapsulates the summing of the numerators and denominators in the intermediate datatype. This is implicit in the declaration of the two variables in the SpatialViews program using the sumreduction keyword on lines 3 and 4. The final processing task, g, performs the division equivalent to that on line 9.

[10]This example has been reproduced from Figure 3 of [189].

With regards to parallelism, the SpatialViews language restricts access to the reduction variables to allow the code to be executed on each node in the spatial view concurrently. On the other hand, in the task graph paradigm, the parallelism between the processing tasks implementing h is explicit. Furthermore, in SpatialViews, parallelism can only arise from visiteach blocks. All other code is executed centrally, rather than employing automatic task assignment to execute it in the most appropriate place.

The particular processors on which the processing will take place is abstracted by the spatial view concept. To a certain extent, this is similarly abstracted in the late physical binding performed by a task graph compiler. In an initial task graph, only one instance of the h processing task is required because the Merge–Processing transformation can be performed to yield the appropriate number of h tasks. However, the correct number of source tasks are still needed and these must be bound to the appropriate physical light sensors. It is an area for future work to fully abstract this from a task graph, perhaps by using a type system like SpatialViews'.

Finally, SpatialViews supports only a limited range of reduction operations. The version described in [189] supports only sum and product, although the authors concede that any commutative and associative operation would suffice. This concept is generalised in the definition of a merge task which can embody any such operation.

5.8 Further work

There are various areas in which investigation is required to further this research.

Abstracting source and sink tasks. Although late physical binding using task graphs aims to allow programmers to make no assumptions about the architecture in which the application will execute, this is not fully achieved. The programmer must know in advance the appropriate number of source and sink tasks to employ. A means of abstracting this could be designed so that the task graph can be fully independent of the architecture on which it is executed. A technique analogous to the definition of spatial views in the SpatialViews framework could be adopted.

Quasi-static applications. In Section 5.4.1 we defined quasi-static applications to be those in which mobile nodes, whose movements satisfy particular statistical properties, can be grouped into regions. A protocol could be designed for implementing regions in quasi-static applications in which mobility is abstracted by the use of regions which are treated as a static node in the resource graph.

Local optimisation. The assumption of a global, omniscient controller which determines the assignment function and allocates work to the processors is rather naïve. A protocol could be designed to allow the optimisation of task assignment to be performed on a local scale, given a description of the optimisation policy.

Cyclicity. It may be considered whether the notion of task graphs could be extended with any degree of cyclicity permitted. Perhaps statefulness could be modelled in a way analogous to Sheeran's work with VLSI design language muFP [216], where functions can be augmented with a second output that is fed back as a second input in order to carry state.

Queueing of values. The present assumption that tasks' input queues are of unbounded length is unrealistic in practice. Moreover, if a merge task has two inputs, one of which receives data at twice the rate of the other, then repeatedly taking the value at the head of each queue is not optimal in many application scenarios because it would involve merging values of different ages together. Consideration would need to be given to how to deal with varying rates of input, perhaps evaluating whether it is sensible to execute binary merge functions with only a single value under certain circumstances, and whether it is ever sensible to discard old data from input queues.

Non-deterministic data flow. In some applications, it may be desirable that data flow is not deterministic but has some statistical properties. The merits of a new, 'stochastic' kind of task could be considered, which has one input and two outputs, whose input values are copied to one output with probability p or to the other output with probability $1 - p$.

5.9 Summary

Traditionally, when designing a distributed application, a programmer must manually define where the components of the application—called tasks—will be executed. The absence in the task definitions of a notion of where in the system they are to be executed allows a compiler to automatically derive an assignment of tasks to processors. Many areas of distributed computing can benefit from automatic task assignment, including sensor networks, ubiquitous computing, grid computing and web services.

The assignment of tasks to processors can be optimised by performing various transformations that change the task graph whilst preserving its semantics. A number of such transformations have been described. Certain transformations only preserve the semantics of the graph when the processing functions involved are monoid homomorphisms. Other transformations have further requirements. We have shown how more complex transformations can be built out of the original transformations.

A task graph is defined to be a directed graph consisting of source, sink, processing, replication, merge and split tasks. These tasks are sufficient since pair and unpair tasks can be expressed in terms of them. This means that a task graph is sufficient to express any algorithm; however, if it involves mutual recursion or feedback loops, the functions involved must all be contained within a single task.

CHAPTER 6

Language and Compiler

In Chapter 5, we introduced a task graph paradigm for designing distributed applications. This paradigm is particularly attractive because optimisations can be performed on the task graph originally specified by the programmer in order to increase its suitability for execution in a particular network.

In this chapter, those abstract ideas are made concrete in a programming language and compiler which we describe. The programming language, described in Section 6.1, enables programmers to express the computation performed by merge, split and processing tasks in a natural fashion similar to object-oriented programming, and describe the task graph which indicates how the tasks are connected. The compiler, described in Section 6.2, takes these definitions along with a description of the computational resources and performs transformations to optimise the assignment of tasks to processors. The compiler then outputs an executable for each processor.

Chapter 7 will present some examples of real-world applications implemented in the language introduced here.

6.1 Language

When introducing a new programming paradigm, there is a question over the most suitable means of exposing it to a programmer [189, §5.1]. One option is to create a library and API for an existing programming language. In this approach, the programmer can rapidly adjust to the new paradigm because there is already an existing, familiar tool-chain for him to use and no new language features to learn.

A second option is to embed the new features into an existing language. This is the approach taken by the .NET Language-Integrated Query (LINQ) framework [173], in which database queries are expressed using normal method calls and are compiled into SQL. This brings the advantage of allowing compile-time analyses to be performed that could not be performed by the host language's compiler, but it may be difficult to effectively embed features whose semantics depart dramatically from those of the host language.

A third option is to create a new programming language. This is a disruptive approach: programmers must learn new syntax and program structure as well as needing to gain an intuitive understanding of the function performed by the language's compiler. On the other hand, the new programming language can be precisely tailored to suit the paradigm, perhaps introducing new abstractions which do not exist in other languages. This is likely to make code more readable and hence more maintainable. Again, the creation of a dedicated compiler for the language means that compile-time analyses and program transformations are possible.

Since a major motivation for the task graph paradigm is the possibility to perform compile-time optimisations, we have chosen to create a new programming language to implement the paradigm of late physical binding through task assignment.

To implement an application in this paradigm, the following components must be provided to a compiler:

- a task graph, defining the computation performed by merge, split and processing tasks, and indicating the connections between tasks;
- a resource graph, defining the network of computational resources available to execute the tasks;
- an initial mapping, indicating the resources to which source and sink tasks are bound;
- a cost function, used to optimise the assignment of tasks to resources.

In cases where the network is dynamic or the program is designed to work on a number of network architectures, the resource graph cannot be specified by the programmer. Instead, the topology of the network must be discovered automatically and the resource graph created just prior to compilation.

6.1.1 Datatype definitions

There are various ways in which the task graph computational model could be encoded in a programming language. One approach is task-oriented, in which each processing, merge and split task is a first-class citizen. Instead, a datatype-oriented approach is adopted in which merge, split and processing tasks are encapsulated in the definitions of the datatypes they operate on. This approach ties in well with the modelling of some datatypes as commutative monoids, with their associated binary operation. For datatypes that can be modelled in this way, it is natural to encapsulate the underlying set, binary

```
mergeable splittable datatype PartialAv {
    private double numer;
    private int denom;

    public PartialAv() [cpu=0, out size=1] {          identity element, (0, 0)
        this(0, 0);
    }

    public PartialAv(double numer, int denom) {       singleton constructor
        this.numer = numer;                           for choosing a value
        this.denom = denom;                           from ℝ × ℕ
    }

    public PartialAv merge(PartialAv a)               merge function, ⊕
        [cpu=1, out size=sum]
    {
        return new PartialAv(this.numer + a.numer,
            this.denom + a.denom);
    }

    public PartialAv[] split()                        split function, ⋇
        [cpu=2, out size=in/2]
    {
        double seminumer = this.numer / 2.0;
        int semidenom = this.denom / 2;
        return new PartialAv[] {
            new PartialAv(seminumer, semidenom),
            new PartialAv(seminumer,
                semidenom + this.denom % 2)
        };
    }

    processto Average [cpu=1, out size=1] {           processing function, g,
        return new Average(this.numer / this.denom);  returning an instance of
    }                                                 the Average datatype,
}                                                     not defined here
```

Figure 6.1: Declaration of the datatype `PartialAv` representing the monoid $(\mathbb{R} \times \mathbb{N}, \oplus, (0, 0))$.

operation and identity element in a single logical unit. Processing tasks that can process data of a particular datatype are also encapsulated within that same logical unit.

This datatype-oriented approach fits in well with object-orientation. Hence we have chosen to base the language on Java, although any other object-oriented language would have provided an adequate basis.

Each datatype is defined in its own file and has a syntax built on top of a Java class. Metrics relating to tasks that are used by the cost function to evaluate a mapping are also specified in this file. A datatype is declared using the `datatype` keyword. To facilitate physical distribution of the datatype's components, static methods and static fields (except public static final fields) are not permitted in datatype declarations.

Datatypes must specify a constructor by which to create a instance of the datatype that wraps a single element from the underlying set of values. This allows new items of data to be instantiated.

6.1.1.1 Merge and split tasks

As introduced in Section 5.5.6.1, some datatypes are *mergeable*. These datatypes can be modelled mathematically as commutative monoids. They are defined in terms of an underlying set of values, an identity value and a merge function.

Declarations of such datatypes use the `mergeable` modifier to indicate that they are mergeable. Figure 6.1 shows the datatype declaration for the monoid $(\mathbb{R} \times \mathbb{N}, \oplus, (0,0))$ used in the arithmetic mean example in Section 5.5.7.1, which is a typical mergeable datatype. The use of the `mergeable` modifier entails two requirements:

- The datatype is a monoid so must have an identity element. This is implemented by requiring that mergeable datatypes support a constructor that takes no arguments.
- The binary merge operation must be specified. This is implemented by requiring that mergeable datatypes of type α support a publicly visible non-static method `merge` which takes an argument of type α and returns a value of type α. The merge function is specified such that the expression $a = a_1 \star a_2$ can be expressed in the fashion `a = a1.merge(a2)`.

Some datatypes define an operation to split them into a pair of smaller elements. Whilst it is necessarily true that all mergeable datatypes are also splittable in theory, it may be that the algorithm for implementing balanced splitting is significantly harder to implement than merging. For example, in the monoid $(\mathbb{N}, \times, 1)$, merging is multiplication (easy) but splitting is factorisation (hard). It is also conceivable that the converse is true for some datatypes: it may be much easier to express a balanced split operation than a merge operation. Therefore, it is not mandatory that mergeable datatypes necessarily support a split operation. The `splittable` modifier is used on datatypes that implement a `split` operation as a publicly visible non-static method which returns a pair of items.[1] Programmers of datatypes that are both mergeable and splittable need to ensure that the merge operation is the inverse of the split operation; in general it is undecidable for a compiler to check this statically, but it can be easily unit tested.

The example in Figure 6.1 defined a split task $\divideontimes$ which returns two smaller instances of the datatype of balanced magnitude, such that

$$\forall x \in \mathbb{N} \times \mathbb{R}.\ \oplus(\divideontimes(x)) = x.$$

6.1.1.2 Processing tasks

In the computational model, a processing task transforms data of one type into another type; see Section 5.5.6.2. Each datatype thus has zero or more other datatypes into which it can be processed. For each such possibility, the datatype declaration contains the code describing the processing task. These are defined in `processto` functions, which must each return an object of the target type.

In the Java-based implementation, this is implemented in the ‘source’ datatype’s declaration rather than as a constructor in the ‘destination’ datatype’s declaration so that private members of the source datatype can be accessed, but the alternative approach would have been equally valid.

[1]Returning a pair of items is implemented as an array of length two.

Note that the presence of a processing function in a datatype's declaration does not imply that it will necessarily be part of a task graph; it merely indicates that such a function exists.

Processing functions that are monoid homomorphisms are marked with the `homomorphism` keyword, to notify the compiler that transformations appropriate to homomorphisms can be safely applied in applications using this function. Again, it is undecidable for a compiler to check this statically, so it is the programmer's responsibility to ensure that this keyword is used on appropriate occasions. In Figure 6.1, the processing task representing g is not a homomorphism so is not marked in this way.

6.1.1.3 Cost annotations

Annotations describing the values of various metrics that are employed by the cost function are required for processing functions, merge functions, split functions and constructors that are used as source tasks. The metrics are specified as a comma-separated list of key–value pairs, enclosed in square brackets, where the key is a string known to the cost function and the value is a simple arithmetic expression. Several of these are evident in Figure 6.1.

Keys may include the `out` modifier to indicate that they are metrics characterising data on egress edges from the corresponding node in a task graph. Other values characterise the node itself. Egress edge values may use the special value `in` to refer to the value of the input for the corresponding key. For merge functions, which are unique in having more than one ingress edge, values may use the keywords `num`, `sum`, `max`, `min`, `avg` to refer to the number, sum, minimum, maximum or average of the input values for the corresponding key. For split functions, the keyword `outs` refers to the number of outputs.

For example, a merge task may be annotated with

```
cpu=50, out size=sum, out privacy=max
```

to indicate that its CPU load is fifty units; that the size of the output is the sum of the sizes of its inputs; and that the degree of sensitivity with respect to privacy is the largest such from among its inputs.

It is necessary for these annotations to be attached to the definitions of the functions, rather than the task graph, because the compiler is free to apply transformations to the task graph, and needs to know the values of the metrics on nodes and edges that it creates in the graph.

6.1.2 Task graph definition

Once the datatypes that are involved in the application have been defined, the programmer needs to indicate how the tasks are wired together, in a task graph definition file.

An example task graph definition for the arithmetic mean application is given in Figure 6.2. The task graph it defines corresponds to the merging-before-processing approach and can be thought of as a three-input version of the task graph depicted in Figure 5.5b.

```
taskgraph AvTemp {
    sourcetask<TempSet> c0 ["/dev/ttyS0"], c1 ["/dev/ttyS0"], c2 ["/dev/ttyS0"];
    mergetask<TempSet> m0 [1 => inf, 2 => inf, 3 => inf];
    processtask<TempSet, PartialAv> p0;
    processtask<PartialAv, Average> p1;
    sinktask<Average> s0;

    c0 -> m0; c1 -> m0; c2 -> m0;
    m0 -> p0;
    p0 -> p1;
    p1 -> s0;
}
```

Figure 6.2: Task graph definition for the temperature-averaging application.

The `taskgraph` block is used to supply a name for the application (`AvTemp` in the example) and specify an initial task graph for the application. This task graph will not necessarily reflect that which is eventually executed; the compiler is free to perform transformations to optimise it. Hence, programmers are encouraged to describe the initial task graph in as clear a fashion as possible.

Each task in the graph is declared to be either a `sourcetask`, a `sinktask`, a `mergetask`, a `splittask`, a `processtask` or a `reptask`. Where appropriate, the ingress and egress datatypes for these tasks are specified in angled brackets. Programmers specify the egress datatype for source tasks; the ingress datatype for sink tasks; the ingress and egress datatype for merge tasks, split tasks and replication tasks; and both the ingress and egress datatypes for processing tasks.

The links between tasks, indicating the direction of data flow, are specified using the identifiers given to the tasks. Tasks with multiple inputs or outputs can be specified by having several edges terminating at or leading from the task; this is the case for `m0` in the example, which is a ternary merge task.

Source tasks can optionally be given a list of arguments that are to be passed to the constructor of its datatype, if any are required. In a sensor network, the source tasks generate the application's input data, so the arguments can be used to create an instance of the source datatype appropriate to each source task. In the example, each source task is passed a string indicating the device file corresponding to the sensor from which temperature data is to be sampled.

Merge tasks specified in the task graph can be annotated with an array of timeouts. (See Section 5.5.4 for the motivation for this facility.) For an n-ary merge task, timeouts are specified for each number of potential inputs received from 1 to n. The timeout for k inputs indicates the longest duration of time the task should wait, after having received $k-1$ input values, for the next. If a timeout expires, the merge task treats that input as if it had received the datatype's identity element. The special timeout value `inf` denotes an infinite duration, implying that it is not acceptable for the merge task to produce an output without having received further input values. The timeout value 0 for a particular input implies that the merge task can always execute without a value being present on that input. If no timeouts are specified, it is assumed that the timeout for all numbers of inputs up to and including n are infinite. Infinite timeouts are generally to be avoided in systems such as sensor networks where the failure of nodes is frequent, as a single failure

```
resourcegraph {
    sensor0: 192.168.0.100 [speed => 2];
    sensor1: 192.168.0.101 [speed => 2];
    sensor2: 192.168.0.102 [speed => 2];
    host3:   192.168.0.103 [speed => 10];

    sensor0 -- host3   [bandwidth => 5, latency => 1];
    sensor1 -- sensor0 [bandwidth => 1, latency => 1];
    sensor2 -- host3   [bandwidth => 5, latency => 1];
}
```

Figure 6.3: Example resource graph.

```
mapping {
    c0 -> sensor0;
    c1 -> sensor1;
    c2 -> sensor2;
    s0 -> host3;
}
```

Figure 6.4: Example initial mapping file definition.

may prevent the system from producing output.[2]

6.1.3 Resource graph definition

A resource graph definition file describes the computational resources in the network (the nodes) and the communications links between them (the edges). An example is shown in Figure 6.3.

Resources are given names, and their IP addresses or hostnames are supplied. Whilst the current implementation of the compiler produces code for an IP network, this need not be the case: this information is merely used to provide a name for an endpoint for communication between tasks.

The links between nodes are specified as shown in the example. Symmetric bi-directional links are indicated by the `--` operator; uni-directional links are indicated by `->`. Asymmetric links can thus be specified using two uni-directional links.

Both the nodes and edges of the graph can be annotated with costs, specified as a comma-separated list of key–value pairs within square brackets. These costs denote properties of the processors or the communication links. Keys, such as `speed`, `bandwidth` and `latency` in the example, are names which are known to the cost function.

6.1.4 Initial mapping definition

An initial mapping file specifies an initial assignment function from tasks to processors. An example is shown in Figure 6.4.

The assignment function produced by a compiler is a superset of the mapping specified in the initial mapping file. In other words, the mappings specified here indicate fixed tasks that will definitely be executed on particular processors. Furthermore, tasks which are

[2]Despite this, infinite timeouts are used as the default because there is no more appropriate alternative.

bound to processors in the mapping are not eligible to be participants in a task graph transformation effected by the compiler.

The task names correspond to names specified in the task graph file; the resource names correspond to those specified in the resource graph file. In the example, the three source tasks are mapped to the three sensor nodes and the sink task is mapped to the other node.

Whilst there is no requirement to map any tasks to any processors, this facility will be used in many scenarios. For example, in a sensor network, a particular source task may need to be executed on a particular node because it has the sensor to be sampled, and a particular sink task may need to be executed on a particular node where the result of the processing is to be known. However, this feature can also be used by the programmer to lock other tasks to particular processors if desired, perhaps where special hardware support is required.

6.1.5 Cost function definition

The final element of an application provided to a compiler by the programmer is the cost function, which is used to evaluate a task assignment function.

In the present implementation, this is specified as a PERL script containing a function which computes the cost. This function is given a total assignment function, the task graph and the resource graph along with the attributes of their nodes and edges; it returns a positive real number indicating the efficacy of the assignment. A smaller return value denotes a more desirable assignment.

6.1.5.1 Built-in functions

A library of built-in functions from which a cost function can be simply constructed is provided with the compiler. An example of one such function which this library contains is an ‘execution time’ function. This function simulates the execution of the application and computes the elapsed time between the application’s source tasks producing a value and the result of processing arriving at all the sink tasks. The simulation of the application involves scheduling the tasks on the processors, respecting the parallelism inherent in the network and assuming that a single processor executes multiple tasks sequentially in the order of which task’s inputs are ready first.

The time to execute a task is calculated from the speed of the processor and the task’s CPU load. The processor’s speed is specified in the resource graph file as the `speed` attribute, measured in instructions per second. The task’s CPU load is specified in the datatype definition file as the `cpu` attribute, measured in instructions.

The communication time is calculated from the anticipated size of the data and the characteristics of the communication links. Source tasks’ `size` attributes specify the size of the data that they output in the datatype definition file for the source datatype; other tasks’ `size` attributes specify the relation between the size of their input and the size of their output. Hence the size of the data on any edge in the task graph can be estimated.

The characteristics of the communication links are specified in the resource graph file as `bandwidth` and `latency` attributes. Also, a boolean `parallelinputs` attribute can be used to specify whether a task with multiple inputs is capable of receiving data from predecessor tasks in parallel or whether this must be done sequentially.

6.1.5.2 Static nature of attributes

Of course, the attributes provided in the datatype definition and resource graph files are necessarily static values. This is because the cost function is executed at compile time and the attributes are specified by the programmer. Hence they can only be an estimate of the actual, dynamic characteristics of the application.

If the appropriate values of the attributes are not clear to the programmer, they could be estimated through off-line profiling. Alternatively, as is the case in many dynamic task assignment systems (see Section 2.2.3.2), dynamic values of attributes can be collected at run-time through on-line profiling. It is conceivable that a similar feedback mechanism could be implemented in order to update the attributes in accordance with true values experienced at run-time.

6.2 Compiler

We now turn our attention to describing the compiler, which applies late physical binding to applications and produces executables.

The inputs provided to the compiler are as follows:

- the datatype definition files;
- the task graph definition file;
- the resource graph definition file;
- the initial mapping definition file; and
- the cost function definition file.

The job of the compiler is to consider semantically-equivalent alternatives to the task graph supplied, and to derive a total task assignment function mapping each task to the processor defined in the resource graph found to be most appropriate to execute it. This is described in Section 6.2.1. Details regarding the compiler's implementation are provided in Section 6.2.2.

The output of the compiler is an executable for each processor that has at least one task assigned to it. In addition to the code written by the programmer, these executables will contain the code that performs inter-task communication. The compiler also provides a variety of facilities that the programmer can interact with to aid the development process, described in Section 6.2.3.

6.2.1 Task graph optimisation and assignment

The compiler analyses the task graph and the resource graph to determine the best locations to execute the tasks. As part of this process, the annotations associated with processors and communication links in the co-ordinator file are examined by the cost function to determine the relative suitability of any particular mapping of tasks to processors. The compiler applies the program transformations described in Section 5.5.7 as well as task assignment to determine the best task graph and best mapping of its tasks to processors.

Solving the task assignment problem is a long-established research area, sometimes referred to as *task scheduling*. However, the majority of this research has been done in the context of homogeneous networks; that is, where the processors have identical characteristics. We require an algorithm that optimises task assignment in a heterogeneous network.

We do not aim to make a direct contribution to this field—we have not attempted to make the task assignment algorithm implemented by the compiler state-of-the-art; improving it is an area for future work. However, to the best of our knowledge, we are not aware of the existence of a task assignment algorithm that considers performing transformations on the task graph to improve the assignment.

6.2.1.1 Task assignment algorithms

Kwok and Ahmad's extensive survey of static task assignment algorithms [144] describes 27 algorithms for scheduling directed task graphs on homogeneous multi-processor systems. However, the authors highlight that little work has been done in task assignment for heterogeneous systems, where the processors do not necessarily share similar characteristics [144, p455].

Algorithms for task assignment are NP-complete in all but a few restricted cases [72], meaning that it is usually infeasible to computationally determine the optimal assignment, even for the homogeneous case. To render the computation feasible, many polynomial-time sub-optimal heuristic-based approaches have been proposed, and attempts have been made to restrict the problem to simpler cases that can be solved optimally in polynomial time.

Casavant and Kuhl have produced a taxonomy of scheduling algorithms and classify various algorithms against it [42]. For static algorithms, they distinguish optimal and sub-optimal approaches. Sub-optimal approaches are classified as either approximate or heuristic-based.

Kafil and Ahmad present an algorithm for task assignment in heterogeneous environments, which finds optimal solutions [131]. The algorithm uses the A* best-first tree-search strategy. For task assignment, the nodes of the tree represent partial assignment functions, with the root being the totally undefined assignment function and the leaves being total assignment functions. A node's children are examined in order of a lower-bound estimate on the additional cost of assigning the currently unassigned tasks. Whilst this approach remains NP-complete, Kafil and Ahmad claim that the average-case complexity of their approach is usually acceptable 'for medium problems'.

Menascé et al. define a meta-algorithm that can be used to systematically build a range of static heuristic algorithms for heterogeneous environments [174]. This meta-algorithm consists of a loop that repeatedly executes a heuristic function, which selects a processor and a task from the set of unassigned tasks, until all tasks are assigned.

6.2.1.2 Implemented algorithm

Unfortunately, mapping a task graph to a resource graph is too general a problem to have a polynomial-time solution. Furthermore, the potentially computationally-intensive nature of the cost function means that an approach such as Kafil and Ahmad's A* search is not appropriate. Hence, a sub-optimal, heuristic-based algorithm has been implemented.

Implementing a suitable heuristic on which to base the search is challenging because of the possibility of performing task graph transformations. The search space can be thought of as a graph of multi-dimensional spaces. Performing a transformation causes a jump to another space; re-assigning a task from one processor to another causes a change within that space in the dimension corresponding to that task. A good search strategy will therefore have some means of predicting whether a transformation is likely to yield a space which contains a lower-cost assignment. Furthermore, the search must be careful to avoid looping if it were to consider a sequence of transformations whose composition is equivalent to the identity transformation.

The present solution involves starting in a random state. All unmapped tasks are initially assigned to random reachable nodes. For a given task, a reachable node is defined as one for which there exist paths from all nodes to which the task's predecessors are mapped. A method of steepest descent is then used to iteratively search for improvements, in two phases: firstly, all possible immediate task graph transformations are determined; secondly, for every program transformation we consider moving each task in turn to alternative processors in the resource graph. Finally the program transformation and task movement combination with the lowest cost is selected as the starting point for the next search iteration. We terminate our search when no further improvements can be found.

This procedure is repeated a number of times from different random starting states to aim to find a point in the search space with the global minimum cost. Each application of the method of steepest descent will find a local minimum, but the starting position may lead to a poor local minimum. The larger the number of iterations from random starting points, the more likely it is that the global minimum will be found.

Notably, the strategy for attempting transformations is sub-optimal. In particular, it will only perform a sequence of two transformations on a task graph if performing the first transformation results in an improvement. Hence, a small local maximum can obscure a valley of low cost in the direction beyond it. Improving the algorithm is the subject of future work.

6.2.2 Compiler implementation

In our current implementation, Polyglot [191] was used to define the language in which datatypes are defined. Polyglot is a framework that facilitates the creation of extensions to the Java language and compiler. This is achieved through a pre-processor that converts code written in the new language into standard Java code, performing new language-specific compile-time error checking. The standard Java code is then compiled using an ordinary Java compiler and executed using an ordinary Java virtual machine.

For example, the presence of the `mergeable` keyword on a datatype definition causes a check that a merge task with the appropriate signature is present. Code blocks introduced by the `processto` keyword are converted into ordinary methods.

As well as being used for the datatype definition language, Polyglot was also employed to compile the task graph, resource graph and initial mapping files. These are compiled into a single class that describes the graphs in terms of Java objects. When executed, this class performs the task graph optimisation and task assignment using the definition of the cost function, and generates the application's executable code in separate bundles, one for each processor. Each bundle is then compiled using a conventional Java compiler and wrapped up in a JAR file.

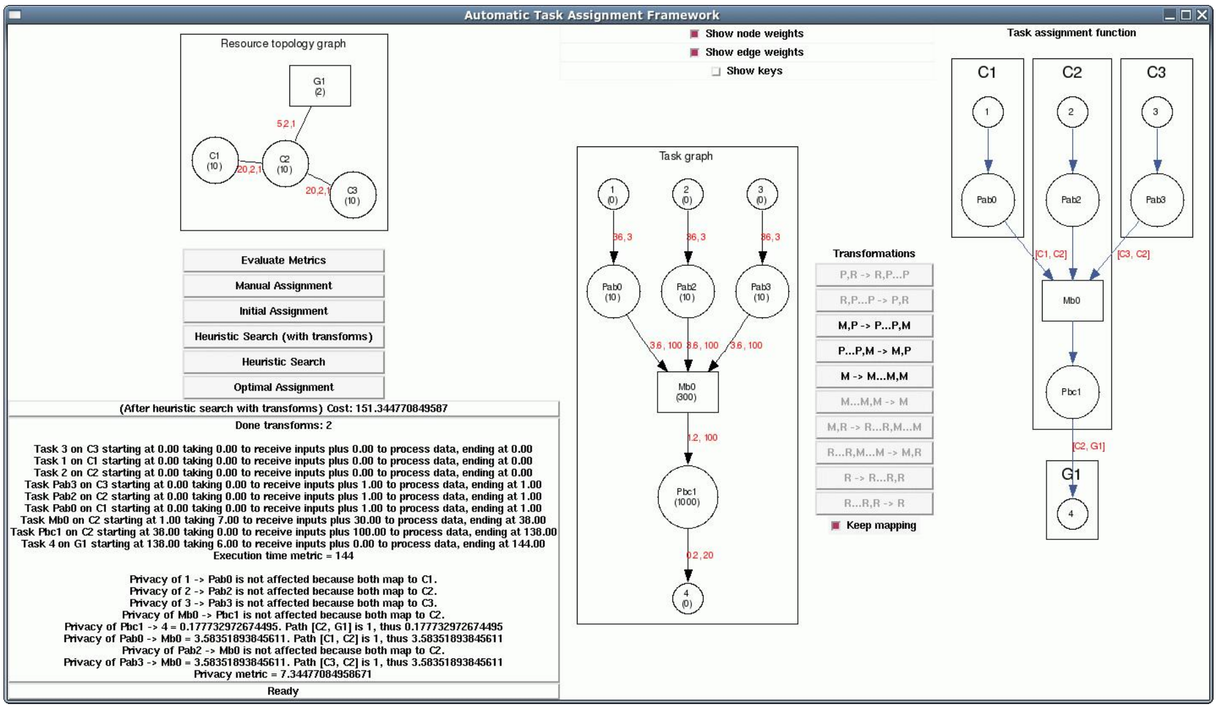
A compiler flag allows a choice between producing bundles to run on ordinary desktop computers or on resource-constrained Sun SPOT devices (see Section B.1). The former option generates a standard Java application that uses Java's built-in RMI for inter-task communications. For networks of resource-constrained devices, RMI is not ideal, since this approach requires a central registry to be present and accessible. The latter option generates a MIDlet for each processor and uses an alternative to RMI that was designed to run on devices with higher resource limitations, which has lower message overheads and does not require centralised registries.

6.2.3 Front end

The compiler also has a graphical user interface that the programmer can optionally interact with to aid the process of developing applications. Figure 6.5 shows a screenshot of this front end.

It provides the following facilities:

- Visualisation of the task graph, resource graph and assignment function, achieved using Graphviz [84] graph layout software.
- The ability to see which task graph transformations are eligible to be performed and perform them, to visualise their consequences.
- For a specified resource graph, the ability to execute the task graph optimisation and assignment algorithm used by the compiler, to preview the assignment.
- A detailed breakdown of the costs which add up to the overall cost of assignment.
- The ability to manually modify a task assignment, seeing the effect on the cost metric.



```
(After heuristic search with transforms) Cost: 151.344770849587
Done transforms: 2

Task 3 on C3 starting at 0.00 taking 0.00 to receive inputs plus 0.00 to process data, ending at 0.00
Task 1 on C1 starting at 0.00 taking 0.00 to receive inputs plus 0.00 to process data, ending at 0.00
Task 2 on C2 starting at 0.00 taking 0.00 to receive inputs plus 0.00 to process data, ending at 0.00
Task Pab3 on C3 starting at 0.00 taking 0.00 to receive inputs plus 1.00 to process data, ending at 1.00
Task Pab2 on C2 starting at 0.00 taking 0.00 to receive inputs plus 1.00 to process data, ending at 1.00
Task Pab0 on C1 starting at 0.00 taking 0.00 to receive inputs plus 1.00 to process data, ending at 1.00
Task Mb0 on C2 starting at 1.00 taking 7.00 to receive inputs plus 30.00 to process data, ending at 38.00
Task Pbc1 on C2 starting at 38.00 taking 0.00 to receive inputs plus 100.00 to process data, ending at 138.00
Task 4 on G1 starting at 138.00 taking 6.00 to receive inputs plus 0.00 to process data, ending at 144.00
Execution time metric = 144

Privacy of 1 -> Pab0 is not affected because both map to C1.
Privacy of 2 -> Pab2 is not affected because both map to C2.
Privacy of 3 -> Pab3 is not affected because both map to C3.
Privacy of Mb0 -> Pbc1 is not affected because both map to C2.
Privacy of Pbc1 -> 4 = 0.177732972674495. Path [C2, G1] is 1, thus 0.177732972674495
Privacy of Pab0 -> Mb0 = 3.58351893845611. Path [C1, C2] is 1, thus 3.58351893845611
Privacy of Pab2 -> Mb0 is not affected because both map to C2.
Privacy of Pab3 -> Mb0 = 3.58351893845611. Path [C3, C2] is 1, thus 3.58351893845611
Privacy metric = 7.34477084958671
Ready
```

Figure 6.5: Screenshot of graphical interface visualisation aid.

6.3 Further work

There is some scope for considering improvements to the language and compiler:

Dynamic task assignment. The compiler presently merely facilitates static task assignment. A run-time middleware could be implemented to consider dynamic modification of the assignment. As part of this, a means of on-line profiling for the application is required, which will dynamically update the cost attributes for the task graph's nodes and edges.

Macro language. A macro language to enable portions of a task graph (and corresponding entries in the initial mapping) to be generated automatically from a single schema would be a useful aid to development. This would permit large task graphs with a regular structure to be created more easily. One possibility would be to adopt an approach akin to Dryad, in which graphs are described in a traditional programming language extended with graph composition operators [123, §3].

Constraining the mapping. The initial mapping currently provides only the facility to bind a task to a particular processor, or to leave it entirely unbound and potentially suitable for execution on any processor. Some degree of middle ground may be desirable. For example, sets of processors in the resource graph could be identified and the initial mapping could allow the programmer to specify that a task can be assigned to any member of one such set. This facility could be useful to group processors that have specialised hardware that a particular task requires.

Dealing with failure. Presently, the code generated by the compiler assumes that the processors executing the tasks do not fail. A technique could be implemented to detect and recover from failure during run-time.

Implement pair and unpair tasks. The current implementation of the compiler does not support pair and unpair tasks natively; see Section 5.6.1 for a discussion of the desirability of this. Rather than explicit support for pair tasks, one option would be to support n-ary processing tasks. However, if the language were to support this, it would demand a change to the decision about where the code defining processing tasks is defined. Currently, they are defined within the datatype whose values they accept as input. But this would no longer make sense if the task takes multiple input values.

Improve task assignment algorithm. The present version of the task graph optimisation and assignment algorithm used by the compiler is somewhat naïve. It could be improved by implementing a more efficient task assignment technique and by investigating whether it is possible to heuristically predict a sequence of task graph transformations that might yield a better assignment.

6.4 Summary

A programming language has been created which can be used to write applications for distributed systems based on the task graph design paradigm introduced in Chapter 5. The language allows programmers to define merge, split and processing tasks encapsulated within the datatypes on which they operate.

The language's compiler takes the datatype definitions, along with descriptions of the task graph, the resource graph and the cost function against which an assignment is evaluated. The compiler performs automatic task assignment and the transformations described in Section 5.5.7 in order to find a strategy for executing the application. The compiler produces a JAR file to execute on each processor to which tasks are assigned, suitable for either Sun SPOT devices or ordinary desktop machines within JVMs.

Chapter 7 will describe the use of this language in practical settings.

CHAPTER 7

Examples

In this chapter, we present two examples of designing practical applications using the task graph paradigm introduced in Chapter 5 and of using the language and compiler described in Chapter 6 to implement them.

The example in Section 7.1 is a ray-tracing application exemplifying a typical grid computing application. This shows that the task graph paradigm can be readily applied to areas of distributed computing beyond just those involving computation involving vehicles. The example in Section 7.2 relates specifically to computing in vehicles and is an implementation of the map generation application of Chapter 4.

7.1 Ray tracing

Ray tracing is a simple technique in computer graphics to render an image of a scene as viewed from a particular location [88]. It involves projecting a straight line (a 'ray') from the viewpoint through each pixel of a screen until it hits an object in the scene. The colour of the object is the colour for that pixel, calculated by computing the angle to the light sources, taking into account the properties of the object's surface.

This procedure is readily amenable to parallelisation[1] as the colour for each pixel can be computed independently. The principle is to have a set of pixels, split it, process each into a partial image and recombine.

[1]We avoid the term *distributed ray tracing* since this usually refers to the technique of altering the directions of the rays to produce effects like motion blur, rather than to the parallelisation of ray tracing.

7.1.1 Datatypes

The application employs three datatypes:

- a set of co-ordinates of pixels to render (the PixelSet datatype);
- a rendered image, consisting of an array of pixel colours (the Image datatype); and
- a datatype to save the image to a file (the Writer datatype).

We will look at the source code for each in turn.

7.1.1.1 Set of co-ordinates

The PixelSet datatype represents a set of co-ordinates of pixels whose colours should be determined. Conceptually, the PixelSet datatype corresponds to the monoid

$$(\mathcal{P}(\mathbb{N} \times \mathbb{N}), \cup, \varnothing),$$

but we also encapsulate the width and height of the desired image for convenience. The source code for the PixelSet datatype is shown in Listing 7.1.

Listing 7.1: Datatype PixelSet

```
package raytracer;

import java.util.*;
import raytracer.*;

mergeable splittable datatype PixelSet {

   private Scene scene = new MyScene(); // hard-coded definition of scene

   private int w; // pixels
   private int h; // pixels
   private Collection set;

   public PixelSet() [cpu=0, out size=3] {
      this(0, 0);
   }

   public PixelSet(int w, int h) [cpu=1, out size=3] {
      this.set = initCollection();
      this.w = w;
      this.h = h;

      for (int i=0; i<w; i++) {
         for (int j=0; j<h; j++) {
            set.add(new Coord(i, j));
         }
      }
   }

   public PixelSet(Collection set, int w, int h) [cpu=1, out size=3] {
      this.set = set;
      this.w = w;
      this.h = h;
   }

   public String toString() {
      return "PixelSet["+set.size()+", w="+w+", h="+h+"]";
   }

   public Collection initCollection() {
```

```
41       return new ArrayList(); // or could be HashSet
42    }
43
44    public Iterator iterator() {
45       return this.set.iterator();
46    }
47
48     public int numPixels() {
49       return this.set.size();
50     }
51
52    public PixelSet merge(PixelSet that) [cpu=1, out size=max] {
53       Collection newSet = initCollection();
54       newSet.addAll(this.set);
55       newSet.addAll(that.set);
56       return new PixelSet(newSet, this.w, this.h);
57    }
58
59    public PixelSet[] split() [cpu=1, out size=in/outs] {
60       int elements = this.set.size();
61
62       Collection one = initCollection();
63       Collection two = initCollection();
64
65       Iterator iterator = this.set.iterator();
66       for (int i=0; i<elements/2; ++i) {
67          one.add(iterator.next());
68       }
69       for (int i=elements/2; i<elements; ++i) {
70          two.add(iterator.next());
71       }
72
73       return new PixelSet[] {
74          new PixelSet(one, this.w, this.h),
75          new PixelSet(two, this.w, this.h)
76       };
77    }
78
79    homomorphism processto Image [cpu=in*100, out size=in/3] {
80       Image image = new Image(this.w, this.h);
81       Viewer viewer = new Viewer(image, true, this.w, this.h);
82
83       int p = 0;
84       Iterator iterator = iterator();
85       while (iterator.hasNext()) {
86          Coord c = (Coord) iterator.next();
87
88          scene.doPixel(image, c.x, c.y);
89
90          // Update the viewer
91          viewer.paintPixel(c.x, c.y);
92          if (++p % 10 == 0) viewer.repaint(); // repaint every so often
93       }
94
95       return image;
96    }
97
98 }
```

Three constructors are defined: a default constructor that initialises an empty set of co-ordinates (lines 14–16); a constructor that produces a set of co-ordinates defining a rectangular area of specified width and height (lines 18–28); and a constructor that can be given an existing set of co-ordinates (lines 30–34).

The datatype is declared to be *mergeable* and *splittable* (line 6). This entails the presence of a *merge* method and a *split* method. The *merge* method returns the union of two sets of co-ordinates (lines 52–57). The *split* method performs a balanced split, partitioning the co-ordinates into two sets of (near-)equal cardinality (lines 59–77).

A processing task that converts an instance of the PixelSet datatype to an instance of the Image datatype is defined (lines 79–96). It is declared to be a homomorphism between these mergeable datatypes (line 79), which means that the programmer is satisfied that merging the results of running this processing task on two sets of co-ordinates achieves the same effect as running it on the union of these sets.

This processing task is where the core ray tracing algorithm is invoked. A fresh instance of the Image datatype of the appropriate height and width is instantiated (line 80). Then, for each co-ordinate in the set, the colour of the pixel is determined and written into the image (line 88). In order to visualise the progress of the rendering, a window showing the image is updated (lines 90–92).

The details of the ray-tracing algorithm by which pixel colours are determined are not shown. In the current implementation, the scene is defined in the MyScene class (not shown) through an object model. This is referenced on line 8. In a future implementation, it would be better for the scene to be passed as a parameter to the rendering processing task.

The cost annotations indicate that a source task producing a PixelSet would require negligible CPU cycles and produce output of size three units (line 14). Merging two instances of PixelSet produces an instance that is the same size as the largest of the two (line 52). Splitting a PixelSet into n parts produces outputs that are $\frac{1}{n}$ of the size of the original (line 59). Processing to an Image costs 100 units of CPU cycles for every pixel and produces an output that consumes a third of the memory (line 79).

7.1.1.2 Image

The Image datatype represents an array of pixel colours. It can be thought of as a monoid consisting of a partial function from pixel co-ordinates to their colours,

$$(\mathbb{N} \times \mathbb{N} \rightharpoonup \mathbb{C}, \rhd, \varnothing),$$

where $\mathbb{C} \triangleq [0,1]^3$ represents the red, blue and green components of a colour. The merge function, $\rhd$, behaves like its first argument where defined and otherwise like its second argument, i.e.

$$(f \rhd g)(x) \triangleq \begin{cases} f(x) & \text{if } x \in \text{dom}(f) \\ g(x) & \text{otherwise.} \end{cases}$$

Although in general $\rhd$ is not commutative, it is only ever used when $\text{dom}(f) \cap \text{dom}(g) = \varnothing$, so will always behave in a commutative fashion. The intent of this datatype is that a finished image will be a total function from the set of the co-ordinates of all pixels in a rectangle to their colours. Even in circumstances where there is an overlap between the domains of f and g, $(f \rhd g)(x)$ would still be suitable to use provided that $f(x) = g(x)$; this would be useful in implementations where re-computation is used to mask failures.

The source code for the Image datatype is shown in Listing 7.2. The partial function underlying the datatype is encoded as a two-dimensional array of type Colour (line 12), a class whose definition is not shown here.

Listing 7.2: Datatype Image

```
1 package raytracer;
2
3 import java.awt.*;
4 import java.awt.image.*;
```

```
import java.io.*;
import com.sun.image.codec.jpeg.*;

mergeable datatype Image {

   private int w; // pixels
   private int h; // pixels
   private Colour[][] pixels;

   public Image() [cpu=0] {
      this(0, 0);
   }

   public Image(int horizPixels, int vertPixels) [cpu=0] {
      this.w = horizPixels;
      this.h = vertPixels;
      this.pixels = new Colour[w][h];
   }

   public String toString() {
      return "Image["+w+"x"+h+"]";
   }

   public Colour getPixel(int i, int j) {
      return pixels[i][j];
   }

   public void setPixel(int x, int y, Colour col) {
      pixels[x][y] = col;
   }

   public Image merge(Image that) [cpu=1, out size=4] {
      int w = (int)Math.max(this.w, that.w);
      int h = (int)Math.max(this.h, that.h);
      Image m = new Image(w, h);

      for (int i=0; i<w; ++i) {
         for (int j=0; j<h; ++j) {
            Colour a = (i < this.pixels.length && j < this.pixels[i].length)
               ? this.pixels[i][j] : null;
            Colour b = (i < that.pixels.length && j < that.pixels[i].length)
               ? that.pixels[i][j] : null;

            m.pixels[i][j] = (a == null) ? b : a;
         }
      }

      return m;
   }

   processto Writer [cpu=0, out size=0] {
      return new Writer(this);
   }

   public void saveToFile(String filename) throws IOException {
      final int SCALE = Viewer.SCALE;

      BufferedImage image = new BufferedImage(w*SCALE, h*SCALE,
         BufferedImage.TYPE_INT_RGB);
      Graphics g = image.getGraphics();

      for (int i=0; i<w; ++i) {
         for (int j=0; j<h; ++j) {
            Colour c = pixels[i][j];
            if (c == null) {
               g.setColor(Color.WHITE); // default to white if colour is unknown
            } else {
               g.setColor(c.toColor());
            }
            g.fillRect(i*SCALE, j*SCALE, SCALE, SCALE);
         }
      }

      // Encode as a JPEG
      FileOutputStream outStream = new FileOutputStream(filename);
      JPEGImageEncoder jpeg = JPEGCodec.createJPEGEncoder(outStream);
      jpeg.encode(image);
      outStream.close();
83       System.out.println("Saved image to "+ filename);
84    }
85 }
```

The datatype is declared to be *mergeable* (line 8); the merge function implements ▷ (lines 36–53). Likewise, the datatype could also be declared to be *splittable*. However, the programmer has chosen to avoid providing a *split* function as this is not a useful operation to perform given the nature of the ray-tracing application.

A processing task to convert an instance of the Image datatype into an instance of the Writer datatype is provided (lines 55–57). The constructor of Writer will call the image's *saveToFile* method, which outputs the image as a JPEG file (lines 59–84).

7.1.1.3 File writer

The Writer datatype is a non-mergeable datatype used to write an image to a file. Its source code is shown in Listing 7.3.

Listing 7.3: Datatype Writer

```
package raytracer;

datatype Writer {

   private Image image;

   public Writer(Image image) [cpu=0] {
      this.image = image;
      try {
         image.saveToFile("/home/jjd27/scene.jpg");
      } catch (java.io.IOException e) {
         e.printStackTrace();
      }
   }

   public String toString() {
      return image.toString();
   }
}
```

7.1.2 Initial task graph

In much the same way as the example of computing π given in Section 5.5.10, the initial task graph is very simple and contains no explicit parallelism. The source file is shown in Listing 7.4.

The graph consists of a linear chain of four tasks (lines 11–13), starting with a source task which has explicit width and height parameters that are passed to the constructor of PixelSet (line 6). The graph is depicted in Figure 7.1.

Listing 7.4: Task graph definition

```
1 import raytracer.Image;
2 import raytracer.PixelSet;
3 import raytracer.Writer;
4
5 taskgraph {
6    sourcetask<PixelSet> c [600, 400];
7    processtask<PixelSet, Image> p;
8    processtask<Image, Writer> q;
```

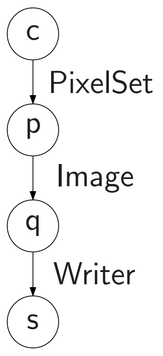


Figure 7.1: Initial task graph for ray-tracing application, with edges annotated with the types of data which travel along them.

```
  sinktask<Writer> s;

  c -> p;
  p -> q;
  q -> s;
}
```

7.1.3 Execution

We present an example scenario in which there are three computers. One is a ‘co-ordinator’ which generates the application’s input and on which the image file is to be written. The other two are fast processors which can efficiently tackle the ray tracing algorithm.

The description of the resource graph for this scenario is shown in Listing 7.5. The resource graph names the co-ordinator processor *coord* and the other processors *proc1* and *proc2*. The other processors are ten times as fast as *coord*. The link between *proc1* and *proc2* is slow and has low bandwidth, whereas each of these processors has a fast link back to *coord*.

Listing 7.5: Resource graph definition

```
resourcegraph {
  coord:  "devon.lce.cl.cam.ac.uk"[speed => 10];
  proc1:  "dtg-proc1.cl.cam.ac.uk"[speed => 100];
  proc2:  "dtg-proc2.cl.cam.ac.uk"[speed => 100];

  coord -- proc1 [bandwidth => 100, latency => 0.1];
  coord -- proc2 [bandwidth => 100, latency => 0.1];
  proc1 -- proc2 [bandwidth => 10, latency => 1];
}
```

The initial mapping is shown in Listing 7.6. In order that the algorithm’s output file be saved on *coord*, tasks `q` and `s` are tied to that processor. The input to the algorithm is assumed to arise on *coord*, so `c` is mapped similarly.

Listing 7.6: Initial mapping definition

```
mapping {
  c -> coord;
  q -> coord;
  s -> coord;
}
```

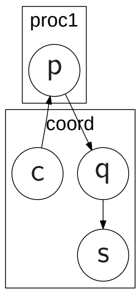


Figure 7.2: Best assignment function for the untransformed task graph.

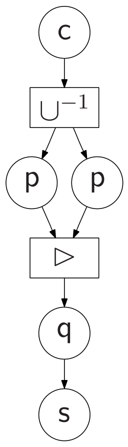


Figure 7.3: Task graph after the application of the Farm transformation.

When the task graph, resource graph and initial mapping files are fed to the compiler, it attempts to optimise the task graph. We elect to use the built-in cost function that seeks to minimise the algorithm's total execution time, described in Section 6.1.5.1. This is composed of the total time spent communicating and computing.

The best assignment function that the compiler can find for the untransformed task graph is shown in Figure 7.2. Task p is assigned to processor *proc1*. The cost function gives this assignment a score of 3.24[2].

However, this can be improved upon. The compiler performs the Farm transformation on task p, yielding the task graph in Figure 7.3, and discovers a better assignment, depicted in Figure 7.4. The assignment of this task graph scores 2.025. This assignment is better because the ray tracing is partitioned into two tasks, performed in parallel by the two fast processors. Even though the co-ordinator is slower, the new split and merge tasks are executed on it to avoid the expensive communication between *proc1* and *proc2*.

The images displayed in the windows showing the rendering progress at the end of the execution of task p on *proc1* and *proc2* are shown in Figure 7.5. The final output, encoded as a JPEG file, is shown in Figure 7.6. These images depict the scene that was hard-coded into the MyScene class. It consists of two lights, one white and one red; a red horizontal

[2]In general, a cost function uses arbitrary units, but since the only metric employed here is the execution time, the actual units of the cost are seconds.

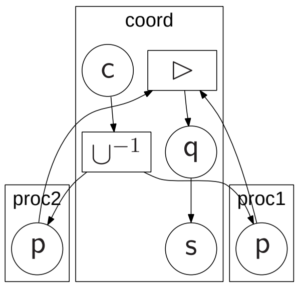
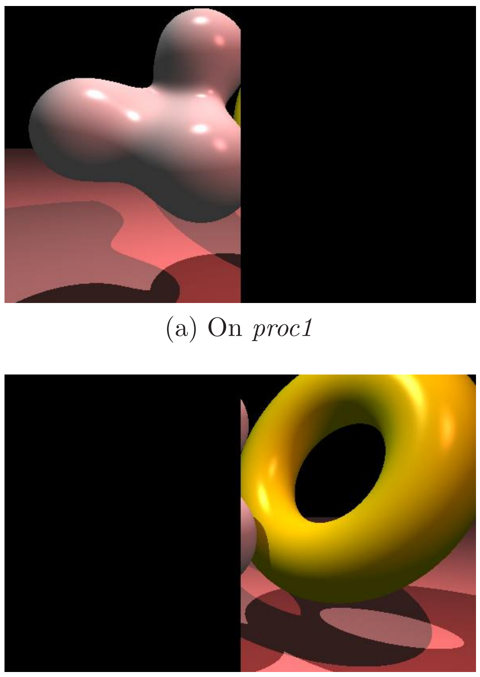


Figure 7.4: Best assignment function for the transformed task graph.


(a) On *proc1*

(b) On *proc2*

Figure 7.5: Images displayed in the viewer windows.

plane; a yellow torus; and a white implicit surface defined in terms of a function of three points.

7.2 Automatic road map generation

In Chapter 4, we proposed an algorithm for inferring a directed graph of the road network from position data collected from vehicles. This application provided the motivation for the program design paradigm of late physical binding as there is a wide range of possibilities for where the different stages of the algorithm could be executed; this decision affects the application's performance.

As noted in Section 4.3.1, the algorithm is naturally parallelisable by partitioning the data into geographic regions. This property can be exploited at a variety of the stages of the

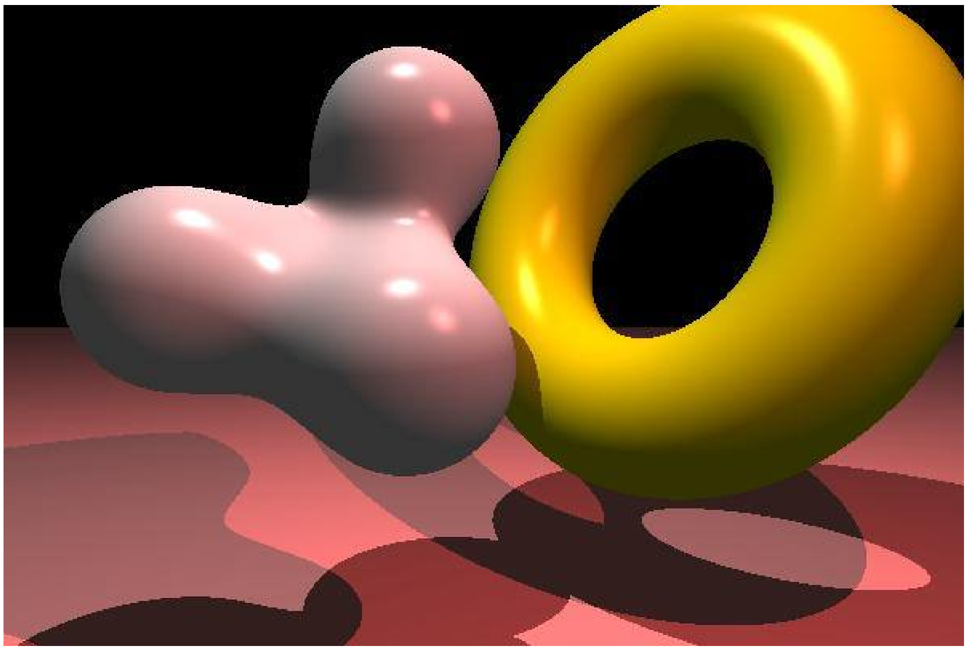


Figure 7.6: Final ray-traced image.

algorithm. For example, maps for different geographic regions could be produced independently by parallel pipelines which converge at the last stage to form a complete map. Alternatively, the vehicle location trace data from multiple vehicles could be combined and processed in a single pipeline.

We will discuss how the application has been designed and implemented in Sections 7.2.1 and 7.2.2. In Section 7.2.3, we will evaluate the suitability of automatic task assignment for the algorithm, examining the effect of program transformations and of execution in different environments.

7.2.1 Design

The core algorithm was described in Section 4.1.2. The algorithm was described in terms of a number of stages of processing; these correspond naturally to processing tasks between datatypes. The datatypes are:

SetOfJourneys a set of vehicle location traces, which are ordered sequences of positions;

VectorMap a cellular grid that stores the direction of travel of vehicles moving through each cell;

Histogram a two-dimensional histogram indicating the likelihood of the existence of road in each cell;

BlurredHistogram a blurred histogram, in which the data has been spread into neighbouring cells in an attempt to combat errors in the vehicle trace data;

Bitmap a thresholded histogram, which can be thought of as a bitmap, with a boolean value for each cell indicating whether it is believed to contain road;

RoadEdges the edges of the roads, produced by a contour follower;

HairyCentrelines a Voronoi graph of the area within the edges of the roads;

Centrelines an undirected graph of the centrelines of the roads, produced by removing the artefacts from the Voronoi graph; and

DirectedGraph a directed graph of the centrelines of the roads.

In order to provide the flexibility afforded by partitioning the data, each of the application's datatypes must represent data associated with a geographic region. Although any tesselation of regions could be employed, we will use rectangular tiles since they are simplest to implement.

As noted in Chapter 5, it is desirable for as many datatypes as possible to be mergeable, to permit a wide range of potential task graph transformations. Since instances of all the datatypes are concerned with geographic regions, it seems natural that the merge operations should combine regions. Whilst two instances of a datatype that concern neighbouring rectangular regions could conceivably be merged into a single rectangular region, the merge operation must be general enough to combine data in non-adjacent regions. This is implemented by instances of each datatype storing a set of rectangular tiles for which they hold data. The application's output is a datatype in which this set of tiles fills the plane between specified bounds with no overlaps or gaps.

7.2.1.1 Overlaps

It was also noted in Section 4.3.1 that the output from executing the algorithm on data from one region cannot simply be juxtaposed with the results from a neighbouring region. If this were done, a single, continuous road that spans both regions might not be continuous in the map at the boundary between the tiles. This is because the algorithm introduces errors near the edges of tile at several stages of processing, particularly in the production of the Voronoi graph and when its hairy artefacts are removed. The solution to this problem was to process not only the data within a region in isolation but along with the data from just over the boundaries with its eight surrounding regions. This is implemented by each datatype storing not only the rectangle describing the inner region for which successive stages of the algorithm will produce data that can be safely juxtaposed, but also the rectangle describing the outer region for which data is to be processed. Since this functionality is needed by all the datatypes, it is implemented in a common super-class that all of the datatypes inherit from.

The need for overlapping regions means that the partitioning of data will not result in a speed-up equal to the number of partitions. The speed-up of a parallel application is defined as the ratio of the time to execute in a serial fashion (on a single processor) to the time to execute in a parallel fashion (on n processors). If the partitions could be kept entirely independent, then doubling the number of processors would halve the execution time, so the speed-up would be n (ignoring the cost of communication).

Overlaps between tiles also mean that the algorithm's processing tasks are not true monoid homomorphisms. In other words, processing followed by merging will not necessarily produce identical output to merging followed by processing. However, within the inner bounds of each tile, the results are independent of the order of processing and merging. Since the application only retains the data within the bounds of the tiles, it is therefore safe and appropriate to indicate that the processing tasks are to be treated as homomorphisms by the compiler. This enables a wider range of task graph transformations than would be available if the processing tasks were not treated as homomorphisms.

7.2.2 Implementation

Since the present implementation of the compiler does not support pair tasks, processing tasks that take $n > 2$ inputs must be expressed in terms of n processing tasks $p_{\tau_1}, \ldots, p_{\tau_n}$ and a merge task $\star_{\tau_1 \times \ldots \times \tau_n}$ followed by the n-ary processing task, as proposed in Section 5.6.1.3. There is one n-ary processing task in the map generation algorithm: the task that processes the undirected graph into a directed graph. This task requires three inputs: the grid of vehicles' directions of movement; the road edges and the undirected graph of road centrelines. Hence, in addition to the datatypes identified above, a pseudo-datatype Triple is implemented to model tuples of type VectorMap×RoadEdges×Centrelines.

7.2.2.1 The datatype for sets of journeys

The SetOfJourneys datatype contains sets of ordered sequences of vehicles' positions, each sequence associated with the tiles within which it originated. The datatype's merge operation finds the union of the sets, maintaining the associations to the corresponding tiles.

A processing task is defined that transforms an instance of this datatype into an instance of the Histogram datatype. This procedure is potentially problematic since the instance may contain journeys spanning more than one adjacent tile and thus contain some subsequences of journeys more than once due to appearing in the overlap areas. If this situation were ignored, incrementing the values in the cells through which the sequences pass would lead to a histogram in which the values in the overlap areas are computed to be a multiple of their correct value. Hence, the processing task determines which cells of the histogram are covered by which tiles and, for each cell, divides the value by the number of tiles which cover it.

A further processing task is defined to transform a SetOfJourneys to a VectorMap. Structurally, the VectorMap is similar to a Histogram, so similar precautions are taken to avoid over-counting the journeys in the overlap areas.

7.2.2.2 Datatypes based on a cellular grid

There are four datatypes based on a two-dimensional cellular grid structure: Histogram, BlurredHistogram, Bitmap and VectorMap. We refer to the former three datatypes as histograms, since each cell stores a number which indicates a degree of confidence in the presence of road in the cell. The latter datatype stores eight integers per cell, which count the frequency of (quantised) headings of vehicles travelling through the cell.

Each of these datatypes is implemented as a single histogram that spans the bounding box of all of the tiles for which data is held. For the parts of the histogram that lie outside these tiles, if there are any, the values in the cells will be zero. If the instance holds data for two tiles which are separated by some distance, there will be an excessive number of empty cells to be stored. To avoid consuming a large amount of memory to store this histogram, space-efficient sparse-matrix representation techniques can be employed, such as the use of quadtrees.

The merge operation involves creating a new histogram which spans the union of the tiles of the two histograms. The values of the cells from the two histograms are summed, and as before the values in the overlap areas are divided by the number of overlapping tiles. In order that histograms covering disjoint tiles can be combined into a single histogram, it must be the case that the histogram's cell width and height are factors of the width and height of a tile, respectively. Otherwise, the cells would not align precisely and summing values would be non-sensical.

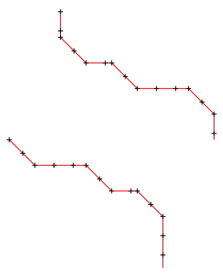
The processing task to convert from a Histogram to a BlurredHistogram involves convolving the cells with a blur filter; the processing task to convert from a BlurredHistogram to a Bitmap involves thresholding the cells' values.

The processing task to convert a Bitmap to a RoadEdges makes use of a contour follower. The current implementation uses the contour follower from the Cantag image processing library [205], which returns a set of closed polygons tracing the boundaries of the regions between cells in which road is present and in which road is absent. The processing task shrinks the boundaries of the overlap regions inwards by three cell widths from each side and clips the contours to this new area. This removes the artefacts produced by the contour follower around the edge of the bitmap and produces instances of RoadEdges that can be safely merged.

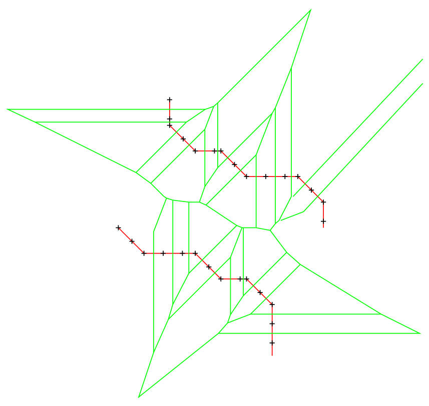
7.2.2.3 The datatype for road edges

The RoadEdges datatype stores a set of piecewise linear polylines. Having had the boundaries shrunk by the processing task which produced a RoadEdges, instances of edges for adjacent regions are guaranteed to agree on the content of the overlap region. Hence, the merging operation can safely juxtapose the data for adjacent regions because it is inconsequential to duplicate edges that precisely coincide.

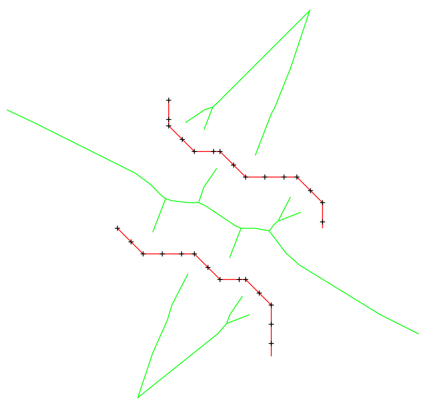
The processing task to convert a RoadEdges to a HairyCentrelines begins by constructing the Voronoi graph of the road edges. This is achieved using Fortune's implementation of the efficient sweepline algorithm [76] by converting the road edges into a series of points. An example is shown in Figure 7.7a. However, as well as containing lines within the road edges, the Voronoi graph of these points also includes lines outside the edges as well as ones which cross through the edges, visible in Figure 7.7b. The lines which cross the edges are removed by testing each Voronoi line for intersection with the road edges, the results of which are shown in Figure 7.7c. This is achieved efficiently by only testing against road edge segments which are known to lie nearby. To remove the Voronoi lines which



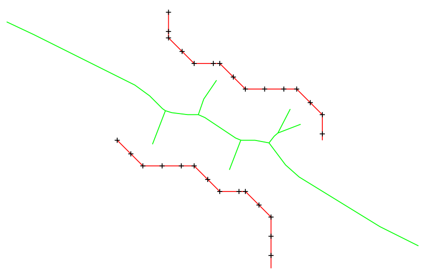
(a) Road edges (red) with Voronoi sites (black).



(b) Voronoi graph (green) generated from the sites (black).



(c) Voronoi edges (green) which do not cross the road edges (red).



(d) Voronoi edges (green) which lie inside the road (red).

Figure 7.7: Removing the Voronoi edges which cross and lie outside the road edges.

lie outside the road edges, a technique akin to the well-known scanline polygon-filling algorithm was adopted: for each Voronoi line, the number of road edges to the left is counted. If it is an even number, the line lies outside the road so is discarded. If it is odd, it lies inside so is retained. The retained edges are shown in Figure 7.7d. The processing task finishes by shrinking the boundaries of the overlap regions to discard the edge-artefacts of the Voronoi graph so that two adjacent tiles agree on the content of the overlap area.

7.2.2.4 Graph-based datatypes

Three datatypes—HairyCentrelines, Centrelines and DirectedGraph—represent graphs, consisting of sets of nodes and sets of edges. In order that the edges are not merely straight lines between two nodes, they may have shape, defined as a piecewise linear polyline travelling between internal nodes.

The processing task that removes the short, ‘hairy’ edges from the HairyCentrelines also removes the overlap regions, leaving behind only the central regions of the tiles. This is done because a road centreline that straddles a tile boundary is guaranteed to match up on either side of the boundary, so there is no longer any need to store any extra data.

Whilst instances of the Centrelines datatype can thus be juxtaposed safely, this is not the case for instances of DirectedGraph: a single road may be viewed as a unidirectional road from the point of view of one tile, but as a bidirectional road from the neighbouring tile. As described in 4.1.2.4, the processing task that produces a DirectedGraph involves sampling the VectorMap at various steps along the roads, counting the number of places at which the road appears to be unidirectional and the number of places at which the road appears to be bidirectional. These numbers are then compared to determine the sense for the entire road. Hence, in order that we can safely merge roads straddling tile boundaries, we need to store these counts for both parts of the road and sum them when joining them into a single road. This enables us to make a better judgment about the sense of the road than we could make just based on the tile-local decisions about the road’s sense.

7.2.3 Evaluation

We consider executing the map generation application after the vehicles’ location histories have already been partitioned into geographic regions. The task graph for producing a map of the roads within a single region is shown in Figure 7.8. When we wish to create a map of more than one region, two questions arise:

1. At which stage in the algorithm should the data from disjoint regions be combined?
2. On which processors should the processing take place?

We will study the first of these questions by comparing the performance of the application when merging takes place at different stages in Section 7.2.3.1. The second question will be addressed in Section 7.2.3.2 by evaluating the performance of the algorithm when varying numbers of processors are available.

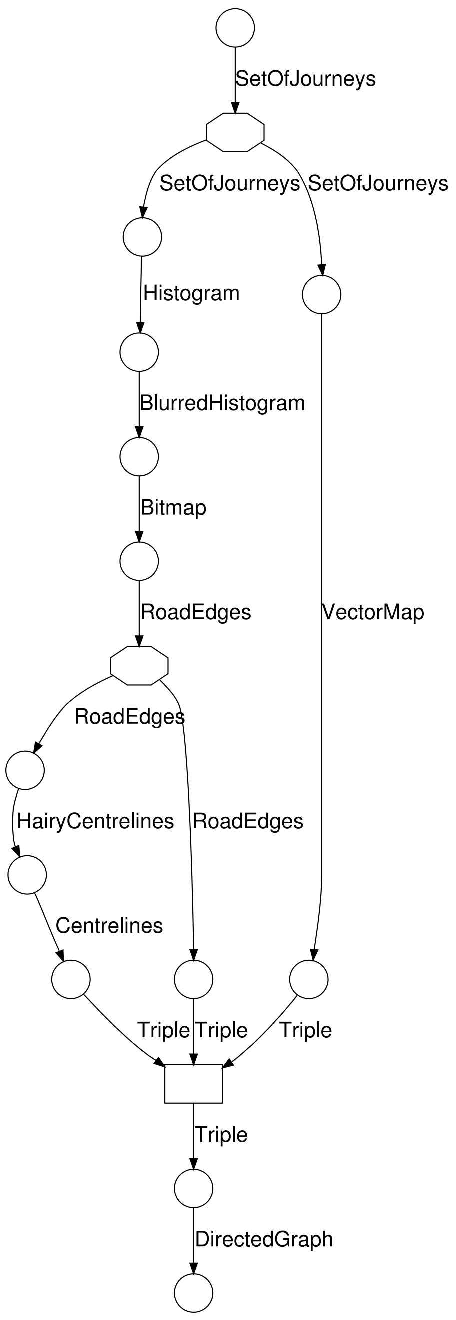


Figure 7.8: Task graph to generate a map for a single tile. Tasks are identified by their shape and input and output datatypes rather than by name.

From	To	Transformations
A	B	Merge–Replication, Merge–Processing^2
B	C	Merge–Processing
C	D	Merge–Processing
D	E	Merge–Processing
E	F	Merge–Replication, Merge–Processing
F	G	Merge–Processing
G	H	Merge–Processing^3, Merge-Reorder*, Merge–Processing

Table 7.1: Summary of transformations used to derive the task graphs. Numeric superscripts indicate multiple applications of a transformation; an asterisk denotes several such applications.

7.2.3.1 Position of merging stage

If data concerning multiple regions is to be processed into a single map, it is necessary at some stage to merge the data from those regions. At one extreme is to combine the vehicle location trace data into a single pool of location histories, and to process that together. At the other extreme is to process each set independently until we have a map of each region, which are then finally stitched together.

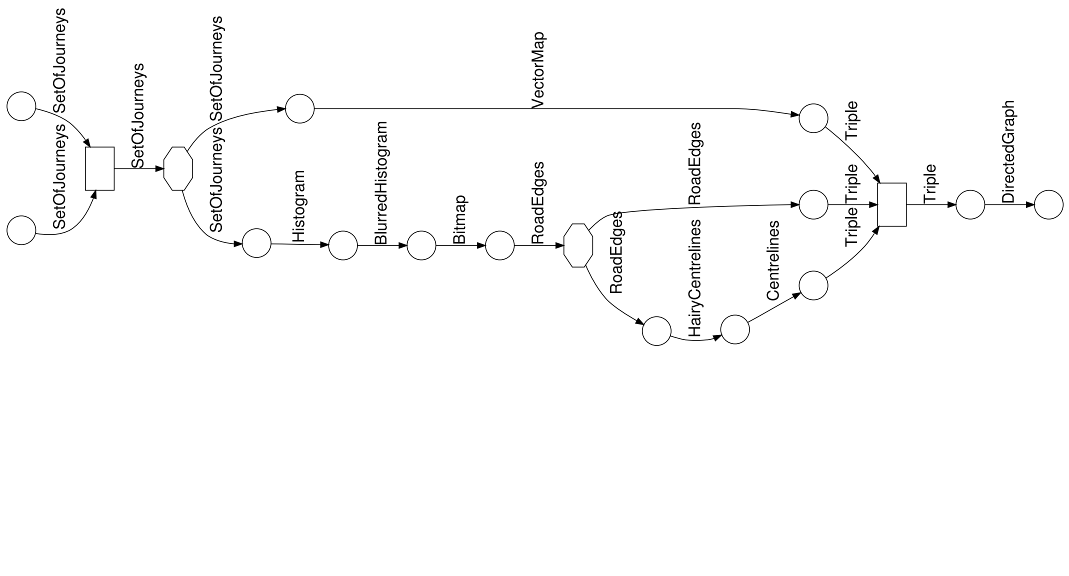
Since all of the datatypes involved in the application are mergeable, the merging could take place at any stage in the pipeline between these two extremes. Since all of the processing tasks are considered to be homomorphisms, it is guaranteed that the stage at which merging takes place does not affect the application's output. As was shown in Chapter 5, these graphs are therefore deemed by the compiler to be equivalent, and it can apply transformations to move between them. Figures 7.9–7.12 depict eight of these task graphs, which we will refer to as Task Graphs A–H. Each successive configuration has a higher degree of potential parallelism than the previous. The task graph transformations used to derive these graphs were identified manually and are summarised in Table 7.1.

We will compare the execution times of these task graphs. For each task graph, the resource graph fed to the compiler describes two identical processors; one set of input data was mapped to each processor. The compiler was instructed to find the best assignments on each of these task graphs without performing any further task graph transformations.

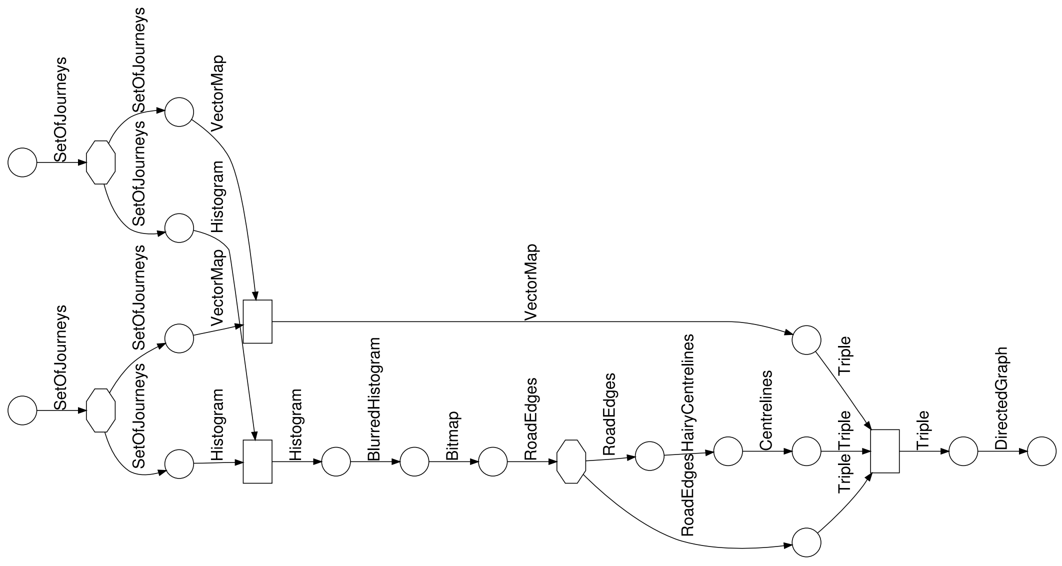
The two tiles each have vehicle trace data containing over 3000 positions. The cell size chosen for the histogram-based data structures was 0.0001° wide and high, which corresponds roughly to a square on the surface of the Earth with sides of length 10 metres. The execution times for the eight configurations are shown in Figure 7.13, with the time for each configuration being an average of three executions.

The large difference between the execution times for task graphs A and H show that employing task graph transformations can have a profound effect. Increasing the parallelism by delaying merging as late as possible means that the tiles can be processed concurrently with some degree of independence.

However, the execution times for configurations B–G are of roughly comparable magnitude. This suggests that increasing the degree of parallelism in the task graph does not

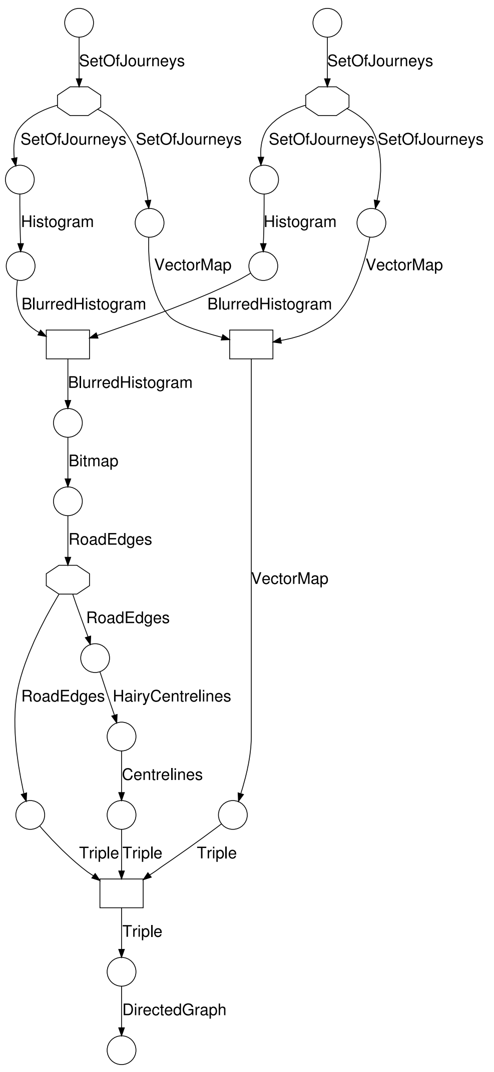


(a) Task graph A: Processing two regions, merging sets of journeys.

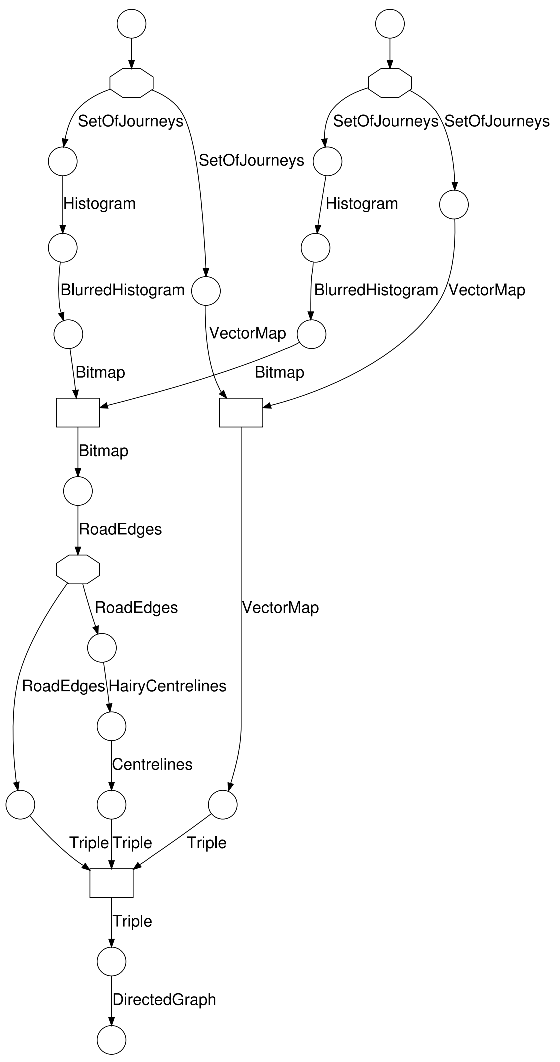


(b) Task graph B: Processing two regions, merging histograms and vector maps.

Figure 7.9: Task graphs A–B.

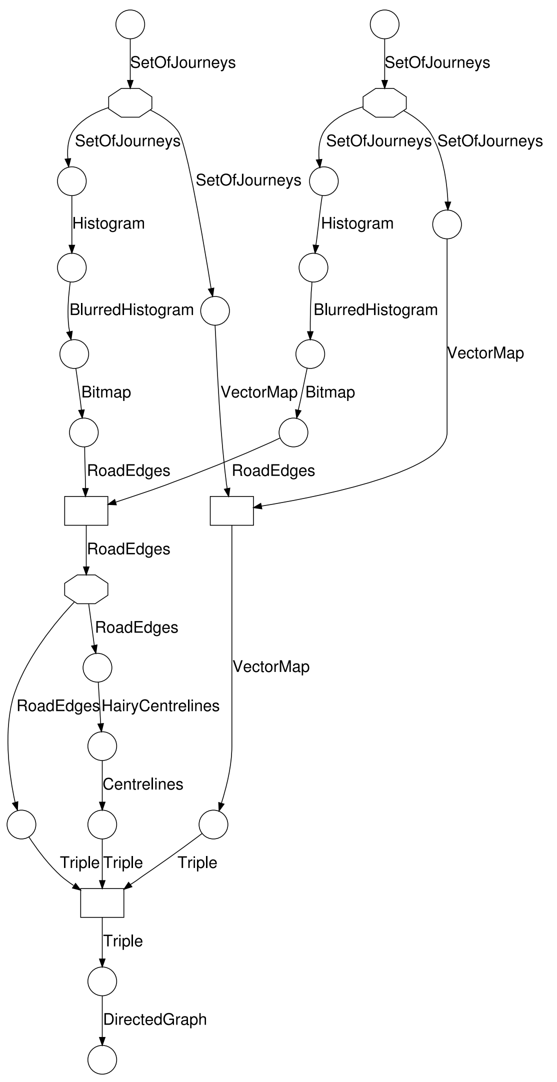


(a) Task graph C: Processing two regions, merging blurred histograms and vector maps.

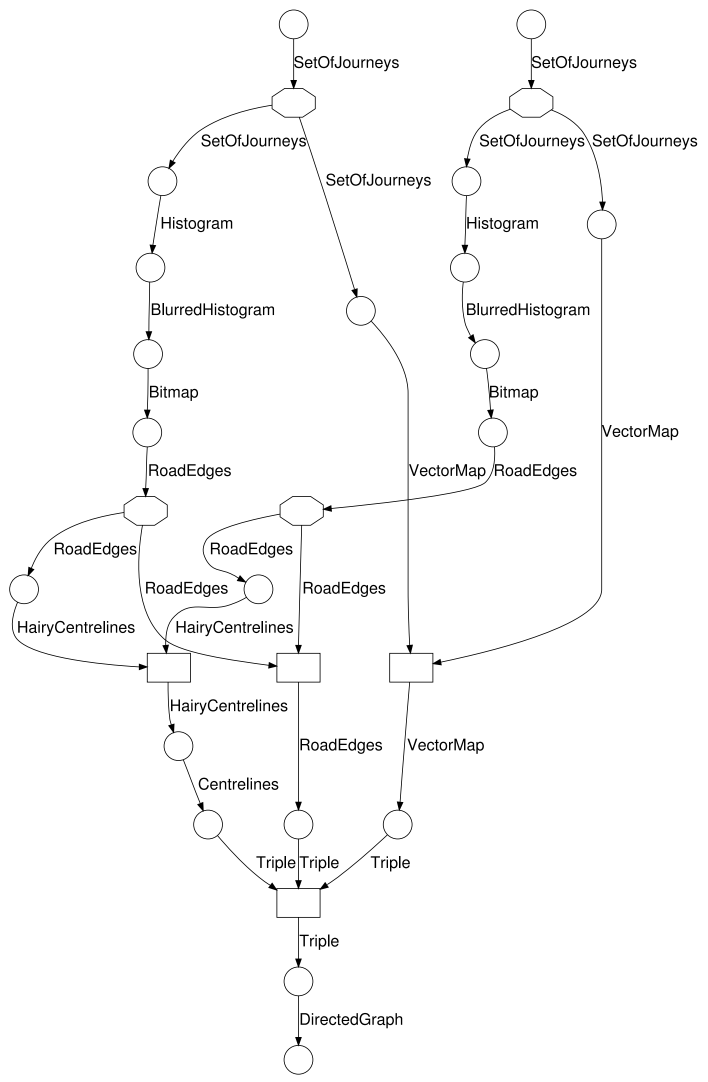


(b) Task graph D: Processing two regions, merging bitmaps and vector maps.

Figure 7.10: Task graphs C–D.

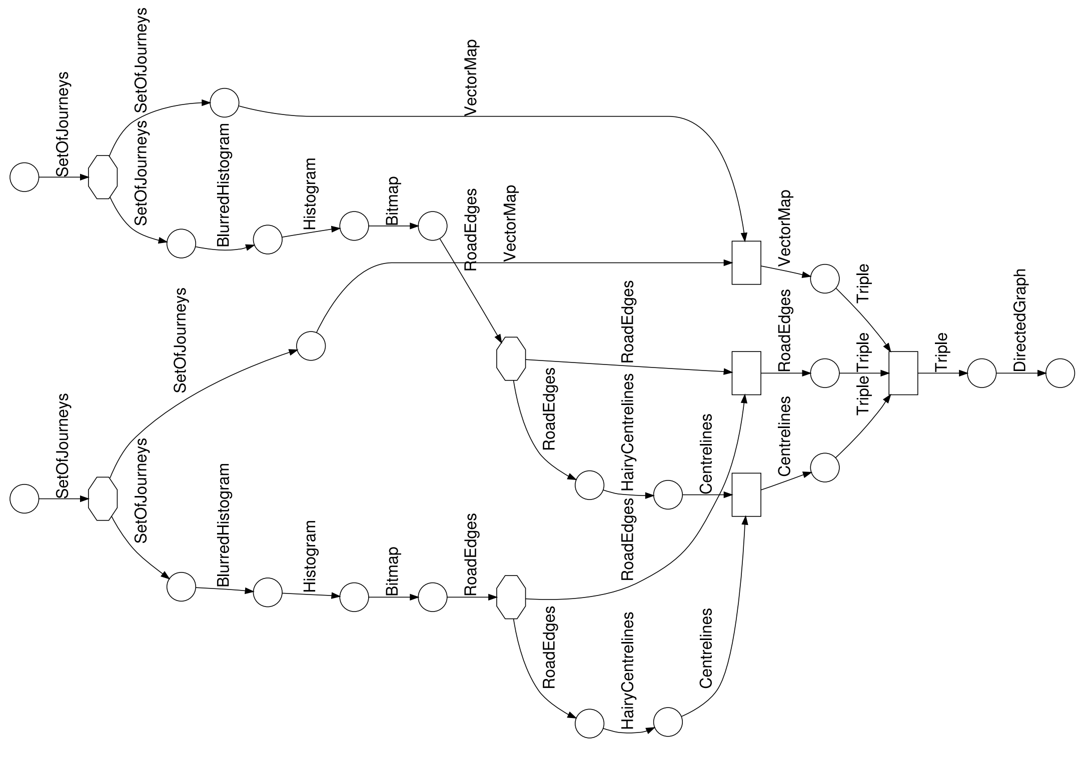


(a) Task graph E: Processing two regions, merging road edges and vector maps.

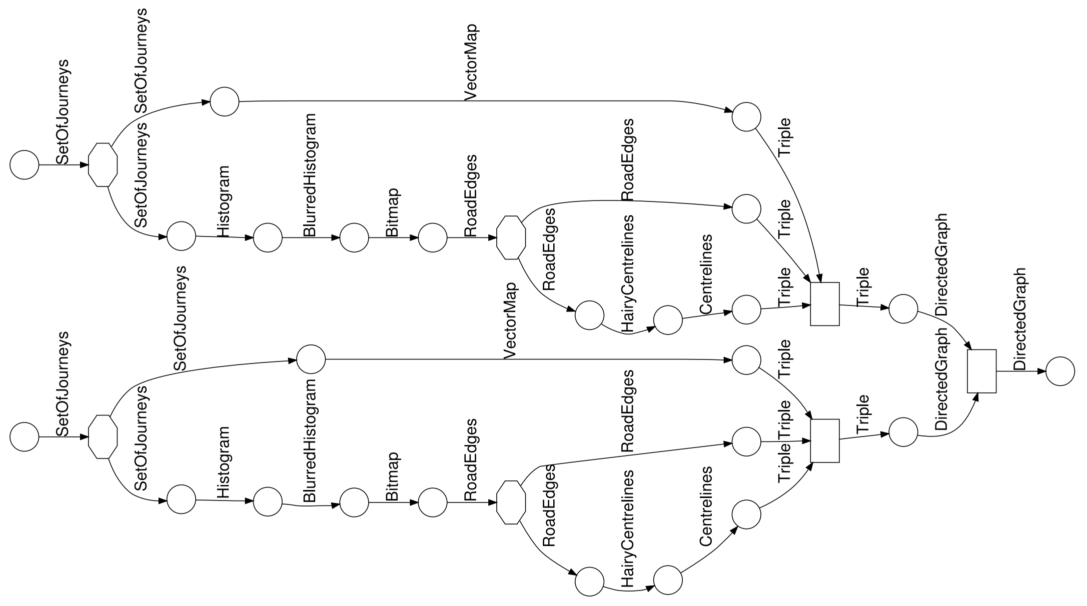


(b) Task graph F: Processing two regions, merging hairy centrelines, road edges and vector maps.

Figure 7.11: Task graphs E–F.



(a) Task graph G: Processing two regions, merging centrelines, road edges and vector maps.



(b) Task graph H: Processing two regions, merging directed graphs.

Figure 7.12: Task graphs G–H.

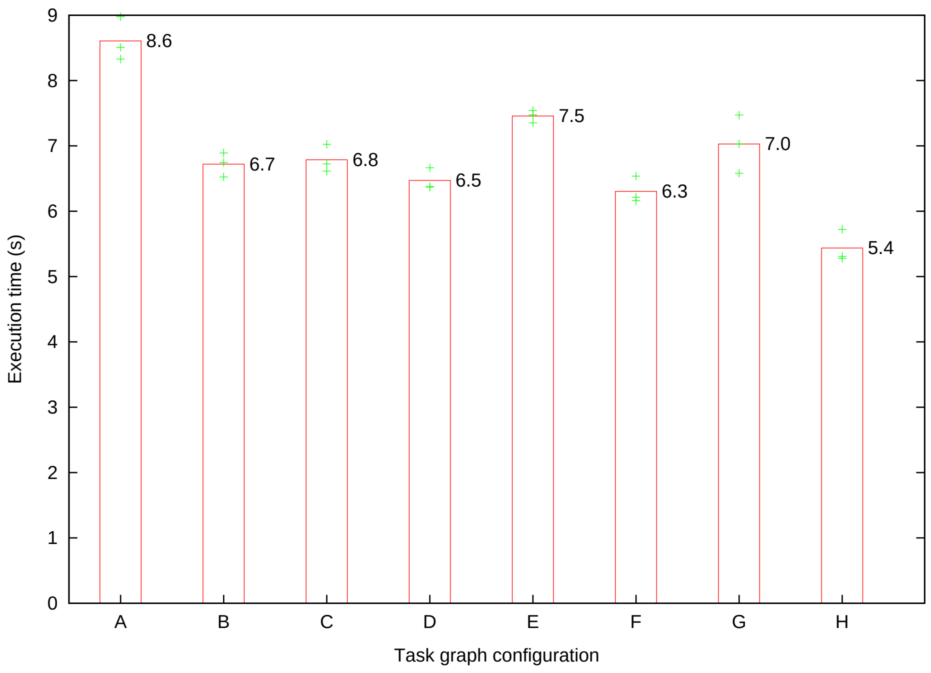


Figure 7.13: Times to execute the map generation algorithm for two tiles, varying the position at which merging takes place. For each configuration, the green points indicate the times of the three executions; the red bar indicates the arithmetic mean of these times.

necessarily lead directly to better performance. It is believed that this may be due to several factors:

- The computational costs of the datatypes' merge tasks vary. The implementations of some merge tasks involve more work than others.
- The task assignment algorithm used by the compiler is necessarily sub-optimal. This means that the assignment that is decided upon may not be the best assignment possible. Furthermore, the implementation in the current version of the compiler is not state-of-the-art and thus may produce inferior assignments.
- The metrics annotated on the source, processing and merge tasks were chosen based on a cursory observation of the algorithm rather than based on detailed profiling data. It is believed that determining more accurate metrics would enable the task assignment algorithm to make better-informed choices about task placement.

7.2.3.2 Number of available processors

For a task graph that processes the vehicle location traces associated with several tiles, we now examine how the number of available processors affects the execution time.

With four dissimilar processors available, we start with the fastest and add in the others in order of decreasing speed.[3] The task graph was designed to process four tiles in

[3]Two of the processors are Xen hosts, each allocated 256 MB of RAM, running on a machine with four

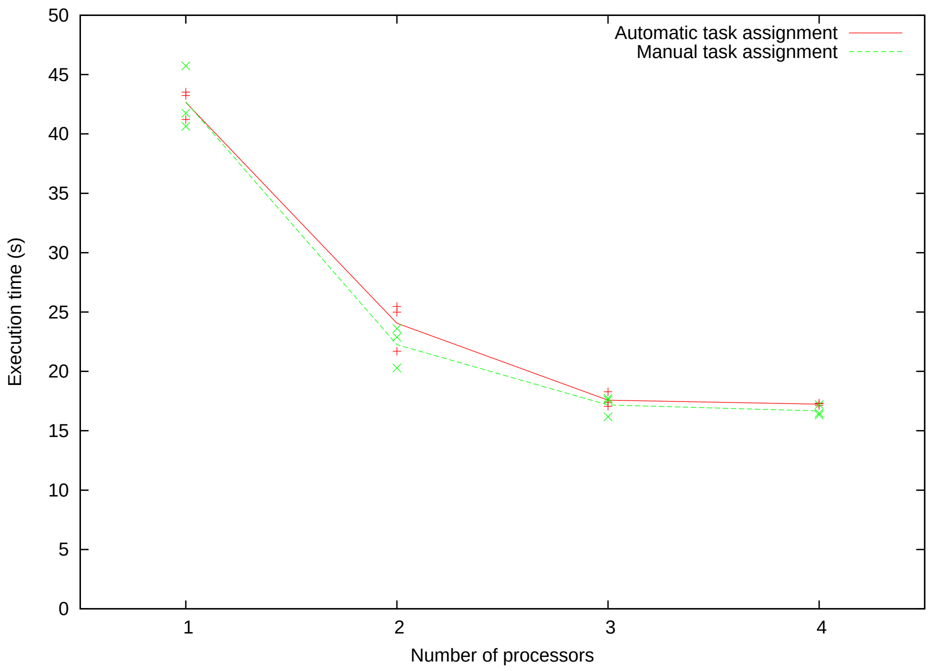


Figure 7.14: Times to execute the map generation algorithm for four tiles on different numbers of processors.

parallel, finally merging at the directed graph stage. The compiler was not permitted to perform any task graph transformations to allow a direct comparison between the effects of different numbers of processors. Ideally, the study would also be performed with task graph transformations enabled, to evaluate their potential benefit, but this was not possible because the size of the task graph meant that the search space was too large for the compiler's naïve strategy for performing transformations. The four tiles used each contained between 40,000 and 130,000 vehicle locations.

Figure 7.14 shows the execution times using different numbers of processors. The red points are from task assignments which were deduced by the compiler's automatic task assignment algorithm; the green points are from hand-crafted assignments. The graph shows that the hand-crafted task graph tends to fare little better than the automatic approach, despite the naïveté of the task assignment algorithm and the imprecise metrics used, as mentioned above.

The curve indicates that, with an increasing number of processors, the execution time diminishes. However, the execution time appears to approach an asymptote. Given that only four tiles are being processed, it is unsurprising that the execution time for two and three processors are similar, since there will be at least one processor on the critical path which has been assigned two tiles to process. However, it is surprising that the execution time for four processors is also similar, because in this scenario each processor can process a single tile. It is suggested that the small difference is because the fastest processor is able to process two tiles in the same time as the slowest processor is able to process one.

3 GHz Intel Xeon CPUs using Citrix XenServer 4.1. The third processor has two 3.2 GHz Intel Pentium 4 CPUs and 1 GB of RAM. The fourth processor has two 2.8 GHz Intel Pentium 4 CPUs and 1 GB of RAM. These machines are interconnected on a 100 Mb/s local-area network.

7.3 Summary

We have described the design of two applications in the late physical binding paradigm of Chapter 5 and their implementations in the language introduced in Chapter 6. One application, ray tracing, was in the style of traditional grid-computing applications. The strategy of performing task graph transformations and automatic task assignment was found to suit this application well because it is inherently parallelisable by partitioning the problem into several sub-problems which can be tackled independently.

The other application, automatic road map generation, was a demonstration that a non-trivial application could be easily described as a task graph and implemented in the language. The task graph paradigm helps us to simply identify a variety of equivalent expressions of an algorithm that have different operational characteristics by the use of the transformations of Chapter 5.

CHAPTER 8

Conclusion

Technological advances in computing and communications mean that a new range of applications involving computing in vehicles is likely to become feasible. In the future, the implementation of these applications may be spurred by the existence of general-purpose computing platforms in vehicles. Such platforms may be provided by vehicle manufacturers in order to avoid redundancy caused by isolated applications or devices which could share common software or hardware. Alternatively, vendors of after-market in-vehicle devices could collaborate to provide interfaces between their devices to allow them to interact. In either scenario, many new applications become possible. A number of applications were suggested in Section 1.2.

Of these applications, some of the most promising yet most challenging to implement involve the collection, processing and distribution of sensor data originating on vehicles. In these applications, vehicles become sensor platforms and nodes in a large-scale wireless sensor network. The vehicular sensor platform described in Chapter 3 is a prototype of such a platform, enabling vehicular sensor data to be gathered, stored and processed. This work highlighted the need to make judicious use of power and cause limited disruption for the vehicle's driver. Furthermore, different sensors were found to require a variety of treatments in terms of sampling, communications and storage.

Perhaps the largest challenge involved in implementing applications which make use of vehicular sensor data is to design them to operate in a scalable fashion. Applications of this kind are usually most efficacious when they use data from many vehicles. For example, if the vehicles operate in different geographic regions then the output becomes more complete. Furthermore, using more vehicles means that more data is available, so that random errors become less significant, potentially leading to a higher accuracy of the output. Hence, the applications must be able to simultaneously handle large numbers—perhaps millions—of vehicles in order to maximise utility.

An example of an application which makes use of vehicular sensor data is that of automatic road map generation, introduced in Chapter 4. This application involves the use of algorithms from image processing and graph theory to process sets of location traces (such as those collected from GPS receivers) into a map of the road network. Generating road maps in this way has some advantages over traditional map-making techniques. For example, the opening of new roads and the closure of existing roads can be reflected in the map in real-time.

Whilst this application involves the processing of one particular type of sensor data—location histories—different applications will make use of other types of data. These applications will also infer higher-level information from low-level raw data. Obvious examples include producing real-time maps of geographically varying quantities, such as temperature or pollution levels. More complex applications, such as an urban traffic management system which optimises the movement of vehicles according to journey time, levels of pollution and congestion, could be implemented along similar lines.

Planet-sized sensor networks will result in a hitherto unprecedented volume, resolution and accuracy of data about the world. This will increase the ability to model the environment and study the effect of human activities on it. This will have an important role to play in using computing to overcome the problems facing the planet's future [115].

The requirement for the scalability of these applications leads to the question of how to arrange the collection and processing of vehicular sensor data in the most resource-efficient manner. One option is a centralised approach in which all of the vehicles transmit their data to a fixed server which performs the processing. However, this suffers from the problems of a communication bottleneck and the presence of a single point of failure. Moreover, it may be difficult for vehicles to communicate their data to a central server cheaply and in a timely fashion. Instead, a distributed approach involving less communication may be preferable.

A country- or even planet-sized network will contain vast quantities of computational resource, constituted from many individual devices with varying characteristics, and there will be many means of communication from one node to another. Judicious selection of processors and communication links to use is crucial in this kind of network because processing may be very costly on some processors, and transferring data may be very costly on some communication links. This is a particular instance of the general problem of making efficient use of available resources. Rather than contributing to the over-consumption of resources, in this way computing can be used to provide strategies for the more prudent use of resources. In order to automate this, an appropriate tool-chain is required.

The traditional approach to programming wireless sensor networks expects the programmer to decide which processors should perform what computation. However, even when ignoring the problems associated with the mobility of vehicles, when given a description of a network containing just a small number of vehicles and fixed nodes, it is non-trivial to determine how best to execute an application in a distributed fashion in it because the trade-offs are complex. Hence it is especially inappropriate to expect programmers to make this decision in large-scale vehicular networks.

Furthermore, the particular computational resources and communication infrastructure available in the network may not be known until run-time, so few sensible decisions about the placement of computation can made in advance by the programmer. There have been several proposals for improved techniques for programming wireless sensor networks, such as the distributed database or active sensors approaches described in Section 2.3. However, these approaches are either too inflexible or do not sufficiently abstract away from a knowledge of the network in which the application is to execute, so are unsuitable.

In Chapter 5, a novel paradigm was developed for programming distributed computing applications in which the computation is described independently of any assumptions about the nature of the network of computers in which it will be executed. This is called 'late physical binding' because the decision about the placement of computation is delayed until execution time. Designing an application in this way involves identifying units of computation called tasks. A graph of tasks is constructed to describe the application, in which edges between tasks correspond to the flow of data. There are six types of task, characterised by the number and type of their inputs and outputs. Merge tasks, which take several inputs of a given type and produce a single output of the same type, are particularly important in applications which involve the processing of large quantities of data, such as those found in vehicular sensor networks. Merge tasks can help to achieve this by aggregating or simplifying the data.

A particular feature of task graphs is that program optimisations are easy to implement. This has led to the definition of a wide range of task graph transformations, specified in Section 5.5.7. Most of these transformations involve changing the order in which tasks are executed. The soundness of the transformations is guaranteed by an appeal to the theory of commutative monoids. The transformations affect the degree of parallelism and other run-time characteristics such as the execution time or energy consumption. Hence the use of transformations can be crucial to the creation of scalable applications whilst the programmer can remain oblivious to the nature of the network in which the application will be executed.

The development of a new programming language in which applications can be specified according to the late physical binding paradigm was presented in Chapter 6 along with its compiler. The language adopts a familiar, object-oriented approach to describing tasks. The compiler takes a task graph along with a description of the network and may apply transformations to the task graph. The compiler determines the best processor on which to execute each task—the problem of automatic task assignment—using a programmer-supplied cost function which is minimised.

The design and implementation of two example applications which demonstrate some of the range of possibilities for this style of programming were described in Chapter 7. Both applications were shown to benefit from late physical binding. With the ray-tracing application, the compiler transformed a non-parallel expression of the algorithm into a parallel form, well-suited to execution in a network containing several processors. With the map generation algorithm which was originally proposed in Chapter 4, task graph transformations were also shown to increase the degree of parallelism, which impacted on the algorithm's execution time. We claim that the language is well-suited to implementing both of these applications, in addition to a broad range of others in the fields of sensor networks, ubiquitous computing, grid computing, and web services.

8.1 Further work

Specific suggestions for further research have been provided in each chapter. This section suggests broader aims for further research in this field.

Communications between vehicles. A fundamental assumption of all of the work described above is that wireless communication between vehicles will become possible, and that vehicles will become citizens of the Internet. At present, this is not yet a reality and is the subject of on-going research.

Handling mobility. Devising strategies to deal with the mobility of vehicles in a network is difficult. High relative speeds and movement through large distances mean that communication links between vehicles and to fixed nodes have rapidly varying characteristics, and frequently establish and break. This creates a challenge for the implementation of applications designed to execute in vehicular networks. At present, the compiler implemented in Chapter 6 produces executables which assume that the characteristics of the network remain unchanged from compilation-time. This problem could be addressed by abstracting groups of nodes in a resource graph and treating them as a single, virtual resource with certain statistical properties—see Section 5.4.1.

Scalability. The experiments relating to the implementation of the automatic map generation application in Section 7.2.3 involved no greater than four processing nodes. However, the calculations in Section 4.2.3 showed that around 750 processing nodes would be required to automatically generate a map of the area of the United Kingdom in a timely fashion.

In order for the paradigm of late physical binding and the compiler to scale gracefully up to thousands of processing nodes, it is necessary to eliminate the need for centralised knowledge of the entire network. This includes the discovery of the nodes which are to participate in the execution of the application and also the determination of the characteristics of those processors and of the communication links, in the dimensions required by the cost function. It also precludes the possibility of centralised optimisation of task placement; instead, a distributed approach must be adopted.

Similar problems have been solved with great success in the implementations of other planet-scale networks, such as the Internet. In these systems, there is no global co-ordination nor global knowledge of the network's characteristics. Instead, each node has limited but sufficient knowledge about the rest of the network that it can collaborate with other nodes to achieve a common goal. Further research should investigate whether similar principles can be applied to automatic task assignment in planet-sized networks. A starting point could be to determine whether it is feasible to allow groups of nodes to dynamically adjust an assignment amongst themselves by making decisions based solely on local knowledge.

Security. An important area for research related to the use of sensor data from vehicles is in security. As well as preventing attacks common to all networks such as denial of service attacks, in sensor networks it must not be possible for malicious participants to invent false data. For data sampled from the environment, it may be possible to identify such fraud by comparing it to other data obtained in the spatial and temporal locality. For other types of data, authentication of the originator and audit trails may be required.

Confidential data must also be handled in an appropriate manner. This is crucial to ensure that the owners of vehicles will be happy to participate in applications in which their data is shared. Safeguards must be implemented to ensure that vehicles wishing to remain private may do so. This is challenging because location privacy is hard to protect when sharing spatially-indexed data. Techniques such as the use of mix zones and varying the resolution of shared data could be applied [24].

When vehicles are able to execute applications developed by third parties, care must be taken over the admission of applications to execute. Perhaps vehicles may only execute signed applications, certified by known authorities. Furthermore, strategies for ensuring that applications are guaranteed to operate within known bounds should also be applied. Techniques such as proof-carrying code and sand-boxing could be employed.

Abstracting the placement of execution. A challenge inherent in the design of any high-level strategy for programming distributed systems is to fully divorce the design of an application from a notion of the processors which will execute it. Keeping these separate means that the decision about the placement of execution can be made automatically, meaning that the application can be adapted to any environment, and that complex trade-offs can be analysed more efficiently than a programmer could manage. This was the motivation behind the use of task graphs to implement the late physical binding paradigm, which largely meets this target. However, sometimes it is not possible to fully abstract away from the notion of the number of processors which will execute a part of the application. For example, in general, n source tasks are required to be specified for a sensor network with n sensor nodes. Hence the number of sensors in the network needs to be known by the programmer. In this way, the use of task graphs falls short of the aim since it is necessary to bind a particular task to a particular node.

This problem could be tackled by adding a further layer of abstraction on top of task graphs in which the general computation to be performed by sensor nodes is declared once rather than once per sensor. This approach seems somewhat reminiscent of the SpatialViews programming language (see Section 2.3.2.1) in which a particular unit of computation is executed on each node of a certain type found to be located within a specified region.

Performance studies. Finally, it would be worthwhile for future work to compare the performance of a program produced by the compiler to a hand-crafted application. This would be instructive from the point of view of evaluating the strength and wider applicability of this work. Performance comparisons between an optimised program and one which has not had task graph transformations applied could also reveal information about the merits of the transformations. This could be useful knowledge for compilers to understand better the circumstances under which it is best to apply each transformation.

APPENDIX A

Vehicle-oriented communications technologies

Various research into vehicle-to-roadside and vehicle-to-vehicle communication has been undertaken, although there are few examples of deployments of these systems of any significant scale. Many of the major vehicle manufacturers are prototyping vehicles equipped with WiFi (802.11a/b/g) and DSRC (dedicated short-range communications). These technologies are expected to feature on production vehicles within a few years.

DSRC is a 5.9 GHz radio band which has been reserved specifically for inter-vehicular communications. This goes hand-in-hand with the IEEE's 802.11p standard for Wireless Access in Vehicular Environments (WAVE). This technology is deemed suitable for use for safety-critical applications [254].

The Drive-Thru Internet project has shown that high-speed Internet access to vehicles via roadside 802.11b access points is feasible [194]. A similar system called MOCCA has attained similar achievements [23]. However, Bergamo et al. point out that performance falls off rapidly in the absence of line-of-sight communication [25].

Murphy et al. have tested the feasibility of using Bluetooth to transfer data between vehicles and to stationary access points [184]. Their findings are that vehicles travelling at 100 km/h can communicate with the access point for 18 seconds. They have also proposed modifications to the Bluetooth protocol which improves its suitability for this kind of application.

A system employing millimetre-wave radio has been suggested by Kim et al. [138]. However, at such high frequencies, propagation distances are very low compared to other radio technologies, so roadside base stations would be required to be of the order of every 100 metres. As well as the cost of deployment of such a system, there is the technical challenge of efficiently dealing with the frequent hand-overs that a fast-moving vehicle would require.

Ribeiro has compared the use of WiFi, WiMAX, MBWA and 3G communication technologies for their suitability to create a wireless vehicular network which includes base-stations [204]. The most significant difference between WiFi and WiMAX was found to be the size of the coverage area, which is an order of magnitude greater for WiMAX, since WiFi was originally designed merely for indoor use. Furthermore, WiFi base stations are more restricted in terms of the number of users which can be supported at a given time. MBWA was found to be very similar to WiMAX, although it was specifically designed for mobile use. Ribeiro suggests that WiMAX will eventually supercede 3G cellular technology.

APPENDIX B

Sensor networks

Over the last few decades, Moore's Law [180] has seen the exponential increase in computing power for a given size of processor. Applied to desktop computers, the physical size of processors has stayed roughly constant and their computing power has increased. However, Moore's Law also applies where the level of functionality is held constant and the size of the processor is decreased. Coupled with improvements in radio technology, this has recently enabled a new range of computational devices which constitute low-power wireless sensor nodes with on-board processors.

B.1 Sensor-equipped devices

Researchers at UC Berkeley have designed and commercialised battery-powered Motes which consist of a microcontroller with internal flash memory, data SRAM and data EEPROM, connected to a set of actuator and sensor devices including LEDs, a low-power radio transceiver, an analogue photo-sensor, a digital temperature sensor, a serial port and a small co-processor unit [109, §3]. These devices have physical sizes of the order of a few cubic centimetres. Figure B.1 shows an instance of the commercial offering.

A similar platform is the BTnode [27], which is based around a microcontroller with a Bluetooth radio module. Rather than having integrated sensors and actuators, the BTnode incorporates several general-purpose interfaces.

Recently, Sun Microsystems have developed an open-source wireless sensor node called a Sun SPOT which natively executes Java bytecode [217]. Figure B.2 depicts two such devices. These devices are modular so that custom sensors or communications interfaces

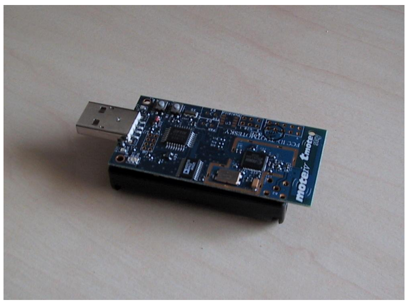


Figure B.1: A Moteiv Tmote Sky mote

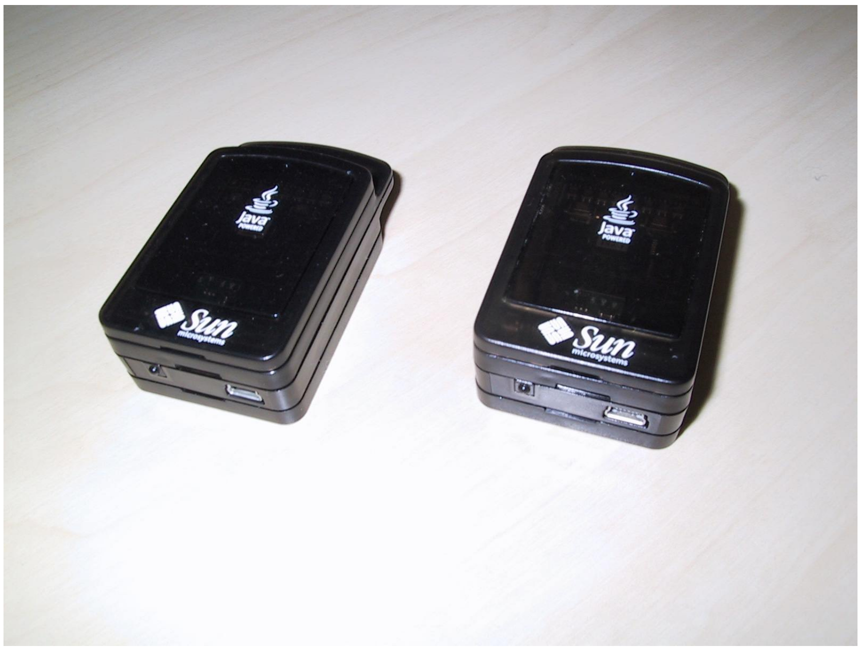


Figure B.2: Two Sun SPOT devices.

can be installed, but the built-in sensors include a three-axis accelerometer, temperature sensor and light sensor, and the built-in communications interface is IEEE 802.15.4 radio. Sun SPOTs can be programmed either via a USB interface or over the radio interface from a USB base-station.

Furthermore, sensor nodes with higher energy consumption are often built from hand-held or pocket computers such as PDAs, or from mobile phones [133].

B.2 Examples of sensor network applications

Sensor networks contain devices sensing a wide variety of ambient conditions: temperature, pressure, humidity, soil make-up, vehicular movement, noise levels, lighting conditions, the presence or absence of certain kinds of objects, mechanical stress on attached objects, and so on [69, §2].

In isolation, these kinds of device are limited in their potential, but when mutually connected in a *wireless sensor network* they can help to achieve many goals in a variety of fields. Akyildiz et al. identify five such fields including military, environmental, medical and the home [3, §2]. We will briefly describe a few examples from these areas.

Military applications. In an environment controlled by the enemy, a military force does not have access to any communications infrastructure but may wish to reconnoitre the environment [3, §2.1]. Hence, a deployment of low-cost sensor nodes, perhaps dropped from an aircraft, are a simple means of obtaining information about the hostile environment. Furthermore, even within friendly circles, the lack of prepared, wired infrastructure on a battlefield means that wireless communications are necessary.

Environmental monitoring applications. Sensor networks have found scientific uses for animal habitat monitoring [43, §3] and animal behaviour studying. The ZebraNet project installed collars on zebras to monitor their movement and flocking patterns [128]. The Great Duck Island project distributed nodes sensing light, temperature, infra-red, humidity and atmospheric pressure in an animal habitat [169]. Similarly, a wireless sensor network has been used to monitor atmospheric conditions along the height of a redwood tree [232]. Other similar monitoring applications include meteorological and atmospheric surveying, where long-term trends are observed.

There is also a variety of event-detection applications which are facilitated by the use of wireless sensor networks. For example, in flood detection [31], forest-fire detection and seismic activity detection it is particularly important to determine an accurate location for the physical extent of the event.

Medical applications. Body-area sensor networks can be used to monitor the health of patients, perhaps notifying a remote doctor when physiological data indicates that there is a health problem.

Home applications. In the vision of pervasive computing, the 'networked home' will involve sensors embedded within home appliances interacting with each other [199].

Other commercial applications. Sensor networks can be applied to the problem of inventory management. Each item in a warehouse has a location sensor; users can query the network to locate a particular item.

Another commonly cited example application is target detection and tracking: identifying a moving object (such as a vehicle or an intruder in a building) and continuously updating a notion of their position [12].

Structural monitoring can also benefit from the use of sensor networks: smart structures have an embedded nervous system of sensors which collects data about its structural integrity, checking for damage or flaws [251].

B.3 Characteristics of sensor networks

Five design requirements for nodes to be used in sensor networks have been specified by Hill et al.: [109, §2]

Small physical size, low power consumption. The device's processing, storage and communication capability are constrained by size and power limitations.

Concurrency-intensive operation. The devices should be able to simultaneously capture data from sensors, process it and communicate it with other devices.

Limited physical parallelism. The device will typically contain a single microcontroller with limited flexibility in the processor–memory interconnect.

Diversity in design and usage. In order to keep the size and weight of the device minimal, the hardware carried by a device will be application-specific, so there will be a large variation in the physical nature of devices.

Robust operation. The devices must be reliable enough to work in harsh environments, where energy limitations preclude the use of redundancy techniques.

Sensor networks have several characteristics that distinguish them from traditional data networks. As well as the differences in constraints on energy, computation, storage and communications bandwidth, Boulis et al. identify that the network's usage model is different [35, §II.A]. Traditional data networks typically act to bring together two parties for point-to-point communication. The user is connected to one node and requires a service from another node. Instead, in a sensor network, a user typically requires results from the network as a whole; the sensor network is viewed as a single distributed system.

Akkaya and Younis further point out [2, p326] that a sensor network may have no global addressing scheme, which precludes the use of IP-based protocols. Furthermore, the sensed data may contain some redundancy because two nearby sensors are likely to give similar readings.

Many sensor network applications require techniques from ad-hoc networking. However, there are a number of general distinguishing differences between wireless sensor networks and ad-hoc networks: [3, p394]

- sensor networks may contain vast numbers of nodes;
- sensor nodes may be densely deployed;
- sensor nodes are prone to failure;
- the topology of a sensor network may change rapidly;
- communication in a sensor network is often achieved via local broadcast rather than point-to-point messaging;
- sensor nodes are limited in power, computational capacity and memory; and
- sensor nodes may not have globally unique identifiers.

B.4 Node discovery

An important issue in wireless sensor networks is the provision of a means by which other nodes can be discovered [193]. In scenarios where a large number of nodes are scattered in a region, the network needs to be self-discovering and self-organising. In addition, when the sensor nodes are mobile, the network needs to be able to adapt to changes in the network topology.

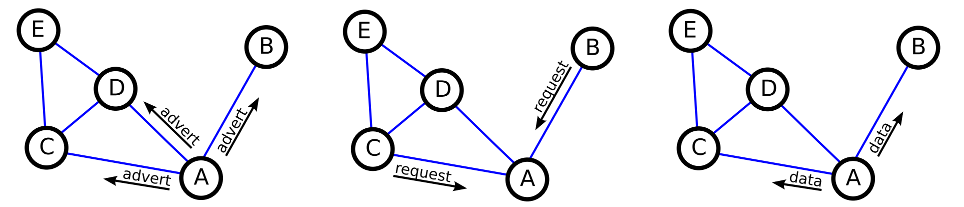
Jini is a discovery protocol which is accessed through a Java API [241]. A discovery scheme must have some means of cataloguing the participants and the services they expose. In Jini, this catalogue is centralised, where nodes register with a central repository. The centralised nature of the catalogue means that Jini is unsuitable for use in large-scale networks where nodes cannot cheaply or reliably communicate with a central server. Furthermore, it cannot be used in networks whose nodes are incapable of running Java, perhaps due to energy, memory or computation constraints. However, some steps have been made to support resource-constrained devices, particularly through the use of the Jini Surrogate Architecture, in which a resoure-rich device acts as a proxy for a resource-constrained device.

Other protocols adopt a distributed cataloguing approach, such as the Bonjour protocol [10] for use in an IP network, or Universal Plug and Play (UPnP) [206] which uses the simple object access protocol (SOAP). In these schemes, the protocol is inherently peer-to-peer. Typically these are intended to run in zero-configuration environments, where there is no assumption of services like DHCP or DNS.

Specifically designed for low-power sensor networks, Dyo et al. propose an energy-efficient discovery protocol in which nodes have a daily energy budget and spread out scans for neighbours according to past activity history [63].

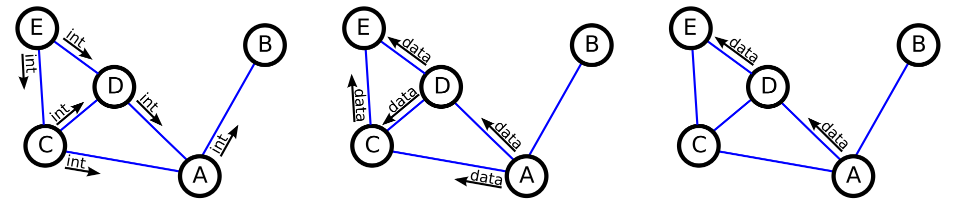
B.5 Routing

In a sensor network, multi-hop routing is often preferable to single-hop routing because transmission power is proportional to the square of the distance between the transmitter and the receiver.



(a) Node A sends advertisements to its neighbours. (b) Nodes B and C reply with requests for the data. (c) Node A sends its data to B and C.

Figure B.3: An overview of the SPIN protocol.



(a) The sink node, E, sends out interests which are propagated through the network. (b) Data from the sensor on node A is sent in the directions from which interests were received. (c) One path from A to E is reinforced.

Figure B.4: An overview of the Directed Diffusion protocol.

There have been a wide range of protocols proposed to arrange the relationships between nodes for the purposes of the transmission of data across the network. Akkaya and Younis classify routing protocols into three categories: data-centric, hierarchical and location-based [2]. We will examine the major protocols in these three classes.

B.5.1 Data-centric routing protocols

Data-centric protocols do not rely on the ability to name individual nodes, but merely rely on the ability to name the data which the application desires. Hence, these protocols tend to be query-based: the sink injects a query into the network and waits for data to return.

A naïve means of relaying data in a sensor network is *flooding* [2, §2.1]. In this approach, nodes in possession of data broadcast this data to all neighbours. Eventually, the data will reach the sink. Whilst this approach does not require any intelligent routing or topology maintenance, it sends more messages than are necessary which incurs a large communication cost. The redundancy in communication is embodied in the problem of *implosion*, where a node receives more than one duplicate of a message from different neighbours.

Rather than forwarding a message to all neighbours, the *gossiping* approach to data dissemination involves nodes selecting random neighbours with which to share received data. This makes implosion much less likely but means that the propagation of data through the network is slower.

A less naïve strategy is the SPIN protocol in which desired data is described using high-level meta-data [107]. Sensor nodes advertise the meta-data describing the kind of data that they can produce (Figure B.3a). These adverts are sent to neighbouring nodes. If any such node wishes to receive data, it sends a 'request' message back to the sensor (Figure B.3b) which replies with some data (Figure B.3c). In turn, a node which has received the data now advertises to its neighbours to invite them to request the data, and so on. This protocol avoids the problem of implosion, takes advantage of meta-data being smaller than the data it describes, and avoids data being sent unnecessarily.

Perhaps the best-known routing strategy is Directed Diffusion [122]. Like SPIN, Directed Diffusion also makes use of meta-data to describe data. Unlike SPIN, where the meta-data travels from data sources to the sink in the form of advertisements, in Directed Diffusion the meta-data travels from the sink to the sources in the form of *interests* (Figure B.4a). An interest is propagated throughout the network, where it is cached at nodes. Incoming data is compared against cached interests, and forwarded in the direction from which the interest was received if it matches (Figure B.4b). Hence, data is sent to the sink on demand. When a sink starts receiving new data, it can *reinforce* the link from which it received the data by reiterating the interest, requesting a higher data rate (Figure B.4c). In turn, this interest will propagate back along the path, reinforcing the entire path as the main channel for data to be relayed along.

A number of variants of Directed Diffusion have been proposed. Shah and Rabaey observed that the reinforcement of a single path in Directed Diffusion can cause uneven energy consumption in a network, depleting the energy resources of the nodes on the reinforced path at a faster rate than those of other nodes. Instead, they propose to occasionally send data along sub-optimal paths [215]. Paths are chosen probabilistically, with the probability inversely proportional to the cost of the path. In this way, the aim is for the maximisation of the lifetime of the network as a whole.

Rumor Routing is another variant of Directed Diffusion, which avoids the need to flood the entire network with interests [36]. Although the description of the meta-data expressing the desired data in the interests is often smaller than the data itself, this is not always the case. In Rumor Routing, sensor nodes send out 'agents' which propagate information about the data across the network in a single direction. A query generated by a sink travels through the network in a random direction until it is received at a node which the agent has passed through.

Gradient-based routing is another derivative of Directed Diffusion in which nodes receiving interests keep track of the number of hops from the sink [214]. This value is known as the 'height' of a node. Data is forwarded from a node to the neighbour with the link of steepest gradient, indicating the shortest path to the sink. As well as incorporating the number of hops to the sink, the height metric can be manipulated by a node to control how desirable it appears for neighbouring nodes to use. For example, if a node's energy resources are running low, it can increase its height to discourage other nodes from sending data to it.

B.5.2 Hierarchical routing protocols

Hierarchical routing protocols involve appointing particular nodes to perform distinguished roles within the network. A popular form of hierarchical routing is for nodes to form clusters which have a distinguished 'cluster head' which communicates on behalf of the members of the cluster to the sink. This approach provides the potential to perform aggregation at the cluster heads, which means that fewer messages need to be sent.

The Low-Energy Adaptive Clustering Hierarchy (LEACH) scheme [106] forms local clusters of nodes based on received signal strength. The cluster heads act as routers in the direction of the sink for the members of their clusters. Over time, cluster heads change randomly to balance energy consumption.

Power-Efficient Gathering in Sensor Information Systems (PEGASIS) [155] adopts a rather different hierarchical approach, in which the hierarchy is linear rather than tree-based. The nodes organise themselves into a chain, which traverses all nodes in the network, such that each node only ever transmits to and receives from a particular neighbour. One node in the chain is then randomly selected to communicate directly to the sink; this selection changes over time to balance energy consumption. Compared to the cluster-forming in LEACH, the chain-forming in PEGASIS is much cheaper due to a reduced amount of communication required [71, p20].

The Threshold-Sensitive Energy Efficient Sensor Network Protocol (TEEN) [170] takes a *reactive* approach to transmitting sensor data from nodes. Rather than the above proactive approaches, where data is sent at regular intervals or whenever a node receives a query, in a reactive approach, data is only transmitted when there is a significant change in the sensed value. In this way, energy is saved through reducing the number of transmissions at the expense of a higher-fidelity view of the sensed data. This is achieved through the announcement of two thresholds by cluster heads to the members of their clusters: a hard threshold and a soft threshold. Nodes only transmit data updates if the new value exceeds the hard threshold and when the data is different from the last transmitted value by at least the soft threshold. As with LEACH and PEGASIS, the nodes take turns at being cluster heads.

B.5.3 Location-based routing protocols

Location-based routing protocols assume that nodes have location sensors and use the information about their positions to reduce the number of messages that need to be sent.

Rodoplu and Meng proposed a routing scheme based on a 'minimum energy communication network' (MECN) [208]. This is a simple local optimisation scheme which attains a network-wide global minimum energy usage in stationary networks, and close to the minimum in mobile networks. A node i determines the nodes j for which it is preferable to route via an intermediate node rather than transmitting directly to j. From this, minimum-cost paths throughout the network can be derived.

Geographic Adaptive Fidelity (GAF) [252] is an ad-hoc network routing protocol which identifies groups of nodes which are equivalent from the point of view of routing, and sends the unnecessary ones to sleep. Two nodes are equivalent if they are located in the same cell of a square grid. Nodes use location sensors to establish which cell they belong to and communicate with the equivalent nodes to negotiate which node should sleep and for how long.

Finally, Yu et al. have proposed a Geographical and Energy Aware Routing strategy (GEAR) [257] for routing queries concerning particular geographical regions. A node relays a query in the direction of the target region whilst also taking account of its neighbours' energy consumption and attempting to balance this.

APPENDIX C

Tightly-coupled systems

Tightly-coupled distributed systems are distinguished from loosely-coupled distributed systems by memory being shared between processors, depicted diagrammatically in Figure 2.1. Shared memory implies that reads and writes to memory locations are immediately globally visible. This implies some degree of inter-dependence between processors; on the other hand, the concurrent nature of multiple processors implies some degree of independence between processors. Hence, there are a number of difficulties which arise when programming concurrent applications in an environment in which memory is shared.

C.1 Dealing with concurrency

Before languages began to provide facilities to deal with the problems associated with concurrency, the norm was for programmers to create buffers to support inter-task communication. The buffers would support *get* and *put* operations to consume a value from and insert a value into the buffer, along with *full* and *empty* operations to determine the state of the buffer. In order to support concurrent access to a buffer, semaphores were used which exploit atomic machine instructions to cause a process to wait whilst another is performing an operation on the buffer.

However, this paradigm was at too low a level to be practically useful for controlling access to larger data structures. Relying heavily on the programmer's correct handling of semaphores, it was prone to error.

We examine a few solutions adopted by conventional programming languages to work around or avoid these problems altogether.

C.1.1 Object-oriented programming

Java attempts to assist programmers to get around these problems by providing primitives such as *mutual exclusion locks* and *condition variables*. Mutual exclusion locks, or 'mutexes', can be used to demarcate critical regions in a program in which it must be guaranteed that no more than one thread is executing at any given time, removing the possibility of race conditions between threads. Condition variables are available to the programmer to avoid the race condition associated with busy-wait loops, where the process spins while waiting for a certain condition to hold.

Arslan et al. argue that the Java approach is too low-level, and is thus hard to understand and prone to programming errors [14]. Instead, in their proposal for Simple Concurrent Object Oriented Programming (SCOOP), they propose a higher-level approach which provides transaction-semantics and a type system which can statically guarantee concurrency-related safety properties. In their programming model, each object has a single owning 'processor', which may be a thread or a process on a physical CPU. For a method invocation, if the invoker has a different owning processor to the object on which the method is invoked, the call is asynchronous. A keyword *separate* is introduced, used when declaring a variable to indicate that it may be handled by a different processor. Calls to methods with *separate* parameters block until the value passed is exclusively handled by the caller. This allows static type-checking to ensure safety against synchronisation defects.

An alternative approach to object-oriented concurrency is to use *active objects*. In traditional object-oriented programming, the object abstraction encapsulates data and actions which can be performed on the data. The active objects paradigm takes this concept one step further and encapsulates data and actions along with a thread of control. This one-to-one mapping of threads to objects means that the threads must be lightweight enough to allow hundreds of thousands to co-exist, and to allow them to be created and destroyed with minimal cost. Programming with active objects usually entails a continuation-passing style in which active objects voluntarily pass control to others. Kilim is an implementation of active objects in Java [223].

C.1.2 Functional programming

Functional programming languages such as ML [178] and Haskell [117] are gaining in popularity. One reason for this gathering momentum is that they admit many compiler optimisations because of the pure, near-mathematical form of their programs. Compilers are becoming sufficiently clever that large, semantics-preserving optimisations can be performed which achieve performance at least as good as hand-optimised procedural programs, whilst allowing the programmer to express the program at a higher level. Burstall and Darlington describe a system of rules for transforming recursive functional programs into more efficient ones whilst maintaining semantic correctness [38]. However, functional programming languages are also inherently amenable to parallelisation because they are *referentially transparent*. This means that the meaning of an expression is always the same and is built from its component sub-expressions, and does not depend on computation that has occurred in the past.

Consider the expression $f(g(a), h(b))$ in a functional language. Since g and h are functional—that is, they have no side effects—the order of execution of $g(a)$ and $h(b)$ does not affect the outcome as it might in a procedural language. Hence they could be executed independently, in parallel.

Consider also the example of the map function which applies a function to every element of a list. This is amenable to parallelisation since the function can be applied to each of the elements in the list in any order. Hence, a large list could instead be partitioned into distinct sub-lists, each processed independently, and the results appended to give the same result. Hammond has presented an overview of the history of parallelism in functional languages [99].

C.2 Architectural models

Over the years, a number of models of multi-processor machines have been proposed as potential successors to the von Neumann architecture. These models provide high-level abstractions which enable algorithm designers to specify algorithms in an architecture-neutral fashion whilst modelling realistic features which are readily implementable in hardware.

The parallel random access machine (PRAM) model assumes a single shared memory which any processor can access [77]. This early model was criticised for its gross over-simplifications, primarily that it unrealistically assumed there to be no cost for communications between processors. This meant that it encouraged the design of parallel algorithms which make excessive use of communication, which do not perform well in practical multi-processor machines. Despite these flaws, it is still a model in widespread use.

Many extensions to the PRAM model have been proposed, which address the basic model's over-simplifications. These include better modelling of memory contention; of asynchrony of processors; of the bandwidth and latency of inter-processor communication; and of memory hierarchy [50, p10].

A survey of parallel algorithms was presented by Karp and Ramachandran [136]. These algorithms include list, tree, graph and sorting algorithms. They employed the PRAM model as the model of computation, despite acknowledging its weaknesses, describing it as an "extremely useful vehicle for studying the logical structure of parallel computation in a context divorced from the issues of parallel computation".

C.3 Multi-processor machines

Most contemporary commodity multi-processor machines are of the variety termed symmetric multi-processors (SMP). These consist of a set of identical processing elements connected to a single shared memory via a common bus. Hence the cost of accessing memory is the same for all processors. The adjective *symmetric* refers to the homogeneity of the processors.

The programming model for an SMP machine is identical to that for a uniprocessor machine. Since memory is globally shared, accesses to it are sequential. However, with the concurrency introduced by the presence of multiple processing elements, it is now possible that instructions from a single process could execute on different processors. Therefore the programmer must ensure that his program interacts carefully with the operating system so that only streams of instructions which are independent are executed on separate processors. Typically this is achieved through the use of threads within a process, and the operating system will only use a single processor for a uni-threaded process.

C.4 Data-parallel architectures

Data-parallel computer architectures are characterised by operations which can be performed in parallel on each element of a large, regular data structure [51, §1.2.5]. Array processors were born out of the recognition that many mathematical computations involved performing the same operation on all elements of an array or matrix. These computers formed the basis of many supercomputers of the 1980s, popular in scientific computing. In the von Neumann style, programs logically contain a single locus of control in programs. Typically, a control processor would broadcast each instruction to each of a bank of processing elements, to which each is given an element of the data structure. In these machines, there is usually some scope for communication between neighbouring processors, for example to cause all processors to pass an intermediate value to the processor to their right. For some applications, this favours a two-dimensional grid layout rather than a one-dimensional array of processing elements.

Computers falling into this category merit classification as tightly-coupled distributed computers since the programming model was usually oriented around a shared, global memory, across which the data structure to process was laid. Each memory element was owned by a particular processing element. However, due to the deterministic, single-threaded nature of the programming model, these architectures fall largely into Flynn's SIMD category.

APPENDIX D

Modelling distributed computing

There have been a variety of approaches to modelling concurrent computation. The majority of these calculi have in common a concept of a *process*[1] which performs, often repeatedly, a sequence of computational steps, which may involve communication to or from another process as a fundamental operation.

MacQueen distinguishes between calculi in terms of the nature of the communication channels between processes [164, §2.3.2]:

Direct communication entails processes having *globally* valid names, and any process can communicate directly with another process if it knows its name. Hoare's CSP is an example of a calculus employing direct communication.

Indirect communication entails the existence of explicit channels between processes. Milner's CCS and Kahn's dataflow approach use indirect communication, the latter with storage of messages on channels and the former without.

We will describe the models mentioned in more detail.

D.1 Process algebras

There are many algebras which have been invented for the purpose of modelling concurrent computation. Baeten states that "a process algebra is the study of the behaviour of

[1]As in Chapter 2, we use the term 'process' throughout this appendix to refer to the encapsulation of computation and not necessarily to refer to an operating system process.

parallel or distributed systems by algebraic means" in an overview of the history of process algebras [17].

A process algebra provides a means of specifying a parallel system and a way of verifying properties about it through equational reasoning. Furthermore, two equivalent systems with different compositions can be shown to be similar using mathematical techniques.

However, the worth of these benefits hang on the ability of the algebra to model complex systems. The calculi described below scale gracefully. For this to be the case, Milner states that a process algebra fundamentally needs a 'product' operator which composes two components into a single component, taking account of their interaction, and an 'encapsulation' operation which hides the internal details of a component [176, p303].

D.1.1 Milner's CCS

In Milner's calculus of communicating systems (CCS), a process is modelled as having input and output ports [177]. The actions it can perform are to receive an input signal or to emit an output signal. A process P which waits for an input on port α then emits an output on port $\overline{\beta}$, then repeats, is modelled as

$$P = \alpha.\overline{\beta}.P.$$

For example, a binary semaphore can be modelled as follows:

$$\begin{aligned} A &= p.B \\ B &= q.A \end{aligned}$$

The behaviour A represents the 'free' state and B represents the 'busy' state. Input signals p and q are used to alternate between these states.

A process Q which can output on either one of two ports $\overline{\alpha}$ or $\overline{\beta}$, then repeats, is modelled as

$$Q = \overline{\alpha}.Q + \overline{\beta}.Q.$$

In this way, addition is used to denote non-determinism: any one of the summands could execute next. Addition is known as the 'choice' operator.

Processes are composed by parallel composition, denoted by a vertical bar. A process outputting on port $\overline{\alpha}$ composed with a process waiting for input on port α can mutually communicate, known as a 'reaction'. Informally,

$$(\alpha.P + \ldots) \mid (\overline{\alpha}.Q + \ldots) \quad \longrightarrow \quad P \mid Q.$$

This style of communication is indirect according to MacQueen's classification and takes place synchronously since there is no storage on the channel.

Graphically, processes are drawn as nodes with labelled output ports and arcs linking complementary ports α and $\overline{\alpha}$, as shown in Figure D.1.

Encapsulation of a network of processes is achieved through a *restriction* operator. The expression $P\backslash\alpha$ behaves like P but does not allow other processes to communicate via

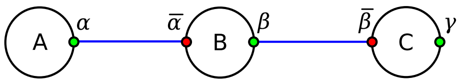


Figure D.1: Three CCS processes, showing complementary ports linked.

channel α. Hence, channel α is hidden. This allows a complex network to behave abstractly just like an individual process: communicating in the same way, being subject to the same composition operations, and having the same sort of semantics.

Milner shows how CCS can be used to provide a semantics to a conventional programming language called P [177, §9]. This language consists of standard control-flow constructs such as *if-then-else*, *while* loops, procedure calls and sequential composition as well as parallel composition. Expressions and commands of P can be translated into CCS recursively.

D.1.2 Hoare's CSP

Hoare's communicating sequential processes (CSP) [111] is similar in spirit to CCS. Van Glabbeek has surveyed the differences between the two algebras [234]. Where Milner uses one choice operator, Hoare distinguishes between two: a deterministic version, known as 'external choice', where the chosen path is determined by the environment; and a non-deterministic version, known as 'internal choice'.

Like CCS, communication between processes is synchronous. As mentioned above, CSP uses direct communication, according to MacQueen's taxonomy. Two processes communicate when they share an action α, where α is some global name, whereas in CCS, processes share a unidirectional channel consisting of α and $\overline{\alpha}$. Whilst CCS uses only one operator for parallel composition, CSP uses two: one for interleaving, and one for communication. The communication operator is similar to the use of parallel composition with restriction in CCS.

D.1.3 The π-calculus

The π-calculus is an extension of CCS which adds the concept of *mobility*, which is the capacity to change the connectivity of a network of processes [179].

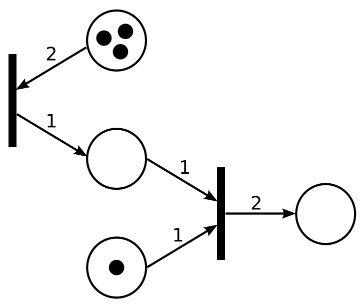
In the π-calculus, rather than merely sending a signal to an output port or receiving a signal from an input port, the actions which processes can perform involve passing a *name* to an output or receiving a name from an input. For example, the expression $\overline{x}\langle y\rangle$ denotes the output of name y down the channel with name x. Conversely, the expression $x(u)$ denotes the input of a name from channel x, where u is bound to the value received.

Now, a reaction is defined as follows:

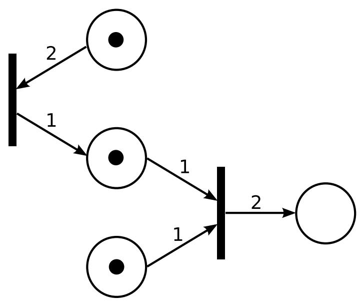
$$(x(y).P + \dots) \mid (\overline{x}\langle z\rangle.Q + \dots) \quad \longrightarrow \quad P[z/y] \mid Q,$$

where $[z/y]$ denotes the substitution of free occurrences of y with the name z.

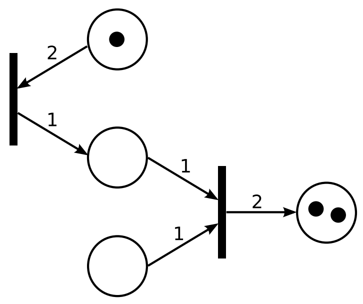
Mobility arises when names of channels (perhaps even private channels, created using the restriction operator) are passed to other processes. Milner exemplifies this concept



(a) The initial marking.



(b) The marking after the first transition has fired.



(c) The marking after the second transition has fired.

Figure D.2: An example Petri net, with arc weights indicated as edge labels.

by analogy with the telephone system, whereby the name that is passed is a telephone number: telling someone your telephone number creates a new communications link to that person [179, p93].

D.2 Petri nets

Perhaps the earliest approach to model parallel computation was in 1962 by Carl Adam Petri. Since then, his 'nets' have become widely researched and applied to many areas of computing [183]. A Petri net is defined in terms of a set P of *places* and T of *transitions*, a set of directed *arcs* $F \subseteq (P \times T) \cup (T \times P)$ going between places and transitions. Each arc is assigned a *weight* by a weight function $W : F \to \mathbb{N}$. An initial *marking* $M_0 : P \to \mathbb{N}_0$ assigns a number of *tokens* to each place.

Petri nets are 'executed' by transitions *firing*, which results in a new marking. A transition t fires when each predecessor place p has at least $W(p,t)$ tokens. This causes those numbers of tokens to be removed from each p, and $W(t,q)$ tokens are added to each output place q.

Figure D.2 shows an example of a Petri net and its execution. Places are denoted by circles and transitions are denoted by vertical bars. Markings are indicated by drawing tokens inside the places; the initial marking is indicated in Figure D.2a. The first step of execution is a firing of the transition on the left, which consumes two tokens from the uppermost place, depositing a single token in the central place; the resulting marking is shown in Figure D.2b. Subsequently, the second transition can fire, consuming a single input token from each of its inputs and depositing two tokens in the final place, shown in Figure D.2c. After this point, no further progress can be made.

The control flow of a parallel system can be modelled in this elegant and intuitive fashion. Phenomena such as deadlock are simple to model. However, MacQueen notes that one of the drawbacks of Petri nets is the lack of a direct way of describing the flow of *data* through a net. He concludes that Petri nets "seem most useful for fundamental studies of the semantics of parallelism which relate only indirectly to the issues of programming" [164, p6].

D.3 Dataflow models

Dataflow models are a special case of stream processing systems [226]. These systems consist of a collection of *filters* that execute in parallel, communicating via *channels*. Stephens classifies dataflow systems according to three dimensions [226, §3]:

- whether the filters are synchronous or asynchronous with respect to each other;
- whether the filters are deterministic (functions) or non-deterministic; and
- whether the channels are uni- or bi-directional.

The filters and channels model of dataflow lends itself naturally to a graph-based representation. Dataflow systems can also be described using a functional programming language; however, difficulties arise when describing a filter with multiple outputs [164, p29].

The operational semantics of a dataflow system determine whether it is evaluated in an eager or a lazy fashion [226, §3.3]. Eager execution refers to a data-driven approach whereby filters send output on their egress channels without regard to whether the successor filter is in a waiting state. Lazy execution refers to a demand-driven approach whereby filters demand input from predecessors when they have finished processing the last inputs.

D.3.1 Kahn's dataflow networks

The most influential contribution to dataflow computation was made by Gilles Kahn. He defined a programming language and a denotational semantics for his class of dataflow

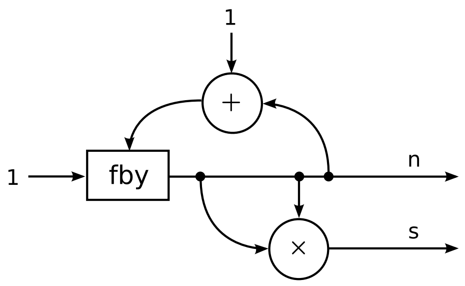


Figure D.3: A dataflow network to output a stream of natural numbers, n, and a stream of perfect squares, s.

networks [164, §6]. The programming language was a simple, ALGOL-like parallel language for representing dataflow networks. The model is of dedicated channels with storage between filters, and computation is demand-driven. Channels in Kahn's model thus represent unbounded queues of data elements, which yields pipelining between filters.

The denotational semantics is based upon functions on streams (called 'histories'). Denotationally, streams can be thought of as similar to lists but without a test for the null list; this test is not possible since it is undecidable whether a predecessor filter will output another value. Loops in the network imply recursion in the denotation.

D.3.2 Other languages

An early approach to describe dataflow systems in a high-level, functional style was implemented in the language SISAL (Streams and Iteration in a Single-Assignment Language) [89]. At a glance, SISAL had the appearance of contemporary conventional imperative languages such as Pascal, but restricted each name to bind to at most one single value, giving rise to its functional, side-effect free nature. SISAL compilers automatically parallelise the program, outputting a dataflow graph.

Perhaps the most famous dataflow language is Lucid [240]. Lucid adopts a demand-driven model of evaluation. A Lucid program can be thought of as a system of recursive equations, each of which defines a network of filters and their communication channels. Variables in the equations represent streams of values; streams can be composed using the 'followed-by' operator, fby. For example,

$$\begin{aligned} n &= 1 \,\mathsf{fby}\, n + 1 \\ s &= n \times n \end{aligned}$$

defines a simple network which outputs a stream of the natural numbers called n and a stream of the perfect squares called s, depicted graphically in Figure D.3.

Lustre is a dataflow language designed for specifying 'reactive' systems which continually process input data [97]. In contrast to Lucid's asynchronous nature, Lustre is synchronous: each event of a program takes place at a known time. This results in temporally-deterministic programs, so it has found uses in many real-time systems.

Thies et al. observed that these kinds of language have not been widely adopted by practical implementations of applications involving streaming media; this has led to the design of the high-level language StreamIt [230]. The aim of this language is to provide a high-level stream processing abstraction whilst still maintaining the flexibility of a general-purpose programming language. A StreamIt program consists of a hierarchical composition of stream processing elements and constructs which enable serialisation, parallelisation and feedback loops. The use of a hierarchical graph of stream processing elements, rather than an arbitrary graph, means that certain compiler analyses and optimisations are possible.

D.4 Categorical datatypes

Categorical datatypes are a generalisation of abstract datatypes, with the addition of parallel operations [221, §2.5]. The programming model of using categorical datatypes is a data-parallel model which is architecture-independent and conceptually simple to reason about because it is single-threaded. Moreover, the computation and communication requirements are known when the datatype is built. Parallelism arises through operations on the structure of the datatype. Categorical datatypes have been used to develop a parallel programming model for lists, bags, trees, arrays, molecules and graphs.

D.4.1 Bird-Meertens Formalism

A Bird-Meertens theory begins with base types and extends them to new data-parallel types using type functors [219, §2]. Given the constructors for a type, a large set of identities are automatically generated. This is a significant aid to program development: programs which are clearly correct can be written in an equational style, and a compiler can use the identities to transform it into one which is more computationally efficient. Not only is the set of such identities large, it is also guaranteed to be complete. Hence, any two equivalent computations can be derived from each other.

Crucially for parallelism, these data-parallel types have a generalised 'map' operation which enables a single operation to be applied to all elements of the type in parallel. Furthermore, for some types, a generalised 'reduce' operation is available, which recursively collapses a structured value into a single element, with the possibility of further parallelism.

As an example, consider the type functor which takes any type and maps it to lists of that type. We can then *lift* any function f from type τ_1 to type τ_2 to a function map f which maps lists of τ_1 to lists of τ_2 by applying f to each element. Functionals map and red are defined as follows:

$$\begin{aligned}\mathrm{map}\, f[x_1,\ldots,x_n] &= [f(x_1),\ldots,f(x_n)];\\ \mathrm{red}(\star)[a] &= a\\ \mathrm{red}(\star)[x \mathbin{+\!\!+} y] &= (\mathrm{red}(\star)x)\star(\mathrm{red}(\star)y),\end{aligned}$$

where $+\!\!+$ is list concatenation.

Homomorphisms on types can then be expressed as the composition of a generalised map and a generalised reduction. This means that any homomorphism can be expressed as a two-stage, highly parallel operation. For lists in particular, Bird proved that all homomorphisms on lists can be decomposed into a map stage and a reduce stage. Many important functions can be expressed as list homomorphisms, including sorting [219, §2].

Formally, Bird's theorem can be expressed as follows for lists:

Theorem D.1 (Bird's First Homomorphism Theorem [28])**.** *A function h is a homomorphism from a monoid of lists* $(\mathrm{list}(S), +\!\!+, [])$ *to* $(T, \otimes, i)$ *iff*

$$h = \mathrm{red}(\otimes) \circ \mathrm{map}(f), \tag{D.1}$$

where $f(a) = h([a])$*, and this decomposition is unique.*

The map stage can be totally parallelised, and the reduce stage can be parallelised in a tree-like structure [92, p402]. If the operation $\otimes$ has constant time complexity then the parallel algorithm for the homomorphism will have complexity $\mathrm{O}(\log n)$ where n is the length of the list.

For distributed inputs, where we have a list of lists rather than just a single list as input, a consequence of Bird's theorem called the *promotion law* is applicable [91, p151]:

$$h \circ \mathrm{red}(+\!\!+) = \mathrm{red}(\otimes) \circ \mathrm{map}(h). \tag{D.2}$$

D.4.2 Stages and Transformations paradigm

The 'stages and transformations' program design paradigm proposed by Gorlatch involves the initial expression of a program as a sequential composition of *stages* [91]. Each stage may consist of work done in parallel. This program is then transformed into a parallel composition of sequential work using semantics-preserving transformations. It is suggested by Gorlatch that this approach is better than initially starting with the program in the parallel form because the sequential form is far easier to design, understand and reason about.

APPENDIX E

Inter-process communication techniques

This appendix describes some practical implementations of inter-process communication techniques for loosely-coupled distributed systems.

E.1 Distributed shared virtual memory

A distinction is made between page-based and fine-grained distributed shared virtual memory (DSVM). Page-based systems allocate each page of memory to a processor and transfer the contents of the page to a processor at which a page fault occurs because it is not present. This approach suffers from the potential problem of *false sharing*, when two processes concurrently write to unrelated items of data which are located in the same page, causing the page to bounce between the processes, incurring a large network cost. A fine-grained approach separates each item of data and thus does not suffer from the false sharing problem because the unit of coherence is smaller. However, it means that checks for the local presence of a page must be made in software rather than relying on a hardware memory-management unit.

Two early implementations of page-based DSVM at the operating system level are found in the PLUS system [29] and Dash multi-processor machine [147]. More recently, TreadMarks is a user-level implementation of page-based DSVM on loosely-coupled networks of commodity computers [7]. Java/DSM is an implementation of a Java virtual machine on top of TreadMarks [256].

Shasta is a fine-grained implementation of DSVM [213]. It rewrites executables, intercepting loads and stores to insert checks whether data is available locally, communicating with other processors if it is not. Jackal is similarly fine-grained, but works on Java bytecode [236]. It also analyses Java source code, if available, to reduce the number of access checks necessary.

High-Performance Fortran is a set of extensions to Fortran 90 which adds constructs for explicit parallelism, such as *forall* which identifies loops which can be processed in parallel [139]. Again, the programming model is of a shared, global memory whilst data is distributed throughout separate physical memories. Unlike in the languages above, the process which owns an item of data does not change, but the compiler inserts message-passing logic to share data. Programmers use *align* and *distribute* directives to advise the compiler of which processors to map items of data to.

In a cluster-computing environment, the MOSIX virtualisation layer on top of Linux can create the appearance of a single operating system which runs across multiple machines, referred to as a 'single system image' (SSI) [20]. In this environment, programs are executed in the same fashion as on a uni-processor machine, with shared-memory assumptions, and the virtualisation layer automatically and transparently performs workload distribution via process migration.

E.2 Shared objects

The C Region Library (CRL) [126] aims to avoid the false sharing problem by allowing programmers to annotate their programs. Programmers define shared-data *regions*, which are arbitrarily-sized contiguous areas of memory, and surround reads and writes with access-check calls.

Orca is a programming language which supports shared data objects to allow processes to communicate [18]. Unlike DSVM, where access to shared memory is low-level, through reads to and writes from memory locations, Orca allows shared data to be manipulated through user-defined, high-level, composite operations. An example of a shared data structure might be a FIFO queue, with high-level operations *push*, *pop* and *peek*. The implementation of Orca distributes replicas of shared objects between processes. This permits read operations to proceed concurrently and with minimal cost; however, write operations entail a broadcast of the new value to all replicas.

Jade is a programming language based on C which adds constructs which allow the programmer to annotate how shared objects are accessed [207]. Programs are written in a serial fashion, and the Jade compiler then identifies which parts of the program can be executed in parallel whilst preserving the program's semantics.

The Parallel Virtual Machine (PVM) library, which has bindings to several languages, is a programming environment for loosely-coupled distributed systems [227]. It incorporates primitives for various parallel-processing paradigms, including the sharing of memory regions. This is achieved through the ability to share a contiguous region of memory containing either a series of integers or floating point values or being treated in an untyped fashion [227, §2.3].

Mentat is a language built on top of C++ which adds a distributed object space [93]. Objects on which parallel computation can be performed are distinguished from sequential objects by the programmer.

APPENDIX F

Task partitioning

This appendix describes extant work pertaining to the field of *task partitioning*. This is the problem of determining which program components should be grouped together into a logical unit to execute on a single processor. Task partitioning is a pre-requisite for task assignment (see Section 2.2.3) which is the procedure of determining which processor is to execute each group of components.

F.1 Manual task partitioning

There are several systems in which task partitioning is performed by the programmer and is fed to a compiler in a configuration file. This usually goes hand-in-hand with manual task assignment, where the configuration file will not only define which tasks constitute each partition, but also which processor will execute those tasks.

An early example is CAGES, which was an environment for programming graphics applications which use two processors [98]. Programmers would write an ordinary program, as if for a uniprocessor machine. This is then sent to a pre-processing compiler along with a separate specification of which variables are global, and of which procedures should be assigned to which processor. The pre-processor would then generate individual executables for each machine, using RPC for procedure calls spanning processors.

A more recent example concerns programming network processors. It is common for network processors to be arranged such that not all memory banks are available from each processor. Hence it is necessary for the processors which execute each part of an application, and the memory banks which hold each item of data, to be carefully selected, to ensure that data is always available to processors which need it. Presently, this selection of processors is usually achieved through a programmer manually deciding which instructions to execute on which processors, preparing a separate binary for each core. For example, the language PacLang is a high-level, imperative, architecturally-neutral programming language for programming network processors [66].[1] The compiler accepts a PacLang program along with an architecture mapping script (AMS) [67]. The AMS is architecture-specific, and specifies how the high-level program should be partitioned into a set of concurrent tasks suitable for execution on that architecture, and how these tasks are mapped to the available processors.

In a sensor network, Kumar et al. discuss task assignment in a hierarchical network [143]. These networks involve two or more tiers of processors, each with a characteristic processing ability which increases up the hierarchy. They state that the decision about where to partition the application, and which processors to employ, impacts on the application's performance and energy consumption.

F.2 Static automatic task partitioning

On the other hand, various work in automatic task partitioning has been undertaken. Invariably, automatic task partitioning involves the use of profiling—measuring the duration of time spent in each procedure and the sizes of the arguments and return values—in order to gather statistical information about the tasks and the interactions between them. This requires the program to be executed whilst augmented with instrumentation code. This is superior to a static, off-line approach which can only estimate information about which tasks communicate with which other tasks, and not deduce how costly this communication is or how frequently it occurs.

Early work in automatic task partitioning was performed by the Interconnected Processor System (ICOPS) [175]. The motivation for that work was an interactive graphics application which was to be distributed between a host processor (where the application's data was located) and a satellite graphics processor (where the display was located). Metrics gathered through profiling made it clear which of the program's seven modules should be executed on which processor. A graph-cutting algorithm was used to automatically divide the program's modules into two portions, to run on the two processors.

The Coign system performs task partitioning on an application written using Microsoft's Component Object Model (COM) [120]. From the results of profiling, an inter-component communication graph is derived, with weighted edges indicating the volume of data flow between components. Again, a graph-cutting algorithm is used to divide this graph into two segments—a client portion and server portion—minimising the edge costs across the cuts.

The graph-cutting algorithms employed above could be generalised to divide a program into more than two partitions. However, this is known to be NP-complete. Hogstedt et

[1]PacLang exploits a linear type system which entails each packet on the heap being referenced by precisely one thread at all times. However, packets may be referenced multiple times within a thread. This permits a range of compiler optimisations.

al. describe a number of heuristic approaches to solving the n-way graph-cutting problem in polynomial time [113].

An approach to task partitioning for programs expressed in the SISAL single-assignment dataflow language[2] was presented by Sarkar and Hennessy [211]. They use a recursive-subdivision approach. Initially, the entire program graph is considered to be a single task. Then it is broken down into smaller tasks constituted of sub-graphs. This is then repeated recursively until no bottleneck tasks remain. A task is a bottleneck when its execution would cost more than all of the tasks which can be executed in parallel in the meantime. The costs are derived from static analysis and (optional) hints from the programmer.

Automatic task partitioning has found many applications in the area of 'computation off-loading', also referred to as 'remote execution'. These terms are applied primarily in pervasive or ubiquitous computing, referring to the relieving of the burden of intensive computation from resource-constrained devices (often mobile devices) and its off-loading to a surrogate processor (often fixed computers). Computation off-loading at the process level had shown much promise: Rudenko et al. show that for computationally-intensive applications, the energy savings through remote execution can outweigh the energy losses incurred through transmitting data to and from the remote processor [209]. Task partitioning explores whether savings can be made at a finer granularity than off-loading the execution of entire processes.

Ou et al. describe a scheme [195] to partition a pervasive application expressed in Java bytecode into $k + 1$ parts, where one is a partition which cannot be offloaded because it involves interaction with a user, and the other k partitions can be offloaded to surrogate processors. The profiling scheme they propose generates a weighted graph of program components which indicates not only communication costs on edges but also CPU and memory utilisation on vertices.

Li et al. present a similar approach to a solution to the same problem [154]. They determine function execution times by profiling, and partition the program at the function level using a heuristic-based technique. Their experimental tests with a PDA show that, for a variety of applications, wirelessly off-loading the computation to a more powerful computer often leads to lower energy consumption than turning the (expensive) wireless interface off and performing the computation locally.

J-Orchestra [156] is an automatic partitioning system for ubiquitous computing applications. It transforms a Java program at the bytecode level into a partitioned version by re-writing the code, converting local method calls to remote calls as appropriate. However, the analysis driving the partitioning is static, revealing only the existence of dependencies between classes and not the quantity of communication therein. Through a GUI, a user can edit the partitioning at will.

MagnetOS provides an automatic partitioning service which takes a program written for a single Java virtual machine [22]. It partitions the program into a set of components, performing bytecode-level rewriting to modify object creation, method invocation and field access. Java RMI is employed to provide remote invocation.

[2]SISAL is introduced in Section D.3.2.

F.3 Dynamic automatic task partitioning

The approach described by Gu et al. for pervasive computing environments involves making decisions about off-loading computation during run-time [95]. Assuming an object-oriented programming model, their run-time system forms a graph of classes and the interactions between them. The system analyses the total amount of memory which instances of each class consume to form the node weights, and the amount of data transferred between classes to form the edge weights. Programs start off entirely located on a single mobile device. As the program runs, the system monitors memory usage and contemplates an off-load if memory becomes tight.

APPENDIX G

Example task graphs

In this appendix, we present some examples of simple task graphs. The following examples are common and well-known aggregation functions which either produce a summary of a set of input values or return an exemplar value. The sum, count, count-unique and mean aggregation operations produce a summary of the input; the maximisation, minimisation and median operations return an exemplar value [165, p135].

The task graph of the median operator was described in Section 5.5.5.1, and that of the arithmetic mean operator was described in Section 5.5.7.1.

G.1 Sum

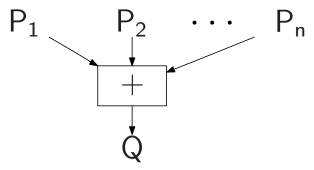


Figure G.1: Summing the input values.

Analogous to the computation performed by the SQL `sum` aggregation operator, the task graph in Figure G.1 sums the input values. One n-ary merge task is employed. The merge function is addition. We use the terms P_1, P_2 and P_n to denote sub-graphs with a single output, and Q to denote a sub-graph with a single input.

G.2 Maximisation and minimisation

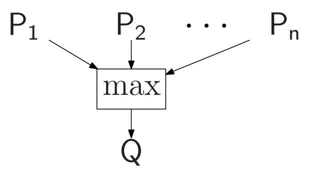


Figure G.2: Finding the maximal input value.

Analogous to the computation performed by the SQL `max` aggregation operator, the task graph in Figure G.2 finds the maximal input value. A similar graph can be constructed to find the minimal input value. In each case, one n-ary merge task is employed, where the merge function is either max or min.

G.3 Count

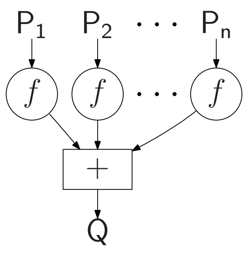


Figure G.3: Counting the number of input values.

Analogous to the computation performed by the SQL `count` aggregation operator, the task graph in Figure G.3 counts the number of input values. For n input values, n processing tasks and one n-ary merge task are employed. The merge function is addition and the processing function is defined as

$$f(x) = 1.$$

G.4 Count of unique values

Analogous to the computation performed by the SQL `count distinct` aggregation operator, the task graph in Figure G.4 counts the number of unique input values. For n input values, $n + 1$ processing tasks and one n-ary merge task are employed.

The processing function f creates a unit list, defined as

$$f(x) \quad = \quad [x].$$

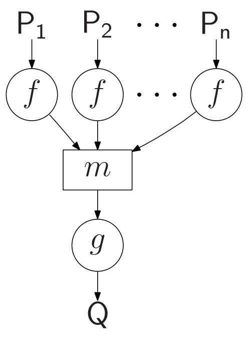


Figure G.4: Counting the number of unique input values.

The merge function m takes two sorted lists of values and combines them into a single sorted list whilst discarding duplicates, defined as

$$\begin{aligned}
m(as, []) &= as \\
m([], bs) &= bs \\
m(a \mathbin{+\!\!+} as, b \mathbin{+\!\!+} bs) &= \begin{cases} a \mathbin{+\!\!+} m(as, bs) & \text{if } a = b \\ a \mathbin{+\!\!+} m(as, b \mathbin{+\!\!+} bs) & \text{if } a < b \\ b \mathbin{+\!\!+} m(a \mathbin{+\!\!+} as, bs) & \text{otherwise.} \end{cases}
\end{aligned}$$

Finally, function g finds the length of a list, defined as

$$\begin{aligned}
g([]) &= 0 \\
g(x \mathbin{+\!\!+} xs) &= 1 + g(xs).
\end{aligned}$$

APPENDIX H

Graphical derivations of ternary transformations

In this appendix, we graphically show how the ternary versions of the program transformations can be expressed in terms of applications of the binary transformations of Section 5.5.7.

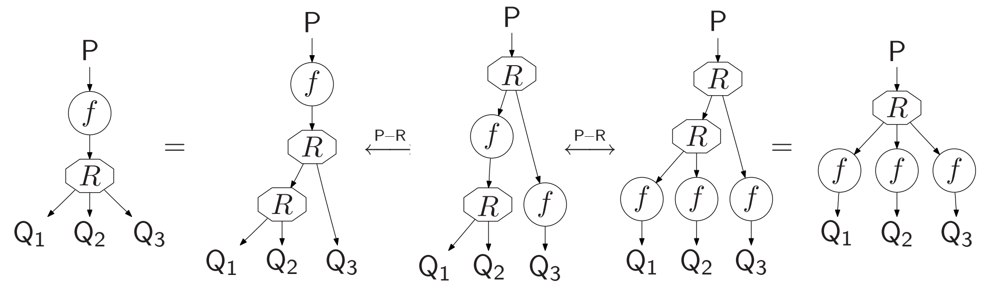


Figure H.1: Derivation of the ternary Processing–Replication transformation.

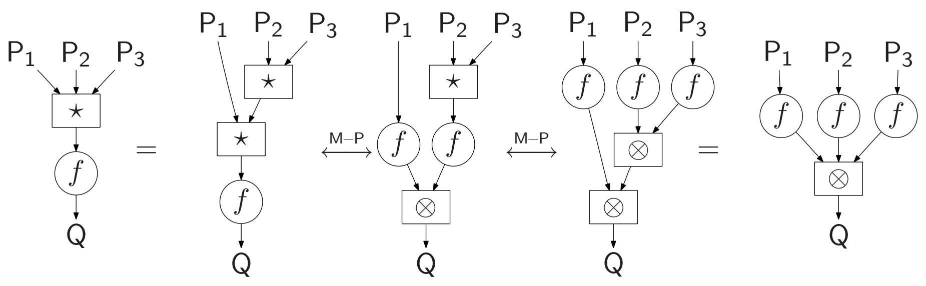


Figure H.2: Derivation of the ternary Merge–Processing transformation.

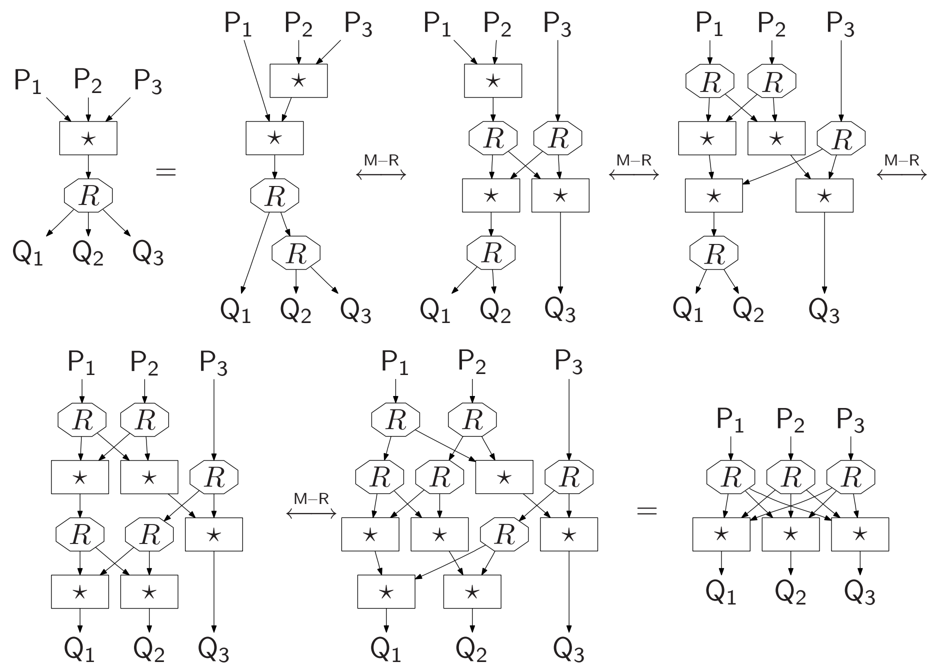


Figure H.3: Derivation of the ternary Merge–Replication transformation.

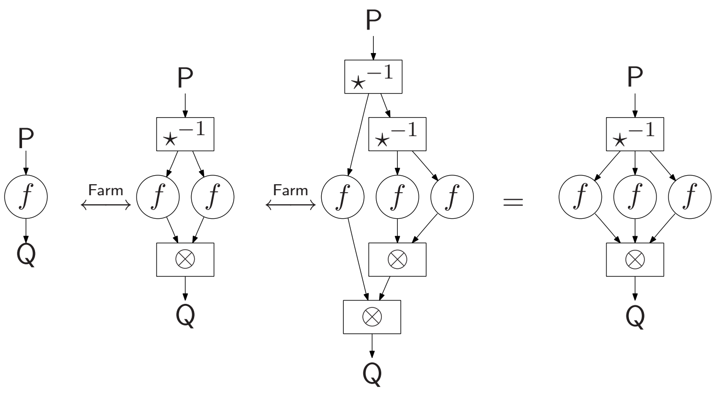


Figure H.4: Derivation of the ternary Farm transformation.

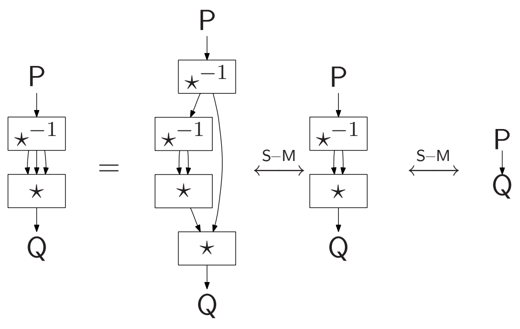


Figure H.5: Derivation of the ternary Split–Merge transformation.

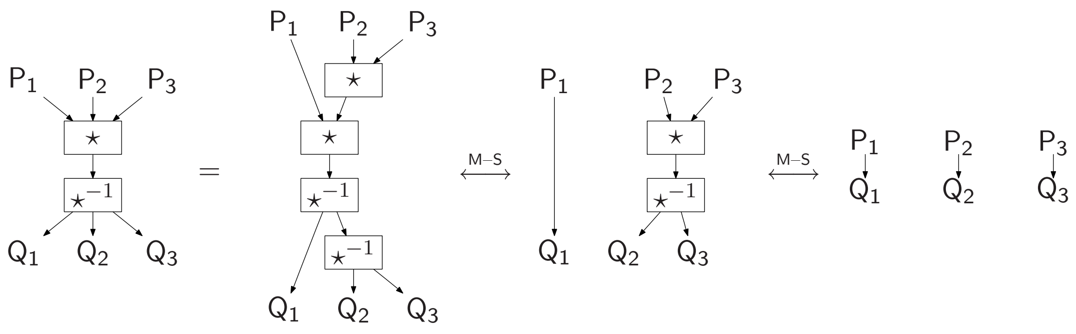


Figure H.6: Derivation of the ternary Merge–Split transformation.

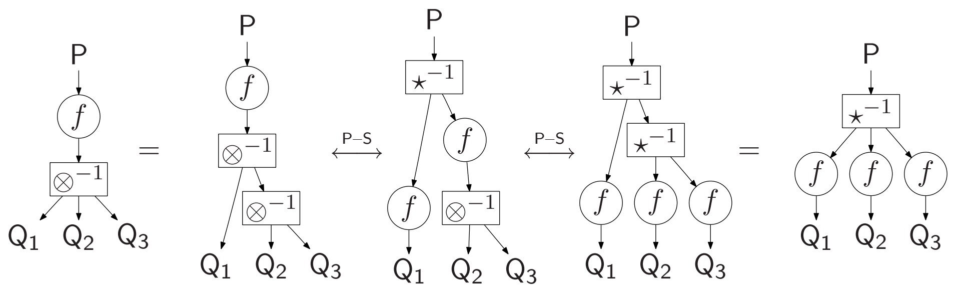


Figure H.7: Derivation of the ternary Processing–Split transformation.

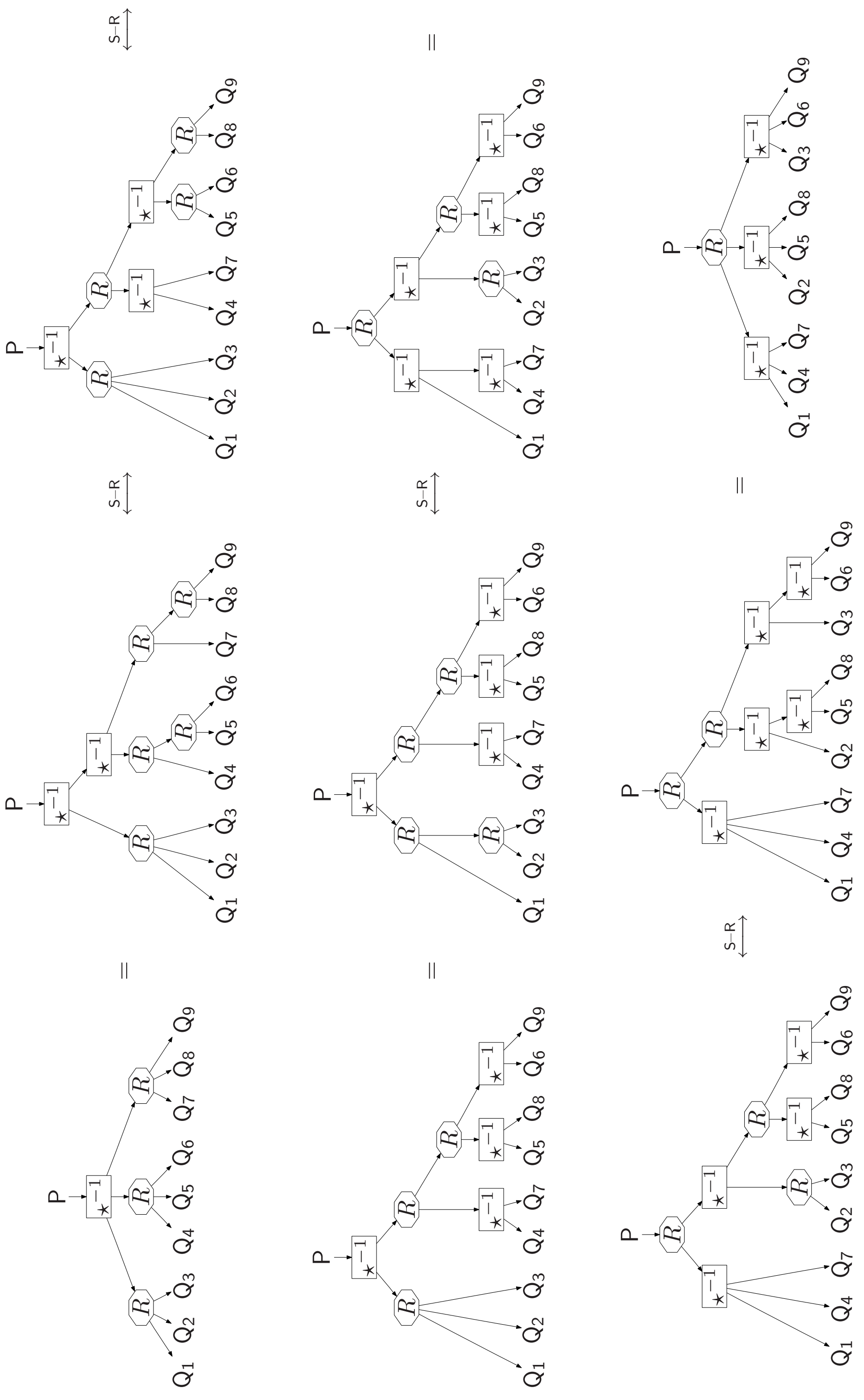


Figure H.8: Derivation of the ternary Split–Replication transformation.

APPENDIX I

Proofs of soundness of n-ary transformations

This appendix contains a collection of proofs of soundness for n-ary task graph transformations, justifying them in terms of the binary transformations of Section 5.5.7.

I.1 Merge–Processing transformation

The denotations of the task graphs in the n-ary version of the Merge–Processing transformation are:

$$\begin{aligned}
\mathsf{Left}_n^{\mathsf{M\text{–}P}}(x_1,\ldots,x_n,y) &\triangleq \exists t.\ \mathsf{Merge}_n(\star)(x_1,\ldots,x_n,t) \wedge \mathsf{Proc}(f)(t,y) \\
\mathsf{Right}_n^{\mathsf{M\text{–}P}}(x_1,\ldots,x_n,y) &\triangleq \exists t_1,\ldots t_n.\ \mathsf{Merge}_n(\otimes)(t_1,\ldots,t_n,y) \wedge \\
&\qquad \textstyle\bigwedge_{i=1}^{n} \mathsf{Proc}(f)(x_i,t_i)
\end{aligned}$$

Proposition I.1. *The n-ary version of the Merge–Processing transformation is sound.*

Proof. We define the property

$$\Phi(n) \triangleq (\mathsf{Left}_n^{\mathsf{M\text{–}P}}(x_1,\ldots,x_n,y) \equiv \mathsf{Right}_n^{\mathsf{M\text{–}P}}(x_1,\ldots,x_n,y))$$

and show that $\forall n \in \mathbb{N}.\ n > 1 \Rightarrow \Phi(n)$ by induction on n. The base case, $\Phi(2)$, was proven

earlier in (5.17). The induction hypothesis is $\Phi(k) \Rightarrow \Phi(k+1)$.

$$\begin{array}{lll}
& \mathsf{Left}^{\mathsf{M\text{–}P}}_{k+1}(x_1,\ldots,x_{k+1},y) & \\
\equiv & \exists t.\ \mathsf{Merge}_{k+1}(\star)(x_1,\ldots,x_{k+1},t) \wedge \mathsf{Proc}(f)(t,y) & \\
\equiv & \exists t.\ (\exists u.\ \mathsf{Merge}_k(\star)(x_1,\ldots x_k,u) \wedge & \text{by (5.9)} \\
& \quad \mathsf{Merge}(\star)(u,x_{k+1},t)) \wedge \mathsf{Proc}(f)(t,y) & \\
\equiv & \exists u.\ \mathsf{Merge}_k(\star)(x_1,\ldots,x_k,u) \wedge & \\
& \quad (\exists t.\ \mathsf{Merge}(\star)(u,x_{k+1},t) \wedge \mathsf{Proc}(f)(t,y)) & \\
\equiv & \exists u.\ \mathsf{Merge}_k(\star)(x_1,\ldots,x_k,u) \wedge & \text{by (5.17)} \\
& \quad (\exists b,c.\ \mathsf{Proc}(f)(u,b) \wedge \mathsf{Proc}(f)(x_{k+1},c) \wedge & \\
& \qquad \mathsf{Merge}(\otimes)(b,c,y)) & \\
\equiv & \exists b,c.\ (\exists u.\ \mathsf{Merge}_k(\star)(x_1,\ldots,x_k,u) \wedge \mathsf{Proc}(f)(u,b)) \wedge & \\
& \quad \mathsf{Proc}(f)(x_{k+1},c) \wedge \mathsf{Merge}(\otimes)(b,c,y) & \\
\equiv & \exists b,c.\ (\exists t_1,\ldots t_k.\ \mathsf{Merge}_k(\otimes)(t_1,\ldots,t_k,b) \wedge & \text{by } \Phi(k) \\
& \quad \bigwedge_{i=1}^{k} \mathsf{Proc}(f)(x_i,t_i)) \wedge & \\
& \quad \mathsf{Proc}(f)(x_{k+1},c) \wedge \mathsf{Merge}(\otimes)(b,c,y) & \\
\equiv & \exists t_1,\ldots t_k,c.\ (\exists b.\ \mathsf{Merge}_k(\otimes)(t_1,\ldots,t_k,b) \wedge & \\
& \qquad \mathsf{Merge}(\otimes)(b,c,y)) \wedge & \\
& \quad \mathsf{Proc}(f)(x_{k+1},c) \wedge \bigwedge_{i=1}^{k} \mathsf{Proc}(f)(x_i,t_i) & \\
\equiv & \exists t_1,\ldots t_k,c.\ \mathsf{Merge}(\otimes)(t_1,\ldots,t_k,c,y) \wedge & \text{by (5.9)} \\
& \quad \mathsf{Proc}(f)(x_{k+1},c) \wedge \bigwedge_{i=1}^{k} \mathsf{Proc}(f)(x_i,t_i) & \\
\equiv & \exists t_1,\ldots t_{k+1}.\ \mathsf{Merge}(\otimes)(t_1,\ldots,t_{k+1},y) \wedge \bigwedge_{i=1}^{k+1} \mathsf{Proc}(f)(x_i,t_i) & \\
\equiv & \mathsf{Right}^{\mathsf{M\text{–}P}}_{k+1}(x_1,\ldots,x_{k+1},y). &
\end{array}$$

Hence $\Phi(n)$ holds for all $n > 1$. □

I.2 Farm transformation

The denotations of the task graphs in the n-ary version of the Farm transformation are:

$$\begin{array}{rcl}
\mathsf{Left}^{\mathsf{Farm}}_n(x,y) & \triangleq & \mathsf{Proc}(f)(x,y) \\
\mathsf{Right}^{\mathsf{Farm}}_n(x,y) & \triangleq & \exists x_1,\ldots x_n,y_1,\ldots,y_n.\ \mathsf{Split}_n(\star^{-1})(x,x_1,\ldots,x_n) \wedge \\
& & \quad \bigwedge_{i=1}^{n} \mathsf{Proc}(f)(x_i,y_i) \wedge \\
& & \quad \mathsf{Merge}(\otimes)(y_1,\ldots y_n,y)
\end{array}$$

Proposition I.2. *The n-ary version of the Farm transformation is sound.*

Proof. We define the property

$$\Phi(n) \triangleq (\mathsf{Left}^{\mathsf{Farm}}_n(x,y) \equiv \mathsf{Right}^{\mathsf{Farm}}_n(x,y))$$

and show that $\forall n \in \mathbb{N}.\ n > 1 \Rightarrow \Phi(n)$ by induction on n. The base case, $\Phi(2)$, was proven

earlier in (5.18). The induction hypothesis is $\Phi(k) \Rightarrow \Phi(k+1)$.

$$\begin{array}{lll}
 & \mathsf{Left}^{\mathsf{Farm}}_{k+1}(x,y) & \\
\equiv & \mathsf{Proc}(f)(x,y) & \\
\equiv & \exists x_1,\dots x_k, y_1,\dots y_k.\ \mathsf{Split}_k(\star^{-1})(x,x_1,\dots,x_k) \wedge & \text{by } \Phi(k)\\
 & \qquad \bigwedge_{i=1}^{k} \mathsf{Proc}(f)(x_i,y_i) \wedge & \\
 & \qquad \mathsf{Merge}_k(\star)(y_1,\dots,y_k,y) & \\
\equiv & \exists x_1,\dots x_k, y_1,\dots y_k.\ \mathsf{Split}_k(\star^{-1})(x,x_1,\dots,x_k) \wedge & \text{by (5.18)}\\
 & \qquad \bigwedge_{i=1}^{k-1} \mathsf{Proc}(f)(x_i,y_i) \wedge & \\
 & \qquad (\exists t_1,t_2,u_1,u_2.\ \mathsf{Split}(\star^{-1})(x_k,t_1,t_2) \wedge & \\
 & \qquad\qquad \mathsf{Proc}(f)(t_1,u_1) \wedge & \\
 & \qquad\qquad \mathsf{Proc}(f)(t_2,u_2) \wedge & \\
 & \qquad\qquad \mathsf{Merge}(\otimes)(u_1,u_2,y_k)) \wedge & \\
 & \qquad \mathsf{Merge}_k(\star)(y_1,\dots,y_k,y) & \\
\equiv & \exists x_1,\dots,x_{k-1},t_1,t_2,y_1,\dots,y_{k-1},u_1,u_2. & \\
 & \quad (\exists x_k.\ \mathsf{Split}_k(\star^{-1})(x,x_1,\dots,x_k) \wedge \mathsf{Split}(\star^{-1})(x_k,t_1,t_2)) \wedge & \\
 & \quad (\exists y_k.\ \mathsf{Merge}_k(\star)(y_1,\dots,y_k,y) \wedge \mathsf{Merge}(\otimes)(u_1,u_2,y_k)) \wedge & \\
 & \quad \bigwedge_{i=1}^{k-1} \mathsf{Proc}(f)(x_i,y_i) & \\
\equiv & \exists x_1,\dots,x_{k-1},t_1,t_2,y_1,\dots,y_{k-1},u_1,u_2. & \text{by (5.9), (5.10)}\\
 & \quad \mathsf{Split}_{k+1}(\star^{-1})(x,x_1,\dots,x_{k-1},t_1,t_2) \wedge & \\
 & \quad \mathsf{Merge}_{k+1}(\star)(y_1,\dots,y_{k-1},u_1,u_2,y) \wedge & \\
 & \quad \bigwedge_{i=1}^{k-1} \mathsf{Proc}(f)(x_i,y_i) & \\
\equiv & \exists x_1,\dots,x_{k+1},y_1,\dots,y_{k+1}.\ \mathsf{Split}_{k+1}(\star^{-1})(x,x_1,\dots,x_{k+1}) \wedge & \\
 & \qquad \bigwedge_{i=1}^{k+1} \mathsf{Proc}(f)(x_i,y_i) \wedge & \\
 & \qquad \mathsf{Merge}_{k+1}(\star)(y_1,\dots,y_{k+1},y) & \\
\equiv & \mathsf{Right}^{\mathsf{Farm}}_{k+1}(x,y). &
\end{array}$$

Hence $\Phi(n)$ holds for all $n > 1$. □

I.3 Processing–Replication transformation

The denotations of the task graphs in the n-ary version of the Processing–Replication transformation are:

$$\begin{array}{rcl}
\mathsf{Left}^{\mathsf{P\text{–}R}}_n(x,y_1,\dots,y_n) & \triangleq & \exists t.\ \mathsf{Proc}(f)(x,t) \wedge \mathsf{Rep}_n(t,y_1,\dots,y_n)\\
\mathsf{Right}^{\mathsf{P\text{–}R}}_n(x,y_1,\dots,y_n) & \triangleq & \exists t_1,\dots,t_n.\ \mathsf{Rep}_n(x,t_1,\dots,t_n) \wedge \bigwedge_{i=1}^{n} \mathsf{Proc}(f)(t_i,y_i)
\end{array}$$

Proposition I.3. *The n-ary version of the Processing–Replication transformation is sound.*

Proof. We define the property

$$\Phi(n) \triangleq (\mathsf{Left}^{\mathsf{P\text{–}R}}_n(x,y_1,\dots,y_n) \equiv \mathsf{Right}^{\mathsf{P\text{–}R}}_n(x,y_1,\dots,y_n))$$

and show that $\forall n \in \mathbb{N}.\ n > 1 \Rightarrow \Phi(n)$ by induction on n. The base case, $\Phi(2)$, was proven

earlier in (5.19). The induction hypothesis is $\Phi(k) \Rightarrow \Phi(k+1)$.

$$\begin{array}{lll}
& \mathsf{Left}^{\mathsf{P\text{–}R}}_{k+1}(x, y_1, \ldots, y_n) & \\
\equiv & \exists t.\ \mathsf{Proc}(f)(x,t) \wedge \mathsf{Rep}_{k+1}(t, y_1, \ldots, y_{k+1}) & \\
\equiv & \exists t.\ \mathsf{Proc}(f)(x,t) \wedge (\exists u.\ \mathsf{Rep}_k(u, y_1, \ldots, y_k) \wedge \mathsf{Rep}(t, u, y_{k+1})) & \text{by (5.11)} \\
\equiv & \exists u.\ \mathsf{Rep}_k(u, y_1, \ldots, y_k) \wedge (\exists t.\ \mathsf{Proc}(f)(x,t) \wedge \mathsf{Rep}(t, u, y_{k+1})) & \\
\equiv & \exists u.\ \mathsf{Rep}_k(u, y_1, \ldots, y_k) \wedge & \text{by (5.19)} \\
& \quad (\exists t_1, t_2.\ \mathsf{Rep}(x, t_1, t_2) \wedge \mathsf{Proc}(f)(t_1, u) \wedge \mathsf{Proc}(f)(t_2, y_{k+1})) & \\
\equiv & \exists t_1, t_2.\ \mathsf{Rep}(x, t_1, t_2) \wedge \mathsf{Proc}(f)(t_2, y_{k+1}) \wedge & \\
& \quad (\exists u.\ \mathsf{Rep}_k(u, y_1, \ldots, y_k) \wedge \mathsf{Proc}(f)(t_1, u)) & \\
\equiv & \exists t_1, t_2.\ \mathsf{Rep}(x, t_1, t_2) \wedge \mathsf{Proc}(f)(t_2, y_{k+1}) \wedge & \text{by } \Phi(k) \\
& \quad (\exists u_1, \ldots, u_k.\ \mathsf{Rep}_k(t_1, u_1, \ldots, u_k) \wedge \bigwedge_{i=1}^{k} \mathsf{Proc}(f)(u_i, y_i)) & \\
\equiv & \exists u_1, \ldots, u_k, t_2.\ (\exists t_1.\ \mathsf{Rep}_k(t_1, u_1, \ldots u_k) \wedge \mathsf{Rep}(x, t_1, t_2)) \wedge & \\
& \quad \mathsf{Proc}(f)(t_2, y_{k+1}) \wedge \bigwedge_{i=1}^{k} \mathsf{Proc}(f)(u_i, y_i) & \\
\equiv & \exists u_1, \ldots, u_k, t_2.\ \mathsf{Rep}_{k+1}(t_1, u_1, \ldots, u_k, t_2) \wedge & \text{by (5.11)} \\
& \quad \mathsf{Proc}(f)(t_2, y_{k+1}) \wedge \bigwedge_{i=1}^{k} \mathsf{Proc}(f)(u_i, y_i) & \\
\equiv & \exists u_1, \ldots, u_{k+1}.\ \mathsf{Rep}_{k+1}(x, u_1, \ldots, u_{k+1}) \wedge \bigwedge_{i=1}^{k+1} \mathsf{Proc}(f)(u_i, y_i) & \\
\equiv & \mathsf{Right}^{\mathsf{P\text{–}R}}_{k+1}(x, y_1, \ldots, y_n). &
\end{array}$$

Hence $\Phi(n)$ holds for all $n > 1$. □

I.4 Split–Merge transformation

The denotations of the task graphs in the n-ary version of the Split–Merge transformation are:

$$\begin{array}{rcl}
\mathsf{Left}^{\mathsf{S\text{–}M}}_n(x, y) & \triangleq & \exists t_1, \ldots, t_n.\ \mathsf{Split}_n(\star^{-1})(x, t_1, \ldots, t_n) \wedge \mathsf{Merge}(\star)(t_1, \ldots, t_n, y) \\
\mathsf{Right}^{\mathsf{S\text{–}M}}_n(x, y) & \triangleq & y = x
\end{array}$$

Proposition I.4. *The n-ary version of the Split–Merge transformation is sound.*

Proof. We define the property

$$\Phi(n) \triangleq (\mathsf{Left}^{\mathsf{S\text{–}M}}_n(x, y) \equiv \mathsf{Right}^{\mathsf{S\text{–}M}}_n(x, y))$$

and show that $\forall n \in \mathbb{N}.\ n > 1 \Rightarrow \Phi(n)$ by induction on n. The base case, $\Phi(2)$, was proven

earlier in (5.24). The induction hypothesis is $\Phi(k) \Rightarrow \Phi(k+1)$.

$$\begin{array}{lll}
& \mathsf{Left}^{\mathsf{S-M}}_{k+1}(x,y) & \\
\equiv & \exists t_1,\ldots,t_{k+1}.\ \mathsf{Split}_{k+1}(\star^{-1})(x,t_1,\ldots,t_{k+1}) \wedge & \\
& \qquad \mathsf{Merge}_{k+1}(\star)(t_1,\ldots,t_{k+1},y) & \\
\equiv & \exists t_1,\ldots,t_{k+1}.\ (\exists u.\ \mathsf{Split}_k(\star^{-1})(u,t_1,\ldots,t_k) \wedge \mathsf{Split}(\star^{-1})(x,y,t_{k+1})) \wedge & \text{by } (5.10) \\
& \qquad \mathsf{Merge}_{k+1}(\star)(t_1,\ldots,t_{k+1},y) & \\
\equiv & \exists t_1,\ldots,t_{k+1}.\ (\exists u.\ \mathsf{Split}_k(\star^{-1})(u,t_1,\ldots,t_k) \wedge \mathsf{Split}(\star^{-1})(x,y,t_{k+1})) \wedge & \text{by } (5.9) \\
& \qquad (\exists v.\ \mathsf{Merge}_k(\star)(t_1,\ldots,t_k,v) \wedge \mathsf{Merge}(\star)(v,t_{k+1},y)) & \\
\equiv & \exists t_{k+1},u,v.\ \mathsf{Split}(\star^{-1})(x,u,t_{k+1}) \wedge \mathsf{Merge}(\star)(v,t_{k+1},y) \wedge & \\
& \qquad (\exists t_1,\ldots,t_k.\ \mathsf{Split}_k(\star^{-1})(u,t_1,\ldots,t_k) \wedge & \\
& \qquad\qquad \mathsf{Merge}_k(\star)(t_1,\ldots t_k,v)) & \\
\equiv & \exists t_{k+1},u,v.\ \mathsf{Split}(\star^{-1})(x,u,t_{k+1}) \wedge \mathsf{Merge}(\star)(v,t_{k+1},y) \wedge & \text{by } \Phi(k) \\
& \qquad v = u & \\
\equiv & \exists t_{k+1},t.\ \mathsf{Split}(\star^{-1})(x,t,t_{k+1}) \wedge \mathsf{Merge}(\star)(t,t_{k+1},y) & \\
\equiv & y = x & \text{by } (5.24) \\
\equiv & \mathsf{Right}^{\mathsf{S-M}}_{k+1}(x,y). &
\end{array}$$

Hence $\Phi(n)$ holds for all $n > 1$. □

I.5 Merge–Split transformation

The denotations of the task graphs in the n-ary version of the Merge–Split transformation are:

$$\begin{array}{rcl}
\mathsf{Left}^{\mathsf{M-S}}_n(x_1,\ldots,x_n,y_1,\ldots,y_n) & \triangleq & \exists t.\ \mathsf{Merge}_n(\star)(x_1,\ldots,x_n,t) \wedge \\
& & \quad \mathsf{Split}_n(\star^{-1})(t,y_1,\ldots,y_n) \\
\mathsf{Right}^{\mathsf{M-S}}_n(x_1,\ldots,x_n,y_1,\ldots,y_n) & \triangleq & \bigwedge_{i=1}^{n} y_i = x_i
\end{array}$$

Proposition I.5. *The n-ary version of the Merge–Split transformation is sound, provided that the task graph can be transformed to a state in which the outputs are combined in a merge stage.*

Proof. We define the property

$$\Phi(n) \triangleq (\mathsf{Left}^{\mathsf{M-S}}_n(x_1,\ldots,x_n,y_1,\ldots,y_n) \equiv \mathsf{Right}^{\mathsf{M-S}}_n(x_1,\ldots,x_n,y_1,\ldots,y_n))$$

and show that $\forall n \in \mathbb{N}.\ n > 1 \Rightarrow \Phi(n)$ by induction on n. The base case, $\Phi(2)$, was discussed in Section 5.5.7.11 under the condition that the outputs are merged. The induction hypothesis is $\Phi(k) \Rightarrow \Phi(k+1)$.

$$\begin{array}{lll}
& \mathsf{Left}^{\mathsf{M\text{–}S}}_{k+1}(x_1,\ldots,x_n,y_1,\ldots,y_n) & \\
\equiv & \exists t.\ \mathsf{Merge}_{k+1}(\star)(x_1,\ldots,x_{k+1},t) \wedge \mathsf{Split}_{k+1}(\star^{-1})(t,y_1,\ldots,y_{k+1}) & \\
\equiv & \exists t.\ (\exists u.\ \mathsf{Merge}_k(\star)(x_1,\ldots,x_k,u) \wedge \mathsf{Merge}(\star)(u,x_{k+1},t)) \wedge & \text{by } (5.9) \\
& \quad \mathsf{Split}_{k+1}(\star^{-1})(t,y_1,\ldots,y_{k+1}) & \\
\equiv & \exists t.\ (\exists u.\ \mathsf{Merge}_k(\star)(x_1,\ldots,x_k,u) \wedge \mathsf{Merge}(\star)(u,x_{k+1},t)) \wedge & \text{by } (5.10) \\
& \quad (\exists v.\ \mathsf{Split}_k(\star^{-1})(v,y_1,\ldots,y_k) \wedge \mathsf{Split}(\star^{-1})(t,v,y_{k+1})) & \\
\equiv & \exists u,v.\ \mathsf{Merge}_k(\star)(x_1,\ldots,x_k,u) \wedge \mathsf{Split}_k(\star^{-1})(v,y_1,\ldots,y_k) \wedge & \\
& \quad (\exists t.\ \mathsf{Merge}(\star)(u,x_{k+1},t) \wedge \mathsf{Split}(\star^{-1})(t,v,y_{k+1})) & \\
\equiv & \exists u,v.\ \mathsf{Merge}_k(\star)(x_1,\ldots,x_k,u) \wedge \mathsf{Split}_k(\star^{-1})(v,y_1,\ldots,y_k) \wedge & (\dagger) \\
& \quad v = u \wedge y_{k+1} = x_{k+1} & \\
\equiv & \exists t.\ \mathsf{Merge}_k(\star)(x_1,\ldots,x_k,t) \wedge \mathsf{Split}_k(\star^{-1})(t,y_1,\ldots,y_k)) \wedge & \\
& \quad y_{k+1} = x_{k+1} & \\
\equiv & \bigwedge_{i=1}^{k} y_i = x_i \wedge y_{k+1} = x_{k+1} & \text{by } \Phi(k) \\
\equiv & \bigwedge_{i=1}^{k+1} y_i = x_i & \\
\equiv & \mathsf{Right}^{\mathsf{M\text{–}S}}_{k+1}(x_1,\ldots,x_n,y_1,\ldots,y_n), &
\end{array}$$

where the step marked (†) is valid under the condition that the outputs are subsequently merged (proof omitted).

Hence $\Phi(n)$ holds for all $n > 1$. □

I.6 Processing–Split transformation

The denotations of the task graphs in the n-ary version of the Processing–Split transformation are:

$$\begin{array}{rcl}
\mathsf{Left}^{\mathsf{P\text{–}S}}_n(x,y_1,\ldots,y_n) & \triangleq & \exists t.\ \mathsf{Proc}(f)(x,t) \wedge \mathsf{Split}_n(\otimes^{-1})(t,y_1,\ldots,y_n) \\
\mathsf{Right}^{\mathsf{P\text{–}S}}_n(x,y_1,\ldots,y_n) & \triangleq & \exists t_1,\ldots,t_n.\ \mathsf{Split}_n(\star^{-1})(x,t_1,\ldots,t_n) \wedge \\
& & \qquad \bigwedge_{i=1}^{n} \mathsf{Proc}(f)(t_i,y_i)
\end{array}$$

Proposition I.6. *The n-ary version of the Processing–Split transformation is sound, provided that the task graph can be transformed to a state in which the outputs are combined in a merge stage.*

Proof. We define the property

$$\Phi(n) \triangleq (\mathsf{Left}^{\mathsf{P\text{–}S}}_n(x,y_1,\ldots,y_n) \equiv \mathsf{Right}^{\mathsf{P\text{–}S}}_n(x,y_1,\ldots,y_n))$$

and show that $\forall n \in \mathbb{N}.\ n > 1 \Rightarrow \Phi(n)$ by induction on n. The base case, $\Phi(2)$, was discussed in Section 5.5.7.12. The induction hypothesis is $\Phi(k) \Rightarrow \Phi(k+1)$.

$$
\begin{array}{lll}
 & \mathsf{Left}^{\mathsf{P\text{-}S}}_{k+1}(x, y_1, \ldots, y_n) & \\
\equiv & \exists t.\ \mathsf{Proc}(f)(x,t) \wedge \mathsf{Split}_{k+1}(\otimes^{-1})(t, y_1, \ldots, y_{k+1}) & \\
\equiv & \exists t.\ \mathsf{Proc}(f)(x,t) \wedge & \text{by } (5.10) \\
 & \quad (\mathsf{Split}_k(\otimes^{-1})(u, y_1, \ldots, y_k) \wedge \mathsf{Split}(\otimes^{-1})(t, u, y_{k+1})) & \\
\equiv & \exists u.\ \mathsf{Split}_k(\otimes^{-1})(u, y_1, \ldots, y_k) \wedge & \\
 & \quad (\exists t.\ \mathsf{Proc}(f)(x,t) \wedge \mathsf{Split}(\otimes^{-1})(t, u, y_{k+1})) & \\
\equiv & \exists u.\ \mathsf{Split}_k(\otimes^{-1})(u, y_1, \ldots, y_k) \wedge & (\dagger) \\
 & \quad (\exists t_1, t_2.\ \mathsf{Split}(\star^{-1})(x, t_1, t_2) \wedge \mathsf{Proc}(f)(t_1, u) \wedge \mathsf{Proc}(f)(t_2, y_{k+1})) & \\
\equiv & \exists t_1, t_2.\ (\exists u.\ \mathsf{Proc}(f)(t_1, u) \wedge \mathsf{Split}_k(\otimes^{-1})(u, y_1, \ldots, y_k)) \wedge & \\
 & \qquad \mathsf{Split}(\star^{-1})(x, t_1, t_2) \wedge \mathsf{Proc}(f)(t_2, y_{k+1}) & \\
\equiv & \exists t_1, t_2.\ (\exists u_1, \ldots, u_k.\ \mathsf{Split}_k(\star^{-1})(t_1, u_1, \ldots, u_k) \wedge \bigwedge_{i=1}^{k} \mathsf{Proc}(f)(u_i, y_i)) \wedge & \text{by } \Phi(k) \\
 & \qquad \mathsf{Split}(\star^{-1})(x, t_1, t_2) \wedge \mathsf{Proc}(f)(t_2, y_{k+1}) & \\
\equiv & \exists u_1, \ldots, u_k, t_2.\ (\exists t_1.\ \mathsf{Split}_k(\star^{-1})(t_1, u_1, \ldots, u_k) \wedge \mathsf{Split}(\star^{-1})(x, t_1, t_2)) \wedge & \\
 & \qquad\qquad \mathsf{Proc}(f)(t_2, y_{k+1}) \wedge \bigwedge_{i=1}^{k} \mathsf{Proc}(f)(u_i, y_i) & \\
\equiv & \exists u_1, \ldots, u_k, t_2.\ \mathsf{Split}_{k+1}(\star^{-1})(x, u_1, \ldots, u_k, t_2) \wedge & \text{by } (5.10) \\
 & \qquad\qquad \mathsf{Proc}(f)(t_2, y_{k+1}) \wedge \bigwedge_{i=1}^{k} \mathsf{Proc}(f)(u_i, y_i & \\
\equiv & \exists u_1, \ldots, u_{k+1}.\ \mathsf{Split}_{k+1}(x, u_1, \ldots, u_{k+1}) \wedge \bigwedge_{i=1}^{k+1} \mathsf{Proc}(f)(u_i, y_i) & \\
\equiv & \mathsf{Right}^{\mathsf{P\text{-}S}}_{k+1}(x, y_1, \ldots, y_n), &
\end{array}
$$

where the step marked (†) is valid under the condition that the outputs are subsequently merged (proof omitted).

Hence $\Phi(n)$ holds for all $n > 1$. □

I.7 Split–Replication transformation

The denotations of the task graphs in the (m, n)-ary version of the Split–Replication transformation are as follows. The index m corresponds to the number of outputs from the split task; the index n corresponds to the number of outputs from the replication task.

$$
\begin{array}{rcl}
\mathsf{Left}^{\mathsf{S\text{-}R}}_{m,n}(x, x_{1,1}, \ldots, x_{m,n}) & \triangleq & \exists t_1, \ldots, t_m.\ \mathsf{Split}_m(\star^{-1})(x, t_1, \ldots, t_m) \wedge \\
 & & \qquad \bigwedge_{i=1}^{m} \mathsf{Rep}_n(t_i, x_{i,1}, \ldots, x_{i,n}) \\
\mathsf{Right}^{\mathsf{S\text{-}R}}_{m,n}(x, x_{1,1}, \ldots, x_{m,n}) & \triangleq & \exists t_1, \ldots, t_n.\ \mathsf{Rep}_n(x, t_1, \ldots, t_n) \wedge \\
 & & \qquad \bigwedge_{i=1}^{n} \mathsf{Split}_m(\star^{-1})(t_i, x_{1,i}, \ldots, x_{m,i})
\end{array}
$$

Proposition I.7. *The n-ary version of the Split–Replication transformation is sound.*

Proof. We define the property

$$\Phi(m, n) \triangleq (\mathsf{Left}^{\mathsf{S\text{-}R}}_{m,n}(x, x_{1,1}, \ldots, x_{m,n}) \equiv \mathsf{Right}^{\mathsf{S\text{-}R}}_{m,n}(x, x_{1,1}, \ldots, x_{m,n}))$$

and show that $\forall m, n \in \mathbb{N}.\ m > 1 \wedge n > 1 \Rightarrow \Phi(m, n)$ by induction on both m and n. The base case, $\Phi(2, 2)$, was proven earlier in (5.20). We will employ two inductive steps, $\Phi(j, k) \Rightarrow \Phi(j, k+1)$ and $\Phi(j, k) \Rightarrow \Phi(j+1, k)$ so that we explore the entirety of $\mathbb{N} \times \mathbb{N}$ space.

To assist with the proof, we begin by showing that $\forall j \in \mathbb{N}.\ j > 1 \Rightarrow \Phi(j, 2)$ by induction on j:

$$
\begin{array}{lll}
& \mathsf{Left}^{\mathsf{S\text{-}R}}_{j+1,2}(x, x_{1,1}, \ldots, x_{j+1,2}) & \\
\equiv & \exists t_1, \ldots, t_{j+1}.\ \mathsf{Split}_{j+1}(\star^{-1})(x, t_1, \ldots, t_{j+1}) \wedge \bigwedge_{i=1}^{j+1} \mathsf{Rep}(t_i, x_{i,1}, x_{i,2}) & \\
\equiv & \exists t_1, \ldots, t_{j+1}.\ (\exists u.\ \mathsf{Split}_j(\star^{-1})(u, t_1, \ldots, t_j) \wedge \mathsf{Split}(\star^{-1})(x, u, t_{j+1})) \wedge & \text{by (5.10)} \\
& \qquad \bigwedge_{i=1}^{j+1} \mathsf{Rep}(t_i, x_{i,1}, x_{i,2}) & \\
\equiv & \exists t_{j+1}, u.\ \mathsf{Split}(\star^{-1})(x, u, t_{j+1}) \wedge \mathsf{Rep}(t_{j+1}, x_{j+1,1}, x_{j+1,2}) \wedge & \\
& \qquad (\exists t_1, \ldots, t_j.\ \mathsf{Split}_j(\star^{-1})(u, t_1, \ldots, t_j) \wedge & \\
& \qquad\qquad \bigwedge_{i=1}^{j} \mathsf{Rep}(t_i, x_{i,1}, x_{i,2})) & \\
\equiv & \exists t_{j+1}, u.\ \mathsf{Split}(\star^{-1})(x, u, t_{j+1}) \wedge \mathsf{Rep}(t_{j+1}, x_{j+1,1}, x_{j+1,2}) \wedge & \text{by } \Phi(j, 2) \\
& \qquad (\exists t_1, t_2.\ \mathsf{Rep}(u, t_1, t_2) \wedge & \\
& \qquad\qquad \mathsf{Split}_j(\star^{-1})(t_1, x_{1,1}, \ldots, x_{j,1}) \wedge & \\
& \qquad\qquad \mathsf{Split}_j(\star^{-1})(t_2, x_{1,2}, \ldots, x_{j,2})) & \\
\equiv & \exists t_1, t_2.\ (\exists t_{j+1}, u.\ \mathsf{Split}(\star^{-1})(x, u, t_{j+1}) \wedge \mathsf{Rep}(u, t_1, t_2) \wedge & \\
& \qquad\qquad \mathsf{Rep}(t_{j+1}, x_{j+1,1}, x_{j+1,2})) \wedge & \\
& \qquad \mathsf{Split}_j(\star^{-1})(t_1, x_{1,1}, \ldots, x_{j,1}) \wedge \mathsf{Split}_j(\star^{-1})(t_2, x_{1,2}, \ldots, x_{j,2}) & \\
\equiv & \exists t_1, t_2.\ (\exists u_1, u_2.\ \mathsf{Rep}(x, u_1, u_2) \wedge \mathsf{Split}(\star^{-1})(u_1, t_1, x_{j+1,1}) \wedge & \text{by (5.20)} \\
& \qquad\qquad \mathsf{Split}(\star^{-1})(u_2, t_2, x_{j+1,2})) \wedge & \\
& \qquad \mathsf{Split}_j(\star^{-1})(t_1, x_{1,1}, \ldots, x_{j,1}) \wedge \mathsf{Split}_j(\star^{-1})(t_2, x_{1,2}, \ldots, x_{j,2}) & \\
\equiv & \exists u_1, u_2.\ \mathsf{Rep}(x, u_1, u_2) \wedge & \\
& \qquad (\exists t_1.\ \mathsf{Split}(\star^{-1})(t_1, x_{1,1}, \ldots, x_{j,1}) \wedge \mathsf{Split}(\star^{-1})(u_1, t_1, x_{j+1,1})) \wedge & \\
& \qquad (\exists t_2.\ \mathsf{Split}(\star^{-1})(t_2, x_{1,2}, \ldots, x_{j,2}) \wedge \mathsf{Split}(\star^{-1})(u_2, t_2, x_{j+1,2})) & \\
\equiv & \exists u_1, u_2.\ \mathsf{Rep}(x, u_1, u_2) \wedge & \text{by (5.10)} \\
& \qquad \mathsf{Split}_{j+1}(\star^{-1})(u_1, x_{1,1}, \ldots, x_{j+1,1}) \wedge & \\
& \qquad \mathsf{Split}_{j+1}(\star^{-1})(u_2, x_{1,2}, \ldots, x_{j+1,2}) & \\
\equiv & \mathsf{Right}^{\mathsf{S\text{-}R}}_{j+1,2}(x, x_{1,1}, \ldots, x_{j+1,2}). &
\end{array}
$$

Hence $\Phi(j, 2)$ holds for all $j > 1$.

Similarly, we show that $\forall k \in \mathbb{N}.\ k > 1 \Rightarrow \Phi(2,k)$ by induction on k:

$$
\begin{array}{lll}
 & \mathsf{Left}^{\mathsf{S\text{–}R}}_{2,k+1}(x, x_{1,1}, \ldots, x_{2,k+1}) & \\
\equiv & \exists t_1, t_2.\ \mathsf{Split}(\star^{-1})(x, t_1, t_2) \wedge & \\
 & \qquad \mathsf{Rep}_{k+1}(t_1, x_{1,1}, \ldots, x_{1,k+1}) \wedge \mathsf{Rep}_{k+1}(t_2, x_{2,1}, \ldots, x_{2,k+1}) & \\
\equiv & \exists t_1, t_2.\ \mathsf{Split}(\star^{-1})(x, t_1, t_2) \wedge & \text{by (5.11)} \\
 & \qquad (\exists u_1.\ \mathsf{Rep}_k(u_1, x_{1,1}, \ldots, x_{1,k}) \wedge \mathsf{Rep}(t_1, u_1, x_{1,k+1})) \wedge & \\
 & \qquad (\exists u_2.\ \mathsf{Rep}_k(u_2, x_{2,1}, \ldots, x_{2,k}) \wedge \mathsf{Rep}(t_2, u_2, x_{2,k+1})) & \\
\equiv & \exists u_1, u_2.\ (\exists t_1, t_2.\ \mathsf{Split}(\star^{-1})(x, t_1, t_2) \wedge & \\
 & \qquad \mathsf{Rep}(t_1, u_1, x_{1,k+1}) \wedge \mathsf{Rep}(t_2, u_2, x_{2,k+1})) \wedge & \\
 & \quad \mathsf{Rep}_k(u_1, x_{1,1}, \ldots, x_{1,k}) \wedge \mathsf{Rep}_k(u_2, x_{2,1}, \ldots, x_{2,k}) & \\
\equiv & \exists u_1, u_2.\ (\exists v_1, v_2.\ \mathsf{Rep}(x, v_1, v_2) \wedge & \text{by (5.20)} \\
 & \qquad \mathsf{Split}(\star^{-1})(v_1, u_1, u_2) \wedge & \\
 & \qquad \mathsf{Split}(\star^{-1})(v_2, x_{1,k+1}, x_{2,k+1})) \wedge & \\
 & \quad \mathsf{Rep}_k(u_1, x_{1,1}, \ldots, x_{1,k}) \wedge \mathsf{Rep}_k(u_2, x_{2,1}, \ldots, x_{2,k}) & \\
\equiv & \exists v_1, v_2.\ (\exists u_1, u_2.\ \mathsf{Split}(\star^{-1})(v_1, u_1, u_2) \wedge & \\
 & \qquad \mathsf{Rep}_k(u_1, x_{1,1}, \ldots, x_{1,k}) \wedge & \\
 & \qquad \mathsf{Rep}_k(u_2, x_{2,1}, \ldots, x_{2,k})) \wedge & \\
 & \quad \mathsf{Rep}(x, v_1, v_2) \wedge \mathsf{Split}(\star^{-1})(v_2, x_{1,k+1}, x_{2,k+1}) & \\
\equiv & \exists v_1, v_2.\ (\exists t_1, \ldots, t_k.\ \mathsf{Rep}_k(v_1, t_1, \ldots, t_k) \wedge & \text{by } \Phi(2,k) \\
 & \qquad \bigwedge_{i=1}^{k} \mathsf{Split}(\star^{-1})(t_i, x_{1,i}, x_{2,i})) \wedge & \\
 & \quad \mathsf{Rep}(x, v_1, v_2) \wedge \mathsf{Split}(\star^{-1})(v_2, x_{1,k+1}, x_{2,k+1}) & \\
\equiv & \exists t_1, \ldots t_k, v_2.\ (\exists v_1.\ \mathsf{Rep}_k(v_1, t_1, \ldots, t_k) \wedge \mathsf{Rep}(x, v_1, v_2)) \wedge & \\
 & \qquad \mathsf{Split}(\star^{-1})(v_2, x_{1,k+1}, x_{2,k+1}) \wedge & \\
 & \qquad \bigwedge_{i=1}^{k} \mathsf{Split}(\star^{-1})(t_i, x_{1,i}, x_{2,i}) & \\
\equiv & \exists t_1, \ldots t_k, v_2.\ \mathsf{Rep}_{k+1}(x, t_1, \ldots, t_k, v_2) \wedge & \text{by (5.11)} \\
 & \qquad \mathsf{Split}(\star^{-1})(v_2, x_{1,k+1}, x_{2,k+1}) \wedge & \\
 & \qquad \bigwedge_{i=1}^{k} \mathsf{Split}(\star^{-1})(t_i, x_{1,i}, x_{2,i}) & \\
\equiv & \exists t_1, \ldots t_{k+1}.\ \mathsf{Rep}_{k+1}(x, t_1, \ldots, t_{k+1}) \wedge \bigwedge_{i=1}^{k+1} \mathsf{Split}(\star^{-1})(t_i, x_{1,i}, x_{2,i}) & \\
\equiv & \mathsf{Right}^{\mathsf{S\text{–}R}}_{2,k+1}(x, x_{1,1}, \ldots, x_{2,k+1}). &
\end{array}
$$

Hence $\Phi(2,k)$ holds for all $k > 1$.

Now we tackle the first inductive step, $\Phi(j,k) \Rightarrow \Phi(j,k+1)$:

$$\begin{array}{lll}
& \mathsf{Left}^{\mathsf{S-R}}_{j,k+1}(x, x_{1,1}, \ldots, x_{j,k+1}) & \\
\equiv & \exists t_1, \ldots, t_j.\ \mathsf{Split}_j(\star^{-1})(x, t_1, \ldots, t_j) \wedge & \\
& \qquad\qquad \bigwedge_{i=1}^{j} \mathsf{Rep}_{k+1}(t_i, x_{i,1}, \ldots, x_{i,k+1}) & \\
\equiv & \exists t_1, \ldots, t_j.\ \mathsf{Split}_j(\star^{-1})(x, t_1, \ldots, t_j) \wedge & \text{by (5.11)} \\
& \qquad\qquad \bigwedge_{i=1}^{j} (\exists u.\ \mathsf{Rep}_k(u, x_{i,1}, \ldots, x_{i,k}) \wedge \mathsf{Rep}(t_i, u, x_{i,k+1})) & \\
\equiv & \exists u_1, \ldots, u_j.\ (\exists t_1, \ldots t_j.\ \mathsf{Split}_j(\star^{-1})(x, t_1, \ldots, t_j) \wedge & \\
& \qquad\qquad\qquad \bigwedge_{i=1}^{j} \mathsf{Rep}(t_i, u_i, x_{i,k+1})) \wedge & \\
& \qquad\qquad \bigwedge_{i=1}^{j} \mathsf{Rep}_k(u_i, x_{i,1}, \ldots, x_{i,k}) & \\
\equiv & \exists u_1, \ldots, u_j.\ (\exists t_1, t_2.\ \mathsf{Rep}(x, t_1, t_2) \wedge & \text{by } \Phi(j,2) \\
& \qquad\qquad\qquad \mathsf{Split}_j(\star^{-1})(t_1, u_1, \ldots, u_j) \wedge & \\
& \qquad\qquad\qquad \mathsf{Split}_j(\star^{-1})(t_2, x_{1,k+1}, \ldots, x_{j,k+1})) \wedge & \\
& \qquad\qquad \bigwedge_{i=1}^{j} \mathsf{Rep}_k(u_i, x_{i,1}, \ldots, x_{i,k}) & \\
\equiv & \exists t_1, t_2.\ \mathsf{Rep}(x, t_1, t_2) \wedge \mathsf{Split}_j(\star^{-1})(t_2, x_{1,k+1}, \ldots, x_{j,k+1}) \wedge & \\
& \qquad (\exists u_1, \ldots, u_j.\ \mathsf{Split}_j(\star^{-1})(t_1, u_1, \ldots, u_j) \wedge & \\
& \qquad\qquad\qquad \bigwedge_{i=1}^{j} \mathsf{Rep}_k(u_i, x_{i,1}, \ldots, x_{i,k})) & \\
\equiv & \exists t_1, t_2.\ \mathsf{Rep}(x, t_1, t_2) \wedge \mathsf{Split}_j(\star^{-1})(t_2, x_{1,k+1}, \ldots, x_{j,k+1}) \wedge & \text{by } \Phi(j,k) \\
& \qquad (\exists u_1, \ldots, u_k.\ \mathsf{Rep}_k(t_1, u_1, \ldots, u_k) \wedge & \\
& \qquad\qquad\qquad \bigwedge_{i=1}^{k} \mathsf{Split}_j(\star^{-1})(u_i, x_{1,i}, \ldots, x_{j,i})) & \\
\equiv & \exists u_1, \ldots, u_k, t_2.\ (\exists t_1.\ \mathsf{Rep}(x, t_1, t_2) \wedge \mathsf{Rep}_k(t_1, u_1, \ldots, u_k)) \wedge & \\
& \qquad\qquad \mathsf{Split}_j(\star^{-1})(t_2, x_{1,k+1}, \ldots, x_{j,k+1}) \wedge & \\
& \qquad\qquad \bigwedge_{i=1}^{k} \mathsf{Split}_j(\star^{-1})(u_i, x_{1,i}, \ldots, x_{j,i}) & \\
\equiv & \exists u_1, \ldots, u_k, t_2.\ \mathsf{Rep}_{k+1}(x, u_1, \ldots, u_k, t_2) \wedge & \text{by (5.11)} \\
& \qquad\qquad \mathsf{Split}_j(\star^{-1})(t_2, x_{1,k+1}, \ldots, x_{j,k+1}) \wedge & \\
& \qquad\qquad \bigwedge_{i=1}^{k} \mathsf{Split}_j(\star^{-1})(u_i, x_{1,i}, \ldots, x_{j,i}) & \\
\equiv & \exists u_1, \ldots, u_{k+1}.\ \mathsf{Rep}_{k+1}(x, u_1, \ldots, u_{k+1}) \wedge & \\
& \qquad\qquad \bigwedge_{i=1}^{k+1} \mathsf{Split}_j(\star^{-1})(u_i, x_{1,i}, \ldots, x_{j,i}) & \\
\equiv & \mathsf{Right}^{\mathsf{S-R}}_{j,k+1}(x, x_{1,1}, \ldots, x_{j,k+1}). &
\end{array}$$

Hence $\Phi(j,k) \Rightarrow \Phi(j,k+1)$ for all $j, k > 1$.

And now the second inductive step, $\Phi(j,k) \Rightarrow \Phi(j+1,k)$:

$$
\begin{array}{lll}
 & \mathsf{Left}^{\mathsf{S\text{–}R}}_{j+1,k}(x, x_{1,1}, \dots, x_{j+1,k}) & \\
\equiv & \exists t_1, \dots, t_{j+1}.\ \mathsf{Split}_{j+1}(\star^{-1})(x, t_1, \dots, t_{j+1}) \wedge & \\
 & \qquad\qquad \bigwedge_{i=1}^{j+1} \mathsf{Rep}_k(t_i, x_{i,1}, \dots, x_{i,k}) & \\
\equiv & \exists t_1, \dots, t_{j+1}.\ (\exists u.\ \mathsf{Split}_j(\star^{-1})(u, t_1, \dots, t_j) \wedge \mathsf{Split}(x, u, t_{j+1})) \wedge & \text{by (5.10)} \\
 & \qquad\qquad \bigwedge_{i=1}^{j+1} \mathsf{Rep}_k(t_i, x_{i,1}, \dots, x_{i,k}) & \\
\equiv & \exists u, t_{j+1}.\ \mathsf{Split}(x, u, t_{j+1}) \wedge \mathsf{Rep}_k(t_{j+1}, x_{j+1,1}, \dots, x_{j+1,k}) \wedge & \\
 & \qquad (\exists t_1, \dots, t_j.\ \mathsf{Split}_j(\star^{-1})(u, t_1, \dots, t_j) \wedge & \\
 & \qquad\qquad\qquad \bigwedge_{i=1}^{j} \mathsf{Rep}_k(t_i, x_{i,1}, \dots, x_{i,k})) & \\
\equiv & \exists u, t_{j+1}.\ \mathsf{Split}(x, u, t_{j+1}) \wedge \mathsf{Rep}_k(t_{j+1}, x_{j+1,1}, \dots, x_{j+1,k}) \wedge & \text{by } \Phi(j,k) \\
 & \qquad (\exists u_1, \dots, u_k.\ \mathsf{Rep}_k(u, u_1, \dots, u_k) \wedge & \\
 & \qquad\qquad\qquad \bigwedge_{i=1}^{k} \mathsf{Split}_j(\star^{-1})(u_i, x_{1,i}, \dots, x_{j_i})) & \\
\equiv & \exists u_1, \dots, u_k.\ (\exists u, t_{j+1}.\ \mathsf{Split}(x, u, t_{j+1}) \wedge \mathsf{Rep}_k(u, u_1, \dots, u_k) \wedge & \\
 & \qquad\qquad \mathsf{Rep}_k(t_{j+1}, x_{j+1,1}, \dots, x_{j+1,k})) \wedge & \\
 & \qquad \bigwedge_{i=1}^{k} \mathsf{Split}_j(\star^{-1})(u_i, x_{1,i}, \dots, x_{j,i}) & \\
\equiv & \exists u_1, \dots, u_k.\ (\exists v_1, \dots v_k.\ \mathsf{Rep}_k(x, v_1, \dots, v_k) \wedge & \text{by } \Phi(2,k) \\
 & \qquad\qquad\qquad \bigwedge_{i=1}^{k} \mathsf{Split}(\star^{-1})(v_i, u_i, x_{j+1,i})) \wedge & \\
 & \qquad \bigwedge_{i=1}^{k} \mathsf{Split}_j(\star^{-1})(u_i, x_{1,i}, \dots, x_{j,i}) & \\
\equiv & \exists v_1, \dots, v_k.\ \mathsf{Rep}_k(x, v_1, \dots, v_k) \wedge & \\
 & \qquad \bigwedge_{i=1}^{k} (\exists u_i.\ \mathsf{Split}_j(\star^{-1})(u_i, x_{1,i}, \dots, x_{j,i}) \wedge & \\
 & \qquad\qquad\qquad \mathsf{Split}(\star^{-1})(v_i, u_i, x_{j+1,i})) & \\
\equiv & \exists v_1, \dots, v_k.\ \mathsf{Rep}_k(x, v_1, \dots, v_k) \wedge & \\
 & \qquad \bigwedge_{i=1}^{k+1} \mathsf{Split}_{j+1}(\star^{-1})(v_i, x_{1,i}, \dots, x_{j+1,i}) & \\
\equiv & \mathsf{Right}^{\mathsf{S\text{–}R}}_{j+1,k}(x, x_{1,1}, \dots, x_{j+1,k}). &
\end{array}
$$

Hence $\Phi(j,k) \Rightarrow \Phi(j+1,k)$ for all $j,k > 1$, which concludes the proof of $\Phi(j,k)$ for all $j,k > 1$. □

I.8 Merge–Replication transformation

The denotations of the task graphs in the (m,n)-ary version of the Merge–Replication transformation are as follows. The index m corresponds to the number of inputs to the merge task; the index n corresponds to the number of outputs from the replication task.

$$
\begin{array}{rcl}
\mathsf{Left}^{\mathsf{M\text{–}R}}_{m,n}(x_1, \dots, x_m, y_1, \dots, y_n) & \triangleq & \exists t.\ \mathsf{Merge}_m(\star)(x_1, \dots, x_m, t) \wedge \\
 & & \quad \mathsf{Rep}_n(t, y_1, \dots, y_n) \\
\mathsf{Right}^{\mathsf{M\text{–}R}}_{m,n}(x_1, \dots, x_m, y_1, \dots, y_n) & \triangleq & \exists t_{1,1} \dots t_{m,n}.\ \bigwedge_{i=1}^{m} \mathsf{Rep}_n(x_i, t_{i,1}, \dots, t_{i,n}) \wedge \\
 & & \qquad\qquad \bigwedge_{i=1}^{n} \mathsf{Merge}_m(t_{1,i}, \dots, t_{m,i}, y_i)
\end{array}
$$

Proposition I.8. *The n-ary version of the Merge–Replication transformation is sound.*

Proof. We define the property

$$\Phi(m,n) \triangleq (\mathsf{Left}^{\mathsf{M\text{–}R}}_{m,n}(x_1, \dots, x_m, y_1, \dots, y_n) \equiv \mathsf{Right}^{\mathsf{M\text{–}R}}_{m,n}(x_1, \dots, x_m, y_1, \dots, y_n))$$

and show that $\forall m, n \in \mathbb{N}.\ m > 1 \land n > 1 \Rightarrow \Phi(m, n)$ by induction on both m and n. The base case, $\Phi(2, 2)$, was proven earlier in (5.21). We will employ two inductive steps, $\Phi(j, k) \Rightarrow \Phi(j, k + 1)$ and $\Phi(j, k) \Rightarrow \Phi(j + 1, k)$ so that we explore the entirety of $\mathbb{N} \times \mathbb{N}$ space.

To assist with the proof, we begin by showing that $\forall j \in \mathbb{N}.\ j > 1 \Rightarrow \Phi(j, 2)$ by induction on j:

$$
\begin{array}{lll}
 & \mathsf{Left}^{\mathsf{M\text{-}R}}_{j+1,2}(x_1, \ldots, x_{j+1}, y_1, y_2) & \\
\equiv & \exists t.\ \mathsf{Merge}_{j+1}(\star)(x_1, \ldots, x_{j+1}, t) \land \mathsf{Rep}(t, y_1, y_2) & \\
\equiv & \exists t.\ (\exists u.\ \mathsf{Merge}_j(\star)(x_1, \ldots, x_j, u) \land \mathsf{Merge}(\star)(u, x_{j+1}, t)) \land & \text{by (5.9)} \\
 & \quad \mathsf{Rep}(t, y_1, y_2) & \\
\equiv & \exists u.\ \mathsf{Merge}_j(\star)(x_1, \ldots, x_j, u) \land & \\
 & \quad (\exists t.\ \mathsf{Merge}(\star)(u, x_{j+1}, t) \land \mathsf{Rep}(t, y_1, y_2)) & \\
\equiv & \exists u.\ \mathsf{Merge}_j(\star)(x_1, \ldots, x_j, u) \land & \text{by (5.21)} \\
 & \quad (\exists t_1, t_2, t_3, t_4.\ \mathsf{Rep}(u, t_1, t_2) \land \mathsf{Rep}(x_{j+1}, t_3, t_4) \land & \\
 & \qquad \mathsf{Merge}(\star)(t_1, t_3, y_1) \land \mathsf{Merge}(\star)(t_2, t_4, y_2)) & \\
\equiv & \exists t_1, t_2, t_3, t_4.\ (\exists u.\ \mathsf{Merge}_j(\star)(x_1, \ldots, x_j, u) \land \mathsf{Rep}(u, t_1, t_2)) \land & \\
 & \quad \mathsf{Rep}(x_{j+1}, t_3, t_4) \land & \\
 & \quad \mathsf{Merge}(\star)(t_1, t_3, y_1) \land \mathsf{Merge}(\star)(t_2, t_4, y_2) & \\
\equiv & \exists t_1, t_2, t_3, t_4.\ (\exists u_{1,1}, \ldots, u_{j,2}.\ \bigwedge_{i=1}^{j} \mathsf{Rep}(x_i, u_{i,1}, u_{i,2}) \land & \text{by } \Phi(j, 2) \\
 & \qquad \mathsf{Merge}_j(\star)(u_{1,1}, \ldots, u_{j,1}, t_1) \land & \\
 & \qquad \mathsf{Merge}_j(\star)(u_{1,2}, \ldots, u_{j,2}, t2)) \land & \\
 & \quad \mathsf{Rep}(x_{j+1}, t_3, t_4) \land & \\
 & \quad \mathsf{Merge}(\star)(t_1, t_3, y_1) \land \mathsf{Merge}(\star)(t_2, t_4, y_2) & \\
\equiv & \exists u_{1,1}, \ldots, u_{j,2}, t_3, t_4.\ (\exists t_1.\ \mathsf{Merge}_j(\star)(u_{1,1}, \ldots, u_{j,1}, t_1) \land & \\
 & \qquad \mathsf{Merge}(\star)(t_1, t_3, y_1)) \land & \\
 & \quad (\exists t_2.\ \mathsf{Merge}_j(\star)(u_{1,2}, \ldots, u_{j,2}, t_2) \land & \\
 & \qquad \mathsf{Merge}(\star)(t_2, t_4, y_2)) \land & \\
 & \quad \bigwedge_{i=1}^{j} \mathsf{Rep}(x_i, u_{i,1}, u_{i,2}) \land \mathsf{Rep}(x_{j+1}, t_3, t_4) & \\
\equiv & \exists u_{1,1}, \ldots, u_{j,2}, t_3, t_4.\ \mathsf{Merge}_{j+1}(\star)(u_{1,1}, \ldots, u_{j,1}, t_3, y_1) \land & \text{by (5.9)} \\
 & \quad \mathsf{Merge}_{j+1}(\star)(u_{1,2}, \ldots, u_{j,2}, t_4, y_2) \land & \\
 & \quad \bigwedge_{i=1}^{j} \mathsf{Rep}(x_i, u_{i,1}, u_{i,2}) \land \mathsf{Rep}(x_{j+1}, t_3, t_4) & \\
\equiv & \exists u_{i,1}, \ldots, u_{j+1,2}.\ \mathsf{Merge}_{j+1}(\star)(u_{1,1}, \ldots, u_{j+1,1}, y_1) & \\
 & \quad \mathsf{Merge}_{j+1}(\star)(u_{1,2}, \ldots, u_{j+1,2}, y_2) & \\
 & \quad \bigwedge_{i=1}^{j} \mathsf{Rep}(x_i, u_{i,1}, u_{i,2}) & \\
\equiv & \mathsf{Right}^{\mathsf{M\text{-}R}}_{j+1,2}(x_1, \ldots, x_{j+1}, y_1, y_2). &
\end{array}
$$

Hence $\Phi(j, 2)$ holds for all $j > 1$.

Similarly, we show that $\forall k \in \mathbb{N}.\ k > 1 \Rightarrow \Phi(2,k)$ by induction on k:

$$\begin{array}{lll}
& \mathsf{Left}^{\mathsf{M\text{–}R}}_{2,k+1}(x_1, x_2, y_1, \ldots, y_{k+1}) & \\
\equiv & \exists t.\ \mathsf{Merge}(\star)(x_1, x_2, t) \wedge \mathsf{Rep}_{k+1}(t, y_1, \ldots, y_{k+1}) & \\
\equiv & \exists t.\ \mathsf{Merge}(\star)(x_1, x_2, t) \wedge & \text{by (5.11)} \\
& \quad (\exists u.\ \mathsf{Rep}_k(t, y_1, \ldots, y_k) \wedge \mathsf{Rep}(t, u, y_{k+1})) & \\
\equiv & \exists u.\ (\exists t.\ \mathsf{Merge}(\star)(x_1, x_2, t) \wedge \mathsf{Rep}(t, u, y_{k+1})) \wedge & \\
& \quad \mathsf{Rep}_k(u, y_1, \ldots, y_k) & \\
\equiv & \exists u.\ (\exists t_1, t_2, t_3, t_4.\ \mathsf{Rep}(x_1, t_1, t_2) \wedge \mathsf{Rep}(x_2, t_3, t_4) \wedge & \text{by (5.21)} \\
& \qquad \mathsf{Merge}(\star)(t_1, t_3, u) \wedge \mathsf{Merge}(\star)(t_2, t_4, y_{k+1})) \wedge & \\
& \quad \mathsf{Rep}_k(u, y_1, \ldots, y_k) & \\
\equiv & \exists t_1, t_2, t_3, t_4.\ \mathsf{Rep}(x_1, t_1, t_2) \wedge \mathsf{Rep}(x_2, t_3, t_4) \wedge & \\
& \qquad \mathsf{Merge}(\star)(t_2, t_4, y_{k+1}) \wedge & \\
& \qquad (\exists u.\ \mathsf{Merge}(\star)(t_1, t_3, u) \wedge \mathsf{Rep}_k(u, y_1, \ldots, y_k)) & \\
\equiv & \exists t_1, t_2, t_3, t_4.\ \mathsf{Rep}(x_1, t_1, t_2) \wedge \mathsf{Rep}(x_2, t_3, t_4) \wedge & \text{by } \Phi(2,k) \\
& \qquad \mathsf{Merge}(\star)(t_2, t_4, y_{k+1}) \wedge & \\
& \qquad (\exists u_{1,1}, \ldots, u_{2,k}.\ \mathsf{Rep}_k(t_1, u_{1,1}, \ldots, u_{1,k}) \wedge & \\
& \qquad\qquad \mathsf{Rep}_k(t_3, u_{2,1}, \ldots, u_{2,k}) \wedge & \\
& \qquad\qquad \bigwedge_{i=1}^{k} \mathsf{Merge}(\star)(u_{1,i}, u_{2,i}, y_i)) & \\
\equiv & \exists u_{1,1}, \ldots, u_{2,k}, t_2, t_4.\ (\exists t_1.\ \mathsf{Rep}(x_1, t_1, t_2) \wedge \mathsf{Rep}_k(t_1, u_{1,1}, \ldots, u_{1,k})) \wedge & \\
& \qquad (\exists t_3.\ \mathsf{Rep}(x_2, t_3, t_4) \wedge \mathsf{Rep}_k(t_3, u_{2,1}, \ldots, u_{2,k})) \wedge & \\
& \qquad \bigwedge_{i=1}^{k} \mathsf{Merge}(\star)(u_{1,i}, u_{2,i}, y_i) \wedge & \\
& \qquad \mathsf{Merge}(\star)(t_2, t_4, y_{k+1}) & \\
\equiv & \exists u_{1,1}, \ldots, u_{2,k}, t_2, t_4.\ \mathsf{Rep}_{k+1}(x_1, u_{1,1}, \ldots, u_{1,k}, t_2) \wedge & \text{by (5.11)} \\
& \qquad \mathsf{Rep}_{k+1}(x_2, u_{2,1}, \ldots, u_{2,k}, t_4) \wedge & \\
& \qquad \bigwedge_{i=1}^{k} \mathsf{Merge}(\star)(u_{1,i}, u_{2,i}, y_i) \wedge & \\
& \qquad \mathsf{Merge}(\star)(t_2, t_4, y_{k+1}) & \\
\equiv & \exists u_{1,1}, \ldots, u_{2,k+1}.\ \mathsf{Rep}_{k+1}(x_1, u_{1,1}, \ldots, u_{1,k+1}) \wedge & \\
& \qquad \mathsf{Rep}_{k+1}(x_2, u_{2,1}, \ldots, u_{2,k+1}) \wedge & \\
& \qquad \bigwedge_{i=1}^{k} \mathsf{Merge}(\star)(u_{1,i}, u_{2,i}, y_i) & \\
\equiv & \mathsf{Right}^{\mathsf{M\text{–}R}}_{2,k+1}(x_1, x_2, y_1, \ldots, y_{k+1}). &
\end{array}$$

Hence $\Phi(2,k)$ holds for all $k > 1$.

Now we tackle the first inductive step, $\Phi(j,k) \Rightarrow \Phi(j,k+1)$:

$$\begin{array}{lll}
 & \mathsf{Left}^{\mathsf{M\text{-}R}}_{j,k+1}(x_1,\ldots,x_j,y_1,\ldots,y_{k+1}) & \\
\equiv & \exists t.\ \mathsf{Merge}_j(\star)(x_1,\ldots,x_j,t) \wedge \mathsf{Rep}_{k+1}(t,y_1,\ldots,y_{k+1}) & \\
\equiv & \exists t.\ \mathsf{Merge}_j(\star)(x_1,\ldots,x_j,t) \wedge & \text{by (5.11)} \\
 & \quad (\exists u.\ \mathsf{Rep}_k(u,y_1,\ldots,y_k) \wedge \mathsf{Rep}(t,u,y_{k+1})) & \\
\equiv & \exists u.\ (\exists t.\ \mathsf{Merge}_j(\star)(x_1,\ldots,x_j,t) \wedge \mathsf{Rep}(t,u,y_{k+1})) \wedge & \\
 & \quad \mathsf{Rep}_k(u,y_1,\ldots,y_k) & \\
\equiv & \exists u.\ (\exists t_{1,1},\ldots,t_{j,2}.\ \bigwedge_{i=1}^{j} \mathsf{Rep}(x_i,t_{i,1},t_{i,2}) \wedge & \text{by } \Phi(j,2) \\
 & \qquad \mathsf{Merge}_j(\star)(t_{1,1},\ldots,t_{j,1},u) \wedge & \\
 & \qquad \mathsf{Merge}_j(\star)(t_{1,2},\ldots,t_{j,2},y_{k+1})) \wedge & \\
 & \quad \mathsf{Rep}_k(u,y_1,\ldots,y_k) & \\
\equiv & \exists t_{1,1},\ldots,t_{j,2}.\ \bigwedge_{i=1}^{j} \mathsf{Rep}(x_i,t_{i,1},t_{i,2}) \wedge & \\
 & \qquad \mathsf{Merge}_j(\star)(t_{1,2},\ldots,t_{j,2},y_{k+1}) \wedge & \\
 & \qquad (\exists u.\ \mathsf{Merge}_j(\star)(t_{1,1},\ldots,t_{j,1},u) \wedge \mathsf{Rep}_k(u,y_1,\ldots,y_k)) & \\
\equiv & \exists t_{1,1},\ldots,t_{j,2}.\ \bigwedge_{i=1}^{j} \mathsf{Rep}(x_i,t_{i,1},t_{i,2}) \wedge & \text{by } \Phi(j,k) \\
 & \qquad \mathsf{Merge}_j(\star)(t_{1,2},\ldots,t_{j,2},y_{k+1}) \wedge & \\
 & \qquad (\exists u_{1,1},\ldots,u_{j,k}.\ \bigwedge_{i=1}^{j} \mathsf{Rep}_k(t_{i,1},u_{i,1},\ldots,u_{i,k}) \wedge & \\
 & \qquad\qquad \bigwedge_{i=1}^{k} \mathsf{Merge}_j(\star)(u_{1,i},\ldots,u_{j,i},y_i)) & \\
\equiv & \exists u_{1,1},\ldots,u_{j,k},t_{1,2},\ldots,t_{j,2}. & \\
 & \quad \bigwedge_{i=1}^{j}(\exists t_{i,1}.\ \mathsf{Rep}(x_i,t_{i,1},t_{i,2}) \wedge \mathsf{Rep}_k(t_{i,1},u_{i,1},\ldots,u_{i,k})) \wedge & \\
 & \quad \bigwedge_{i=1}^{k} \mathsf{Merge}_j(\star)(u_{1,i},\ldots,u_{j,i},y_i) \wedge & \\
 & \quad \mathsf{Merge}_j(\star)(t_{1,2},\ldots,t_{j,2},y_{k+1}) & \\
\equiv & \exists u_{1,1},\ldots,u_{j,k},t_{1,2},\ldots,t_{j,2}. & \\
 & \quad \bigwedge_{i=1}^{j} \mathsf{Rep}_{k+1}(x_i,u_{i,1},\ldots,u_{i,k},t_{i,2}) \wedge & \text{by (5.11)} \\
 & \quad \bigwedge_{i=1}^{k} \mathsf{Merge}_j(\star)(u_{1,i},\ldots,u_{j,i},y_i) \wedge & \\
 & \quad \mathsf{Merge}_j(\star)(t_{1,2},\ldots,t_{j,2},y_{k+1}) & \\
\equiv & \exists u_{1,1},\ldots,u_{j,k+1}.\ \bigwedge_{i=1} j\mathsf{Rep}_{k+1}(x_i,u_{i,1},\ldots,u_{i,k+1}) \wedge & \\
 & \qquad \bigwedge_{i=1}^{k+1} \mathsf{Merge}_j(\star)(u_{1,i},\ldots,u_{j,i},y_i) & \\
\equiv & \mathsf{Right}^{\mathsf{M\text{-}R}}_{j,k+1}(x_1,\ldots,x_j,y_1,\ldots,y_{k+1}). &
\end{array}$$

Hence $\Phi(j,k) \Rightarrow \Phi(j,k+1)$ for all $j,k > 1$.

And now the second inductive step, $\Phi(j,k) \Rightarrow \Phi(j+1,k)$:

$$
\begin{aligned}
& \mathsf{Left}^{\mathsf{M-R}}_{j+1,k}(x_1,\dots,x_{j+1},y_1,\dots,y_k) \\
\equiv\ & \exists t.\ \mathsf{Merge}_{j+1}(\star)(x_1,\dots,x_{j+1},t) \wedge \mathsf{Rep}_k(t,y_1,\dots,y_k) \\
\equiv\ & \exists t.\ (\exists u.\ \mathsf{Merge}_j(\star)(x_1,\dots,x_j,u) \wedge \mathsf{Merge}(u,x_{j+1},t)) \wedge && \text{by (5.9)} \\
& \quad \mathsf{Rep}_k(t,y_1,\dots,y_k) \\
\equiv\ & \exists u.\ \mathsf{Merge}_j(\star)(x_1,\dots,x_j,u) \wedge \\
& \quad (\exists t.\ \mathsf{Merge}(u,x_{j+1},t) \wedge \mathsf{Rep}_k(t,y_1,\dots,y_k)) \\
\equiv\ & \exists u.\ \mathsf{Merge}_j(\star)(x_1,\dots,x_j,u) \wedge && \text{by } \Phi(2,k) \\
& \quad (\exists t_{1,1},\dots,t_{2,k}.\ \mathsf{Rep}_k(u,t_{1,1},\dots,t_{1,k}) \wedge \\
& \qquad \mathsf{Rep}_k(x_{j+1},t_{2,1},\dots,t_{2,k}) \wedge \\
& \qquad \textstyle\bigwedge_{i=1}^{k} \mathsf{Merge}(\star)(t_{1,i},t_{2,i},y_i)) \\
\equiv\ & \exists t_{1,1},\dots,t_{2,k}.\ (\exists u.\ \mathsf{Merge}_j(\star)(x_1,\dots,x_j,u) \wedge \mathsf{Rep}_k(u,t_{1,1},\dots,t_{1,k})) \wedge \\
& \qquad \mathsf{Rep}_k(x_{j+1},t_{2,1},\dots,t_{2,k}) \wedge \textstyle\bigwedge_{i=1}^{k} \mathsf{Merge}(\star)(t_{1,i},t_{2,i},y_i) \\
\equiv\ & \exists t_{1,1},\dots,t_{2,k}.\ (\exists u_{1,1},\dots,u_{j,k}.\ \textstyle\bigwedge_{i=1}^{j} \mathsf{Rep}_k(x_i,u_{i,1},\dots,u_{i,k}) \wedge && \text{by } \Phi(j,k) \\
& \qquad \textstyle\bigwedge_{i=1}^{k} \mathsf{Merge}_j(\star)(u_{1,i},\dots,u_{j,i},t_{1,i})) \wedge \\
& \qquad \mathsf{Rep}_k(x_{j+1},t_{2,1},\dots,t_{2,k}) \wedge \textstyle\bigwedge_{i=1}^{k} \mathsf{Merge}(\star)(t_{1,i},t_{2,i},y_i) \\
\equiv\ & \exists u_{1,1},\dots,u_{j,k},t_{2,1},\dots,t_{2,k}. \\
& \quad \textstyle\bigwedge_{i=1}^{k}(\exists t_{1,i}.\ \mathsf{Merge}_j(\star)(u_{1,i},\dots,u_{j,i},t_{1,i}) \wedge \\
& \qquad \mathsf{Merge}(\star)(t_{1,i},t_{2,i},y_i)) \wedge \\
& \quad \textstyle\bigwedge_{i=1}^{j} \mathsf{Rep}_k(x_i,u_{i,1},\dots,u_{i,k}) \wedge \mathsf{Rep}_k(x_{j+1},t_{2,1},\dots,t_{2,k}) \\
\equiv\ & \exists u_{1,1},\dots,u_{j,k},t_{2,1},\dots,t_{2,k}. \\
& \quad \textstyle\bigwedge_{i=1}^{k} \mathsf{Merge}_{j+1}(\star)(u_{1,i},\dots,u_{j,i},t_{2,i},y_i) \wedge && \text{by (5.9)} \\
& \quad \textstyle\bigwedge_{i=1}^{j} \mathsf{Rep}_k(x_i,u_{i,1},\dots,u_{i,k}) \wedge \mathsf{Rep}_k(x_{j+1},t_{2,1},\dots,t_{2,k}) \\
\equiv\ & \exists u_{1,1},\dots,u_{j+1,k}.\ \textstyle\bigwedge_{i=1}^{k} \mathsf{Merge}_{j+1}(\star)(u_{1,i},\dots,u_{j+1,i},y_i) \wedge \\
& \qquad \textstyle\bigwedge_{i=1}^{j+1} \mathsf{Rep}_k(x_i,u_{i,1},\dots,u_{i,k}) \\
\equiv\ & \mathsf{Right}^{\mathsf{M-R}}_{j+1,k}(x_1,\dots,x_{j+1},y_1,\dots,y_k).
\end{aligned}
$$

Hence $\Phi(j,k) \Rightarrow \Phi(j+1,k)$ for all $j,k > 1$, which concludes the proof of $\Phi(j,k)$ for all $j,k > 1$. □

APPENDIX J

Transformations involving pair and unpair tasks

We defined pair and unpair tasks in Section 5.6.1. In this appendix, we discuss the additional task graph transformations which arise involving these kinds of task.

J.1 Transformations involving both pair and unpair

Firstly, there are three transformations involving both pair and unpair tasks.

J.1.1 Pair–Unpair transformation

Transformation Pair–Unpair is shown in Figure J.1.

Proposition J.1. *Transformation Pair–Unpair is sound.*

Proof. We compare the denotations of the task graphs on the left and the right of the transformation.

$$\begin{aligned}
\mathsf{Left}^{\triangle\text{-}\triangledown}(x_1, x_2, y_1, y_2) &\triangleq \exists t.\ \mathsf{Pair}(x_1, x_2, t) \wedge \mathsf{Unpair}(t, y_1, y_2) \\
&\equiv \exists t.\ t = (x_1, x_2) \wedge (y_1, y_2) = t \\
&\equiv y_1 = x_1 \wedge y_2 = x_2;
\end{aligned}$$

$$\mathsf{Right}^{\triangle\text{-}\triangledown}(x_1, x_2, y_1, y_2) \triangleq y_1 = x_1 \wedge y_2 = x_2.$$

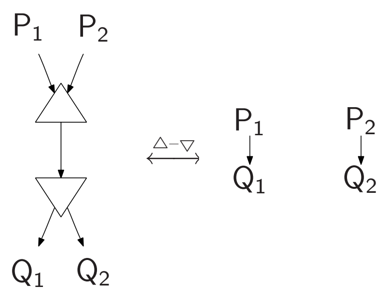


Figure J.1: The Pair–Unpair transformation.

Hence,

$$\mathsf{Left}^{\triangle\text{–}\triangledown}(x_1, x_2, y_1, y_2) \equiv \mathsf{Right}^{\triangle\text{–}\triangledown}(x_1, x_2, y_1, y_2). \tag{J.1}$$

□

J.1.2 Unpair–Pair transformation

Transformation Unpair–Pair is shown in Figure J.2.

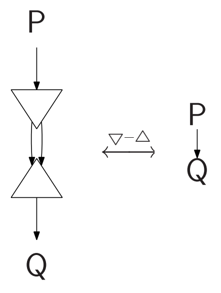


Figure J.2: The Unpair–Pair transformation.

Proposition J.2. *Transformation Unpair–Pair is sound.*

Proof. We compare the denotations of the task graphs on the left and the right of the transformation.

$$\begin{aligned}\mathsf{Left}^{\triangledown\text{–}\triangle}(x, y) &\triangleq \exists t_1, t_2.\ \mathsf{Unpair}(x, t_1, t_2) \wedge \mathsf{Pair}(t_1, t_2, y)\\ &\equiv \exists t_1, t_2.\ (t_1, t_2) = x \wedge y = (t_1, t_2)\\ &\equiv y = x;\end{aligned}$$

$$\mathsf{Right}^{\triangledown\text{–}\triangle}(x, y) \triangleq y = x.$$

Hence,

$$\mathsf{Left}^{\triangledown\text{–}\triangle}(x, y) \equiv \mathsf{Right}^{\triangledown\text{–}\triangle}(x, y). \tag{J.2}$$

□

J.1.3 Combine transformation

A third transformation, Combine, shown in Figure J.3, combines processing functions operating on two elements of a pair into one processing function. Function f must be expressible in terms of g_α and g_β such that

$$\forall a, b.\ f(a, b) = (g_\alpha(a), g_\beta(b)). \tag{J.3}$$

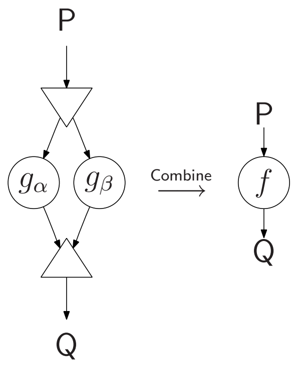


Figure J.3: The Combine transformation.

Note that this transformation is only permitted in the rightwards direction because functions g_α and g_β cannot in general be readily derived from f whereas f is simple to formulate in terms of g_α and g_β.

Proposition J.3. *Transformation Combine is sound.*

Proof. We compare the denotations of the task graphs on the left and the right of the transformation.

$$\begin{aligned}
\mathsf{Left}^{\mathsf{Combine}}(x, y) &\triangleq \exists t_1, t_2, u_1, u_2.\ \mathsf{Unpair}(x, t_1, t_2) \land \\
&\qquad \mathsf{Proc}(g_\alpha)(t_1, u_1) \land \mathsf{Proc}(g_\beta)(t_2, u_2) \land \\
&\qquad \mathsf{Pair}(u_1, u_2, y) \\
&\equiv \exists t_1, t_2, u_1, u_2.\ (t_1, t_2) = x \land \\
&\qquad u_1 = g_\alpha(t_1) \land u_2 = g_\beta(t_2) \land \\
&\qquad y = (u_1, u_2) \\
&\equiv \exists t_1, t_2.\ (t_1, t_2) = x \land y = (g_\alpha(t_1), g_\beta(t_2)) \\
&\equiv \exists t_1, t_2.\ (t_1, t_2) = x \land y = f(t_1, t_2) \qquad \text{by (J.3)} \\
&\equiv y = f(x);
\end{aligned}$$

$$\begin{aligned}
\mathsf{Right}^{\mathsf{Combine}}(x, y) &\triangleq \mathsf{Proc}(f)(x, y) \\
&\equiv y = f(x).
\end{aligned}$$

Hence,

$$\mathsf{Left}^{\mathsf{Combine}}(x, y) \equiv \mathsf{Right}^{\mathsf{Combine}}(x, y). \tag{J.4}$$

□

J.2 Transformations involving pair tasks

Furthermore, there are four transformations involving pair tasks juxtaposed with processing, merge, replication and split tasks respectively.

J.2.1 Pair–Processing transformation

Transformation Pair–Processing is shown in Figure J.4. As with Combine, Pair–Processing is uni-directional.

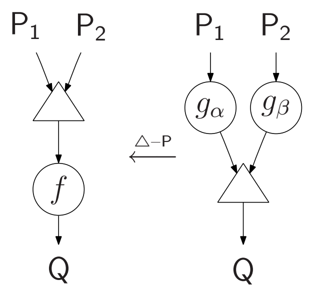


Figure J.4: The Pair–Processing transformation, where $\forall a, b.\ f(a,b) = (g_\alpha(a), g_\beta(b))$.

Proposition J.4. *Transformation Pair–Processing is sound.*

Proof. We compare the denotations of the task graphs on the left and the right of the transformation.

$$\begin{aligned}
\mathsf{Left}^{\triangle\text{–P}}(x_1, x_2, y) &\triangleq \exists t.\ \mathsf{Pair}(x_1, x_2, t) \wedge \mathsf{Proc}(f)(t, y) \\
&\equiv \exists t.\ t = (x_1, x_2) \wedge y = f(t) \\
&\equiv y = f(x_1, x_2);
\end{aligned}$$

$$\begin{aligned}
\mathsf{Right}^{\triangle\text{–P}}(x_1, x_2, y) &\triangleq \exists t_1, t_2.\ \mathsf{Proc}(g_\alpha)(x_1, t_1) \wedge \mathsf{Proc}(g_\beta)(x_2, t_2) \wedge \\
&\qquad\qquad \mathsf{Pair}(t_1, t_2, y) \\
&\equiv \exists t_1, t_2.\ t_1 = g_\alpha(x_1) \wedge t_2 = g_\beta(x_2) \wedge y = (t_1, t_2) \\
&\equiv y = (g_\alpha(x_1), g_\beta(x_2)) \\
&\equiv y = f(x_1, x_2) \qquad \text{by (J.3)}.
\end{aligned}$$

Hence,

$$\mathsf{Left}^{\triangle\text{–P}}(x_1, x_2, y) \equiv \mathsf{Right}^{\triangle\text{–P}}(x_1, x_2, y). \tag{J.5}$$

□

J.2.2 Pair–Merge transformation

Transformation Pair–Merge is shown in Figure J.5. The merge function $\star_{\alpha\times\beta}$ is defined as

$$(a_1, b_1) \star_{\alpha\times\beta} (a_2, b_2) \triangleq (a_1 \star_\alpha a_2, b_1 \star_\beta b_2), \tag{J.6}$$

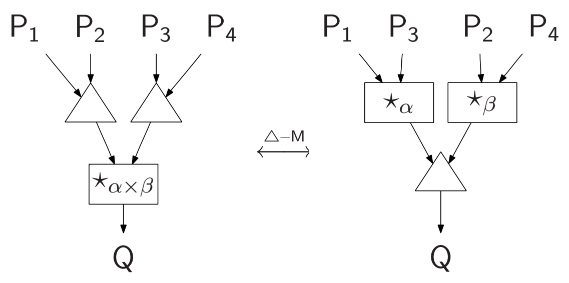


Figure J.5: The Pair–Merge transformation.

Proposition J.5. *Transformation Pair–Merge is sound.*

Proof. We compare the denotations of the task graphs on the left and the right of the transformation.

$$\begin{aligned}
\mathsf{Left}^{\triangle\text{–}\mathsf{M}}(x_1,x_2,x_3,x_4,y) &\triangleq \exists t_1,t_2.\ \mathsf{Pair}(x_1,x_2,t_1)\wedge\mathsf{Pair}(x_3,x_4,t_2)\wedge\\
&\qquad\qquad \mathsf{Merge}(\star_{\alpha\times\beta})(t_1,t_2,y)\\
&\equiv \exists t_1,t_2.\ t_1=(x_1,x_2)\wedge t_2=(x_3,x_4)\wedge\\
&\qquad\qquad y=t_1\star_{\alpha\times\beta}t_2\\
&\equiv y=(x_1,x_2)\star_{\alpha\times\beta}(x_3,x_4)\\
&\equiv y=(x_1\star_\alpha x_3, x_2\star_\beta x_4) \qquad \text{by (J.6);}
\end{aligned}$$

$$\begin{aligned}
\mathsf{Right}^{\triangle\text{–}\mathsf{M}}(x_1,x_2,x_3,x_4,y) &\triangleq \exists t_1,t_2.\ \mathsf{Merge}(\star_\alpha)(x_1,x_3,t_1)\wedge\\
&\qquad\qquad \mathsf{Merge}(\star_\beta)(x_2,x_4,t_2)\wedge\\
&\qquad\qquad \mathsf{Pair}(t_1,t_2,y)\\
&\equiv \exists t_1,t_2.\ t_1=x_1\star_\alpha x_3\wedge t_2=x_2\star_\beta x_4\wedge\\
&\qquad\qquad y=(t_1,t_2)\\
&\equiv y=(x_1\star_\alpha x_3, x_2\star_\beta x_4).
\end{aligned}$$

Hence,

$$\mathsf{Left}^{\triangle\text{–}\mathsf{M}}(x_1,x_2,x_3,x_4,y)\equiv\mathsf{Right}^{\triangle\text{–}\mathsf{M}}(x_1,x_2,x_3,x_4,y). \qquad \text{(J.7)}$$

□

J.2.3 Pair–Replication transformation

Transformation Pair–Replication is shown in Figure J.6.

Proposition J.6. *Transformation Pair–Replication is sound.*

Proof. We compare the denotations of the task graphs on the left and the right of the

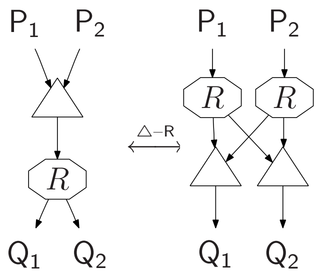


Figure J.6: The Pair–Replication transformation.

transformation.

$$
\begin{aligned}
\mathsf{Left}^{\triangle\text{–R}}(x_1,x_2,y_1,y_2) &\triangleq \exists t.\ \mathsf{Pair}(x_1,x_2,t) \wedge \mathsf{Rep}(t,y_1,y_2)\\
&\equiv \exists t. t=(x_1,x_2) \wedge y_1 = t \wedge y_2 = t\\
&\equiv y_1 = (x_1,x_2) \wedge y_2 = (x_1,x_2);
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{Right}^{\triangle\text{–R}}(x_1,x_2,y_1,y_2) &\triangleq \exists t_1,t_2,t_3,t_4.\ \mathsf{Rep}(x_1,t_1,t_2) \wedge \mathsf{Rep}(x_2,t_3,t_4) \wedge\\
&\qquad \mathsf{Pair}(t_1,t_3,y_1) \wedge \mathsf{Pair}(t_2,t_4,y_2)\\
&\equiv \exists t_1,t_2,t_3,t_4.\ t_1 = x_1 \wedge t_2 = x_1 \wedge\\
&\qquad t_3 = x_2 \wedge t_4 = x_2 \wedge\\
&\qquad y_1 = (t_1,t_3) \wedge y_2 = (t_2,t_4)\\
&\equiv y_1 = (x_1,x_2) \wedge y_2 = (x_1,x_2).
\end{aligned}
$$

Hence,

$$\mathsf{Left}^{\triangle\text{–R}}(x_1,x_2,y_1,y_2) \equiv \mathsf{Right}^{\triangle\text{–R}}(x_1,x_2,y_1,y_2). \tag{J.8}$$

□

J.2.4 Pair–Split transformation

Transformation Pair–Split is shown in Figure J.7. The split function $\star^{-1}_{\alpha\times\beta}$ is defined as

$$\star^{-1}_{\alpha\times\beta}(a,b) = ((a_1,b_1),(a_2,b_2)), \tag{J.9}$$

such that $(a_1,a_2) = \star^{-1}_{\alpha}(a)$ and $(b_1,b_2) = \star^{-1}_{\beta}(b)$.

Proposition J.7. *Transformation Pair–Split is sound.*

Proof. We compare the denotations of the task graphs on the left and the right of the transformation.

$$
\begin{aligned}
\mathsf{Left}^{\triangle\text{–S}}(x_1,x_2,y_1,y_2) &\triangleq \exists t.\ \mathsf{Pair}(x_1,x_2,t) \wedge \mathsf{Split}(\star^{-1}_{\alpha\times\beta})(t,y_1,y_2)\\
&\equiv \exists t. t=(x_1,x_2) \wedge (y_1,y_2) = \star^{-1}_{\alpha\times\beta}(x_1,x_2)\\
&\equiv (y_1,y_2) = \star^{-1}_{\alpha\times\beta}(x_1,x_2);
\end{aligned}
$$

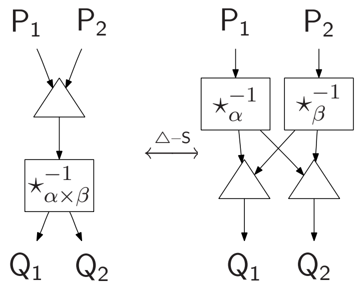


Figure J.7: The Pair–Split transformation.

$$\begin{aligned}
\mathsf{Right}^{\triangle\text{–}\mathsf{S}}(x_1,x_2,y_1,y_2) &\triangleq \exists t_1,t_2,t_3,t_4.\ \mathsf{Split}(\star^{-1}_{\alpha\times\beta})(x_1,t_1,t_2)\wedge\\
&\qquad\qquad\qquad\quad \mathsf{Split}(\star^{-1}_{\alpha\times\beta})(x_2,t_3,t_4)\wedge\\
&\qquad\qquad\qquad\quad \mathsf{Pair}(t_1,t_3,y_1)\wedge\mathsf{Pair}(t_2,t_4,y_2)\\
&\equiv \exists t_1,t_2,t_3,t_4.\ (t_1,t_2)=\star^{-1}_{\alpha}(x_1)\wedge\\
&\qquad\qquad\qquad\quad (t_3,t_4)=\star^{-1}_{\beta}(x_2)\wedge\\
&\qquad\qquad\qquad\quad y_1=(t_1,t_3)\wedge y_2=(t_2,t_4)\\
&\equiv \exists t_1,t_2,t_3,t_4.\ (t_1,t_2)=\star^{-1}_{\alpha}(x_1)\wedge\\
&\qquad\qquad\qquad\quad (t_3,t_4)=\star^{-1}_{\beta}(x_2)\wedge\\
&\qquad\qquad\qquad\quad (y_1,y_2)=((t_1,t_3),(t_2,t_4))\\
&\equiv (y_1,y_2)=\star^{-1}_{\alpha\times\beta}(x_1,x_2)\qquad\text{by (J.9).}
\end{aligned}$$

Hence,

$$\mathsf{Left}^{\triangle\text{–}\mathsf{S}}(x_1,x_2,y_1,y_2)\equiv\mathsf{Right}^{\triangle\text{–}\mathsf{S}}(x_1,x_2,y_1,y_2). \tag{J.10}$$

□

J.3 Transformations involving unpair tasks

Similarly, there are four transformations involving unpair tasks juxtaposed with processing, merge, replication and split tasks respectively.

J.3.1 Unpair–Processing transformation

Transformation Unpair–Processing is shown in Figure J.8. In a similar fashion to Combine and Pair–Processing, the Unpair–Processing transformation is only applicable in the rightwards direction because g_α and g_β cannot be readily derived from f but the converse is possible.

Proposition J.8. *Transformation Unpair–Processing is sound.*

Proof. We compare the denotations of the task graphs on the left and the right of the

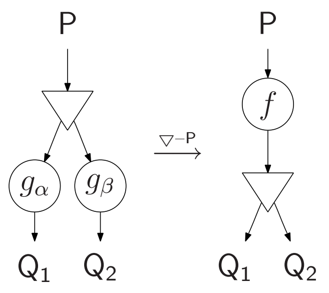


Figure J.8: The Unpair–Processing transformation; $\forall a, b.\ f(a,b) = (g_\alpha(a), g_\beta(b))$.

transformation.

$$\begin{aligned}
\mathsf{Left}^{\triangledown\text{–}\mathsf{P}}(x, y_1, y_2) &\triangleq \exists t_1, t_2.\ \mathsf{Unpair}(x, t_1, t_2) \wedge \\
&\qquad\quad \mathsf{Proc}(g_\alpha)(t_1, y_1) \wedge \mathsf{Proc}(g_\beta)(t_2, y_2) \\
&\equiv \exists t_1, t_2.\ (t_1, t_2) = x \wedge y_1 = g_\alpha(t_1) \wedge y_2 = g_\beta(t_2) \\
&\equiv \exists t_1, t_2.\ (t_1, t_2) = x \wedge (y_1, y_2) = (g_\alpha(t_1), g_\beta(t_2)) \\
&\equiv \exists t_1, t_2.\ (t_1, t_2) = x \wedge (y_1, y_2) = f(t_1, t_2) \qquad \text{by (J.3)} \\
&\equiv (y_1, y_2) = f(x);
\end{aligned}$$

$$\begin{aligned}
\mathsf{Right}^{\triangledown\text{–}\mathsf{P}}(x, y_1, y_2) &\triangleq \exists t.\ \mathsf{Proc}(f)(x, t) \wedge \mathsf{Unpair}(t, y_1, y_2) \\
&\equiv \exists t.\ t = f(x) \wedge (y_1, y_2) = t \\
&\equiv (y_1, y_2) = f(x).
\end{aligned}$$

Hence,

$$\mathsf{Left}^{\triangledown\text{–}\mathsf{P}}(x, y_1, y_2) \equiv \mathsf{Right}^{\triangledown\text{–}\mathsf{P}}(x, y_1, y_2). \tag{J.11}$$

□

J.3.2 Unpair–Merge transformation

Transformation Unpair–Merge is shown in Figure J.9.

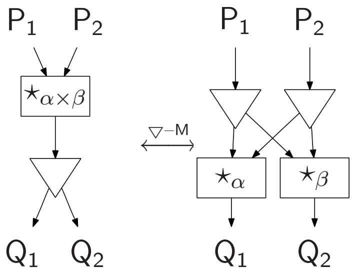


Figure J.9: The Unpair–Merge transformation.

Proposition J.9. *Transformation Unpair–Merge is sound.*

Proof. We compare the denotations of the task graphs on the left and the right of the transformation.

$$\begin{aligned}
\mathsf{Left}^{\triangledown\text{–M}}(x_1, x_2, y_1, y_2) &\triangleq \exists t.\ \mathsf{Merge}(\star_{\alpha\times\beta})(x_1, x_2, t) \wedge \mathsf{Unpair}(t, y_1, y_2)\\
&\equiv \exists t.\ t = x_1 \star_{\alpha\times\beta} x_2 \wedge (y_1, y_2) = t\\
&\equiv (y_1, y_2) = x_1 \star_{\alpha\times\beta} x_2;
\end{aligned}$$

$$\begin{aligned}
\mathsf{Right}^{\triangledown\text{–M}}(x_1, x_2, y_1, y_2) &\triangleq \exists t_1, t_2, t_3, t_4.\ \mathsf{Unpair}(x_1, t_1, t_2) \wedge\\
&\qquad \mathsf{Unpair}(x_2, t_3, t_4) \wedge\\
&\qquad \mathsf{Merge}(\star_\alpha)(t_1, t_3, y_1) \wedge\\
&\qquad \mathsf{Merge}(\star_\beta)(t_2, t_4, y_2)\\
&\equiv \exists t_1, t_2, t_3, t_4.\ (t_1, t_2) = x_1 \wedge (t_3, t_4) = x_2\\
&\qquad y_1 = t_1 \star_\alpha t_3 \wedge y_2 = t_2 \star_\beta t_4\\
&\equiv \exists t_1, t_2, t_3, t_4.\ (t_1, t_2) = x_1 \wedge (t_3, t_4) = x_2\\
&\qquad (y_1, y_2) = (t_1 \star_\alpha t_3, t_2 \star_\beta t_4)\\
&\equiv \exists t_1, t_2, t_3, t_4.\ (t_1, t_2) = x_1 \wedge (t_3, t_4) = x_2\\
&\qquad (y_1, y_2) = (t_1, t_2) \star_{\alpha\times\beta} (t_3, t_4)\\
&\qquad\qquad \text{by (J.9)}\\
&\equiv (y_1, y_2) = x_1 \star_{\alpha\times\beta} x_2.
\end{aligned}$$

Hence,

$$\mathsf{Left}^{\triangledown\text{–M}}(x_1, x_2, y_1, y_2) \equiv \mathsf{Right}^{\triangledown\text{–M}}(x_1, x_2, y_1, y_2). \tag{J.12}$$

□

J.3.3 Unpair–Replication transformation

Transformation Unpair–Replication is shown in Figure J.10.

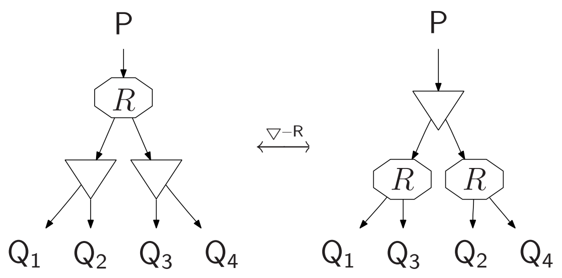


Figure J.10: The Unpair–Replication transformation.

Proposition J.10. *Transformation Unpair–Replication is sound.*

Proof. We compare the denotations of the task graphs on the left and the right of the transformation.

$$\begin{aligned}
\mathsf{Left}^{\triangledown\text{–R}}(x, y_1, y_2, y_3, y_4) &\triangleq \exists t_1, t_2.\ \mathsf{Rep}(x, t_1, t_2) \wedge \\
&\qquad\qquad \mathsf{Unpair}(t_1, y_1, y_2) \wedge \mathsf{Unpair}(t_2, y_3, y_4) \\
&\equiv \exists t_1, t_2.\ t_1 = x \wedge t_2 = x \wedge \\
&\qquad\qquad (y_1, y_2) = t_1 \wedge (y_3, y_4) = t_2 \\
&\equiv (y_1, y_2) = x \wedge (y_3, y_4) = x;
\end{aligned}$$

$$\begin{aligned}
\mathsf{Right}^{\triangledown\text{–R}}(x, y_1, y_2, y_3, y_4) &\triangleq \exists t_1, t_2.\ \mathsf{Unpair}(x, t_1, t_2) \wedge \\
&\qquad\qquad \mathsf{Rep}(t_1, y_1, y_3) \wedge \mathsf{Rep}(t_2, y_2, y_4) \\
&\equiv \exists t_1, t_2.\ (t_1, t_2) = x \wedge \\
&\qquad\qquad y_1 = t_1 \wedge y_3 = t_1 \wedge y_2 = t_2 \wedge y_4 = t_2 \\
&\equiv (y_1, y_2) = x \wedge (y_3, y_4) = x.
\end{aligned}$$

Hence,

$$\mathsf{Left}^{\triangledown\text{–R}}(x, y_1, y_2, y_3, y_4) \equiv \mathsf{Right}^{\triangledown\text{–R}}(x, y_1, y_2, y_3, y_4). \tag{J.13}$$

□

J.3.4 Unpair–Split transformation

Transformation Unpair–Split is shown in Figure J.11.

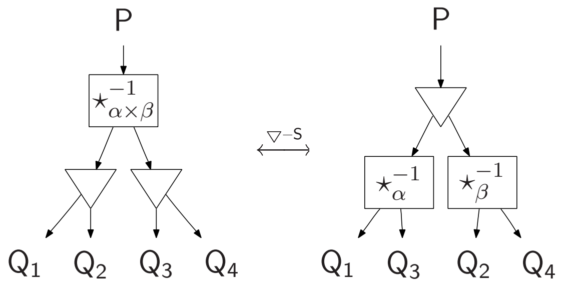


Figure J.11: The Unpair–Split transformation.

Proposition J.11. *Transformation Unpair–Split is sound.*

Proof. We compare the denotations of the task graphs on the left and the right of the

transformation.

$$\begin{aligned}
\mathsf{Left}^{\triangledown\text{–}\mathsf{S}}(x, y_1, y_2, y_3, y_4) &\triangleq \exists t_1, t_2.\ \mathsf{Split}(\star^{-1}_{\alpha\times\beta})(x, t_1, t_2) \wedge \\
&\qquad\quad \mathsf{Unpair}(t_1, y_1, y_2) \wedge \mathsf{Unpair}(t_2, y_3, y_4) \\
&\equiv \exists t_1, t_2.\ (t_1, t_2) = \star^{-1}_{\alpha\times\beta} \wedge \\
&\qquad\quad (y_1, y_2) = t_1 \wedge (y_3, y_4) = t_2 \\
&\equiv ((y_1, y_2), (y_3, y_4)) = \star^{-1}_{\alpha\times\beta}(x);
\end{aligned}$$

$$\begin{aligned}
\mathsf{Right}^{\triangledown\text{–}\mathsf{S}}(x, y_1, y_2, y_3, y_4) &\triangleq \exists t_1, t_2.\ \mathsf{Unpair}(x, t_1, t_2) \wedge \\
&\qquad\quad \mathsf{Split}(\star^{-1}_{\alpha})(t_1, y_1, y_3) \wedge \mathsf{Split}(\star^{-1}_{\beta})(t_2, y_2, y_4) \\
&\equiv \exists t_1, t_2.\ (t_1, t_2) = x \wedge \\
&\qquad\quad (y_1, y_3) = \star^{-1}_{\alpha}(t_1) \wedge (y_2, y_4) = \star^{-1}_{\beta}(t_2) \\
&\equiv \exists t_1, t_2.\ (t_1, t_2) = x \wedge \\
&\qquad\quad ((y_1, y_2), (y_3, y_4)) = \star^{-1}_{\alpha\times\beta}(t_1, t_2) \\
&\equiv ((y_1, y_2), (y_3, y_4)) = \star^{-1}_{\alpha\times\beta}(x).
\end{aligned}$$

Hence,

$$\mathsf{Left}^{\triangledown\text{–}\mathsf{S}}(x, y_1, y_2, y_3, y_4) \equiv \mathsf{Right}^{\triangledown\text{–}\mathsf{S}}(x, y_1, y_2, y_3, y_4). \tag{J.14}$$

□

We summarise the transformations involving pair and unpair tasks denotationally in Figure J.12, and show their n-ary counterparts in Figure J.15.

J.4 Redundancy

As with the original transformations, there is some redundancy in these new transformations; in Figures J.13 and J.14 we show that both Pair–Processing and Unpair–Processing can be expressed in terms of Pair–Unpair and Combine.

J.5 Derivation of transformations involving pair and unpair

In Section 5.6.1.3, we showed that pair and unpair tasks can be expressed in terms of the primitive kinds of task. This led us to question whether the transformations involving pair and unpair tasks defined above in Sections J.1–J.3 can also be derived from the transformations defined for the primitive tasks when we express pair and unpair in this way.

The Pair–Processing, Unpair–Processing and Combine transformations cannot be expressed in terms of the transformations on primitive tasks because they depend solely upon the relationship between a function f and the functions g_α and g_β given in (J.3).

$$
\begin{array}{rcl}
\mathsf{Left}^{\triangle\text{-}\triangledown}(x_1,x_2,y_1,y_2) & \triangleq & \exists t.\ \mathsf{Pair}(x_1,x_2,t) \wedge \mathsf{Unpair}(t,y_1,y_2) \\
\mathsf{Right}^{\triangle\text{-}\triangledown}(x_1,x_2,y_1,y_2) & \triangleq & y_1 = x_1 \wedge y_2 = x_2 \\
\mathsf{Left}^{\triangledown\text{-}\triangle}(x,y) & \triangleq & \exists t_1,t_2.\ \mathsf{Unpair}(x,t_1,t_2) \wedge \mathsf{Pair}(t_1,t_2,y) \\
\mathsf{Right}^{\triangledown\text{-}\triangle}(x,y) & \triangleq & y = x \\
\mathsf{Left}^{\mathsf{Combine}}(x,y) & \triangleq & \exists t_1,t_2,u_1,u_2.\ \mathsf{Unpair}(x,t_1,t_2) \wedge \\
 & & \qquad \mathsf{Proc}(g_\alpha)(t_1,u_1) \wedge \mathsf{Proc}(g_\beta)(t_2,u_2) \wedge \\
 & & \qquad \mathsf{Pair}(u_1,u_2,y) \\
\mathsf{Right}^{\mathsf{Combine}}(x,y) & \triangleq & \mathsf{Proc}(f)(x,y) \\
\mathsf{Left}^{\triangle\text{-}\mathsf{P}}(x_1,x_2,y) & \triangleq & \exists t.\ \mathsf{Pair}(x_1,x_2,t) \wedge \mathsf{Proc}(f)(t,y) \\
\mathsf{Right}^{\triangle\text{-}\mathsf{P}}(x_1,x_2,y) & \triangleq & \exists t_1,t_2.\ \mathsf{Proc}(g_\alpha)(x_1,t_1) \wedge \mathsf{Proc}(g_\beta)(x_2,t_2) \wedge \\
 & & \qquad \mathsf{Pair}(t_1,t_2,y) \\
 & & \text{where } f(a_1,a_2) = (g_\alpha(a_1), g_\beta(a_2)) \\
\mathsf{Left}^{\triangle\text{-}\mathsf{M}}(x_1,x_2,x_3,x_4,y) & \triangleq & \exists t_1,t_2.\ \mathsf{Pair}(x_1,x_2,t_1) \wedge \mathsf{Pair}(x_3,x_4,t_2) \wedge \\
 & & \qquad \mathsf{Merge}(\star_{\alpha\times\beta})(t_1,t_2,y) \\
\mathsf{Right}^{\triangle\text{-}\mathsf{M}}(x_1,x_2,x_3,x_4,y) & \triangleq & \exists t_1,t_2.\ \mathsf{Merge}(\star_\alpha)(x_1,x_3,t_1) \wedge \mathsf{Merge}(\star_\beta)(x_2,x_4,t_2) \wedge \\
 & & \qquad \mathsf{Pair}(t_1,t_2,y) \\
\mathsf{Left}^{\triangle\text{-}\mathsf{R}}(x_1,x_2,y_1,y_2) & \triangleq & \exists t.\ \mathsf{Pair}(x_1,x_2,t) \wedge \mathsf{Rep}(t,y_1,y_2) \\
\mathsf{Right}^{\triangle\text{-}\mathsf{R}}(x_1,x_2,y_1,y_2) & \triangleq & \exists t_1,t_2,t_3,t_4.\ \mathsf{Rep}(x_1,t_1,t_2) \wedge \mathsf{Rep}(x_2,t_3,t_4) \wedge \\
 & & \qquad \mathsf{Pair}(t_1,t_3,y_1) \wedge \mathsf{Pair}(t_2,t_4,y_2) \\
\mathsf{Left}^{\triangle\text{-}\mathsf{S}}(x_1,x_2,y_1,y_2) & \triangleq & \exists t.\ \mathsf{Pair}(x_1,x_2,t) \wedge \mathsf{Split}(\star_{\alpha\times\beta}^{-1})(t,y_1,y_2) \\
\mathsf{Right}^{\triangle\text{-}\mathsf{S}}(x_1,x_2,y_1,y_2) & \triangleq & \exists t_1,t_2,t_3,t_4.\ \mathsf{Split}(\star_{\alpha\times\beta}^{-1})(x_1,t_1,t_2) \wedge \mathsf{Split}(\star_{\alpha\times\beta}^{-1})(x_2,t_3,t_4) \wedge \\
 & & \qquad \mathsf{Pair}(t_1,t_3,y_1) \wedge \mathsf{Pair}(t_2,t_4,y_2) \\
\mathsf{Left}^{\triangledown\text{-}\mathsf{P}}(x,y_1,y_2) & \triangleq & \exists t_1,t_2.\ \mathsf{Unpair}(x,t_1,t_2) \wedge \\
 & & \qquad \mathsf{Proc}(g_\alpha)(t_1,y_1) \wedge \mathsf{Proc}(g_\beta)(t_2,y_2) \\
\mathsf{Right}^{\triangledown\text{-}\mathsf{P}}(x,y_1,y_2) & \triangleq & \exists t.\ \mathsf{Proc}(f)(x,t) \wedge \mathsf{Unpair}(t,y_1,y_2) \\
 & & \text{where } f(a_1,a_2) = (g_\alpha(a_1), g_\beta(a_2)) \\
\mathsf{Left}^{\triangledown\text{-}\mathsf{M}}(x_1,x_2,y_1,y_2) & \triangleq & \exists t.\ \mathsf{Merge}(\star_{\alpha\times\beta})(x_1,x_2,t) \wedge \mathsf{Unpair}(t,y_1,y_2) \\
\mathsf{Right}^{\triangledown\text{-}\mathsf{M}}(x_1,x_2,y_1,y_2) & \triangleq & \exists t_1,t_2,t_3,t_4.\ \mathsf{Unpair}(x_1,t_1,t_2) \wedge \mathsf{Unpair}(x_2,t_3,t_4) \wedge \\
 & & \qquad \mathsf{Merge}(\star_\alpha)(t_1,t_3,y_1) \wedge \mathsf{Merge}(\star_\beta)(t_2,t_4,y_2) \\
\mathsf{Left}^{\triangledown\text{-}\mathsf{R}}(x,y_1,y_2,y_3,y_4) & \triangleq & \exists t_1,t_2.\ \mathsf{Rep}(x,t_1,t_2) \wedge \\
 & & \qquad \mathsf{Unpair}(t_1,y_1,y_2) \wedge \mathsf{Unpair}(t_2,y_3,y_4) \\
\mathsf{Right}^{\triangledown\text{-}\mathsf{R}}(x,y_1,y_2,y_3,y_4) & \triangleq & \exists t_1,t_2.\ \mathsf{Unpair}(x,t_1,t_2) \wedge \\
 & & \qquad \mathsf{Rep}(t_1,y_1,y_3) \wedge \mathsf{Rep}(t_2,y_2,y_4) \\
\mathsf{Left}^{\triangledown\text{-}\mathsf{S}}(x,y_1,y_2,y_3,y_4) & \triangleq & \exists t_1,t_2.\ \mathsf{Split}(\star_{\alpha\times\beta}^{-1})(x,t_1,t_2) \wedge \\
 & & \qquad \mathsf{Unpair}(t_1,y_1,y_2) \wedge \mathsf{Unpair}(t_2,y_3,y_4) \\
\mathsf{Right}^{\triangledown\text{-}\mathsf{S}}(x,y_1,y_2,y_3,y_4) & \triangleq & \exists t_1,t_2.\ \mathsf{Unpair}(x,t_1,t_2) \wedge \\
 & & \qquad \mathsf{Split}(\star_\alpha^{-1})(t_1,y_1,y_3) \wedge \mathsf{Split}(\star_\beta^{-1})(t_2,y_2,y_4)
\end{array}
$$

Figure J.12: Denotations of transformations involving pair and unpair tasks.

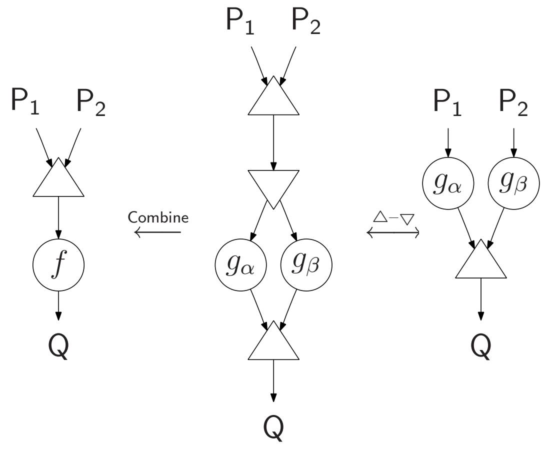


Figure J.13: Derivation of the Pair–Processing transformation.

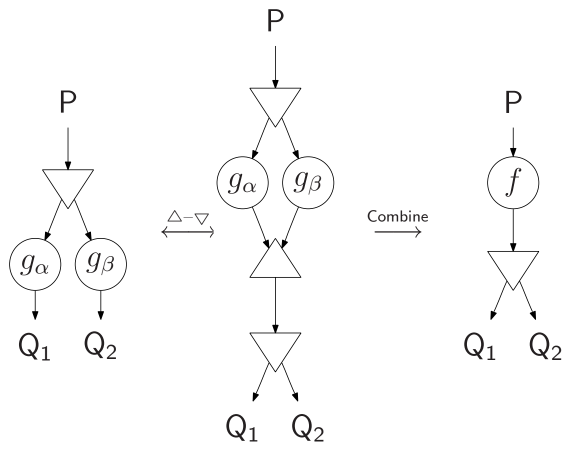


Figure J.14: Derivation of the Unpair–Processing transformation.

$$\begin{aligned}
\mathsf{Left}^{\triangle\text{–}\triangledown}_{n}(x_1,\ldots,x_n,y_1,\ldots,y_n) &\triangleq \exists t.\ \mathsf{Pair}_n(x_1,\ldots,x_n,t) \wedge \mathsf{Unpair}_n(t,y_1,\ldots,y_n)\\
\mathsf{Right}^{\triangle\text{–}\triangledown}_{n}(x_1,\ldots,x_n,y_1,\ldots,y_n) &\triangleq \bigwedge_{i=1}^{n} y_i = x_i\\
\mathsf{Left}^{\triangledown\text{–}\triangle}_{n}(x,y) &\triangleq \exists t_1,\ldots,t_n.\ \mathsf{Unpair}_n(x,t_1,\ldots,t_n) \wedge \mathsf{Pair}(t_1,\ldots,t_n,y)\\
\mathsf{Right}^{\triangledown\text{–}\triangle}_{n}(x,y) &\triangleq y = x\\
\mathsf{Left}^{\mathsf{Combine}}_{n}(x,y) &\triangleq \exists t_1,\ldots,t_n,u_1,\ldots,u_n.\ \mathsf{Unpair}_n(x,t_1,\ldots,t_n) \wedge\\
&\qquad \textstyle\bigwedge_{i=1}^{n} \mathsf{Proc}(g_i)(t_i,u_i) \wedge\\
&\qquad \mathsf{Pair}_n(u_1,\ldots,u_n,y)\\
\mathsf{Right}^{\mathsf{Combine}}_{n}(x,y) &\triangleq \mathsf{Proc}(f)(x,y)\\
\mathsf{Left}^{\triangle\text{–}\mathsf{P}}_{n}(x_1,\ldots,x_n,y) &\triangleq \exists t.\ \mathsf{Pair}_n(x_1,\ldots,x_n,t) \wedge \mathsf{Proc}(f)(t,y)\\
\mathsf{Right}^{\triangle\text{–}\mathsf{P}}_{n}(x_1,\ldots,x_n,y) &\triangleq \exists t_1,\ldots,t_n.\ \mathsf{Pair}_n(t_1,\ldots,t_n,y) \wedge \bigwedge_{i=1}^{n} \mathsf{Proc}(g_i)(x_i)\\
&\quad \text{where } f(a_1,\ldots,a_n) = (g_1(a_1),\ldots,g_n(a_n))\\
\mathsf{Left}^{\triangle\text{–}\mathsf{M}}_{m,n}(x_{1,1},\ldots,x_{m,n},y) &\triangleq \exists t_1,\ldots,t_n.\ \mathsf{Merge}_n(\star_{\alpha_1\times\ldots\times\alpha_n})(t_1,\ldots,t_n,y) \wedge\\
&\qquad \textstyle\bigwedge_{i=1}^{n} \mathsf{Pair}_m(x_{1,i},\ldots,x_{m,i},t_i)\\
\mathsf{Right}^{\triangle\text{–}\mathsf{M}}_{m,n}(x_{1,1},\ldots,x_{m,n},y) &\triangleq \exists t_1,\ldots,t_m.\ \mathsf{Pair}_m(t_1,\ldots,t_m,y) \wedge\\
&\qquad \textstyle\bigwedge_{i=1}^{m} \mathsf{Merge}_n(\star_{\alpha_i})(x_{i,1},\ldots,x_{i,n},t_i)\\
\mathsf{Left}^{\triangle\text{–}\mathsf{R}}_{m,n}(x_1,\ldots,x_n,y_1,\ldots,y_m) &\triangleq \exists t.\ \mathsf{Pair}_n(x_1,\ldots,x_n,t) \wedge \mathsf{Rep}_m(t,y_1,\ldots,y_m)\\
\mathsf{Right}^{\triangle\text{–}\mathsf{R}}_{m,n}(x_1,\ldots,x_n,y_1,\ldots,y_m) &\triangleq \exists t_{1,1},\ldots,t_{m,n}.\ \textstyle\bigwedge_{i=1}^{n} \mathsf{Rep}_m(x_i,t_{1,i},\ldots,t_{m,i}) \wedge\\
&\qquad \textstyle\bigwedge_{i=1}^{m} \mathsf{Pair}_n(t_{i,1},\ldots,t_{i,n},y_i)\\
\mathsf{Left}^{\triangle\text{–}\mathsf{S}}_{m,n}(x_1,\ldots,x_n,y_1,\ldots,y_m) &\triangleq \exists t.\ \mathsf{Pair}_n(x_1,\ldots,x_n,t) \wedge \mathsf{Split}_m(\star^{-1}_{\alpha_1\times\ldots\times\alpha_n})(t,y_1,\ldots,y_m)\\
\mathsf{Right}^{\triangle\text{–}\mathsf{S}}_{m,n}(x_1,\ldots,x_n,y_1,\ldots,y_m) &\triangleq \exists t_{1,1},\ldots,t_{m,n}.\ \textstyle\bigwedge_{i=1}^{n} \mathsf{Split}_m(\star^{-1}_{\alpha_1\times\ldots\times\alpha_n})(x_i,t_{1,i},\ldots,t_{m,i}) \wedge\\
&\qquad \textstyle\bigwedge_{i=1}^{m} \mathsf{Pair}_n(t_{i,1},\ldots,t_{i,n},y_i)\\
\mathsf{Left}^{\triangledown\text{–}\mathsf{P}}_{n}(x,y_1,\ldots,y_n) &\triangleq \exists t_1,\ldots,t_n.\ \mathsf{Unpair}_n(x,t_1,\ldots,t_n) \wedge \bigwedge_{i=1}^{n} \mathsf{Proc}(g_i)(t_i,y_i)\\
\mathsf{Right}^{\triangledown\text{–}\mathsf{P}}_{n}(x,y_1,\ldots,y_n) &\triangleq \exists t.\ \mathsf{Proc}(f)(x,t) \wedge \mathsf{Unpair}_n(t,y_1,\ldots,y_n)\\
&\quad \text{where } f(a_1,\ldots,a_n) = (g_1(a_1),\ldots,g_n(a_n))\\
\mathsf{Left}^{\triangledown\text{–}\mathsf{M}}_{m,n}(x_1,\ldots,x_m,y_1,\ldots,y_n) &\triangleq \exists t.\ \mathsf{Merge}_m(\star_{\alpha_1\times\ldots\times\alpha_n})(x_1,\ldots,x_m,t) \wedge\\
&\qquad \mathsf{Unpair}_n(t,y_1,\ldots,y_n)\\
\mathsf{Right}^{\triangledown\text{–}\mathsf{M}}_{m,n}(x_1,\ldots,x_m,y_1,\ldots,y_n) &\triangleq \exists t_{1,1},\ldots,t_{m,n}.\ \textstyle\bigwedge_{i=1}^{m} \mathsf{Unpair}_n(x_i,t_{i,1},\ldots,t_{i,n}) \wedge\\
&\qquad \textstyle\bigwedge_{i=1}^{n} \mathsf{Merge}_m(\star_{\alpha_1\times\ldots\times\alpha_n})(t_{1,i},\ldots,t_{m,i},y_i)\\
\mathsf{Left}^{\triangledown\text{–}\mathsf{R}}_{m,n}(x,y_{1,1},\ldots,y_{m,n}) &\triangleq \exists t_1,\ldots,t_m.\ \mathsf{Rep}_m(x,t_1,\ldots,t_m) \wedge\\
&\qquad \textstyle\bigwedge_{i=1}^{m} \mathsf{Unpair}_n(t_i,y_{i,1},\ldots,y_{i,n})\\
\mathsf{Right}^{\triangledown\text{–}\mathsf{R}}_{m,n}(x,y_{1,1},\ldots,y_{m,n}) &\triangleq \exists t_1,\ldots,t_n.\ \mathsf{Unpair}_n(x,t_1,\ldots,t_n) \wedge\\
&\qquad \textstyle\bigwedge_{i=1}^{n} \mathsf{Rep}_m(t_i,y_{1,i},\ldots,y_{m,i})\\
\mathsf{Left}^{\triangledown\text{–}\mathsf{S}}_{m,n}(x,y_{1,1},\ldots,y_{m,n}) &\triangleq \exists t_1,\ldots,t_m.\ \mathsf{Split}_m(\star^{-1}_{\alpha_1\times\ldots\times\alpha_n})(x,t_1,\ldots,t_m) \wedge\\
&\qquad \textstyle\bigwedge_{i=1}^{m} \mathsf{Unpair}_n(t_i,y_{i,1},\ldots,y_{i,n})\\
\mathsf{Right}^{\triangledown\text{–}\mathsf{S}}_{m,n}(x,y_{1,1},\ldots,y_{m,n}) &\triangleq \exists t_1,\ldots,t_n.\ \mathsf{Unpair}_n(x,t_1,\ldots,t_n) \wedge\\
&\qquad \textstyle\bigwedge_{i=1}^{n} \mathsf{Split}_m(\star^{-1}_{\alpha_1\times\ldots\times\alpha_n})(t_i,y_{1,i},\ldots,y_{m,i})
\end{aligned}$$

Figure J.15: Denotations of n-ary transformations involving pair and unpair tasks.

The Pair–Unpair and Unpair–Pair transformations cannot be expressed solely in terms of the transformations on primitive tasks—see Figures J.16 and J.17. The derivation of the Pair–Unpair transformation relies on the fact that π_α (used in expressing an unpair task) is a right-inverse for p_α (used in expressing a pair task), and likewise for π_β and p_β.

Proposition J.12. *π_α is a right-inverse of p_α.*

Proof. For arbitrary a,

$$\begin{array}{rcll} \pi_\alpha(p_\alpha(a)) & = & \pi_\alpha(a, 0_\beta) & \text{by definition of } p_\alpha\text{, (5.41)} \\ & = & a & \text{by definition of } \pi_\alpha\text{, (5.45).} \end{array}$$

□

The derivation of the Unpair–Pair transformation relies on p_α being a right-inverse for π_α for the set of values which may be delivered from the $s_{\alpha\times\beta}$ split task, and likewise for p_β and π_β.

Proposition J.13. *For the first element of pairs returned from $s_{\alpha\times\beta}$, p_α is a right-inverse of π_α.*

Proof. From the definition of $s_{\alpha\times\beta}$, (5.44), the first element of the pair returned will be $(a, 0_\beta)$ for some a.

$$\begin{array}{rcll} p_\alpha(\pi_\alpha(a, 0_\beta)) & = & p_\alpha(a) & \text{by definition of } \pi_\alpha\text{, (5.45)} \\ & = & (a, 0_\beta) & \text{by definition of } p_\alpha\text{, (5.41).} \end{array}$$

□

The derivations in Figures J.18–J.20 show that we can express the Pair–Merge, Pair–Replication and Pair–Split transformations in terms of the transformations on the primitive tasks; and those in Figures J.21–J.23[1] show the same for Unpair–Merge, Unpair–Replication and Unpair–Split.

[1]There is a question over the validity of J.23. It relies upon interchanging the order of different split tasks defined for a given type.

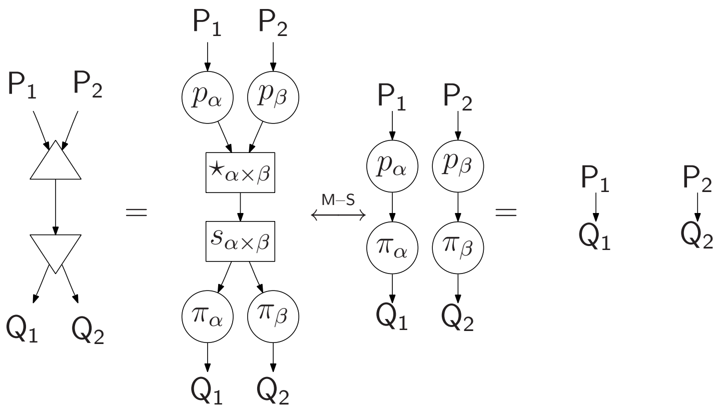


Figure J.16: Derivation of the Pair–Unpair transformation. The final step follows from the fact that π_α is a right-inverse for p_α, and similarly for π_β and p_β.

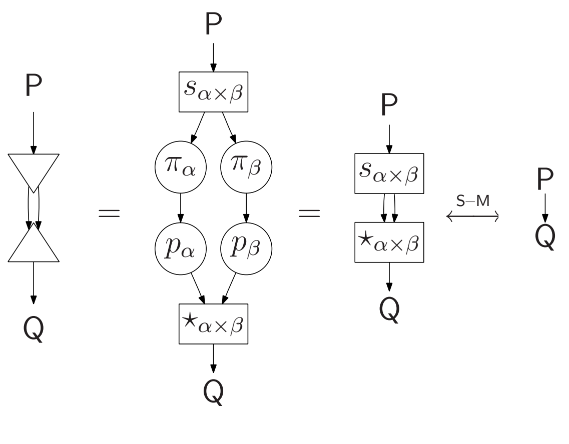


Figure J.17: Derivation of the Unpair–Pair transformation. The second step follows from the fact that $s_{\alpha\times\beta}$ will deliver to the π_α task an item of type α paired with 0_β, and $\forall a.\ p_\alpha(\pi_\alpha(a, 0_\beta)) = (a, 0_\beta)$, and the analogous fact for β.

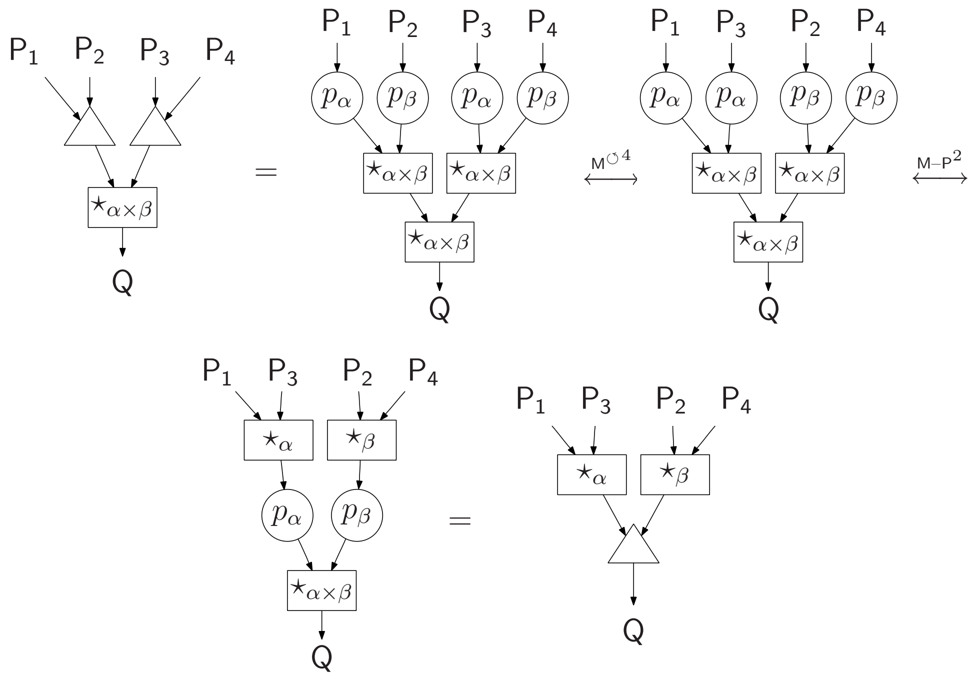


Figure J.18: Derivation of the Pair–Merge transformation. The second step proceeds as shown in Figure 5.14.

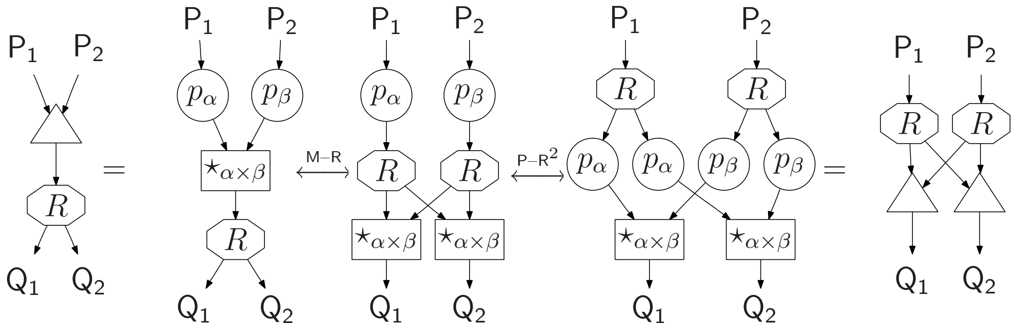


Figure J.19: Derivation of the Pair–Replication transformation.

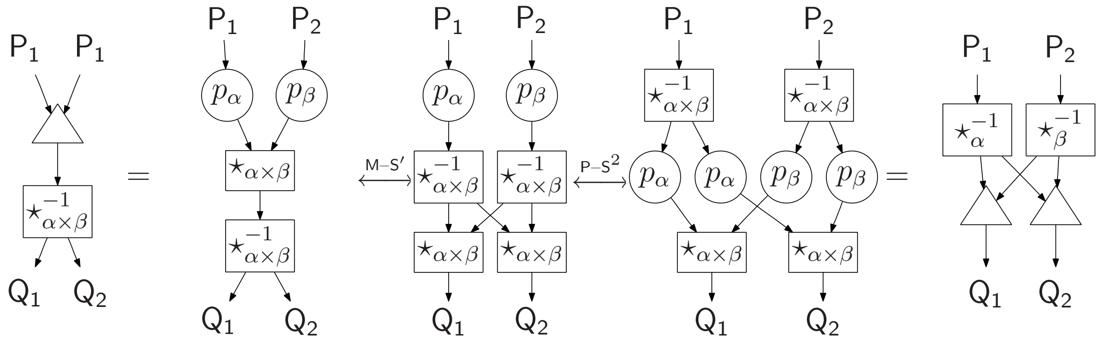

P₁ P₁
P₁ P₂
$\star^{-1}_{\alpha\times\beta}$
Q₁ Q₂
=
p_α p_β
$\star_{\alpha\times\beta}$
M–S′
P–S²
$\star^{-1}_\alpha$ $\star^{-1}_\beta$

Figure J.20: Derivation of the Pair–Split transformation. The second step involves the use of the Merge–Split′ transformation.

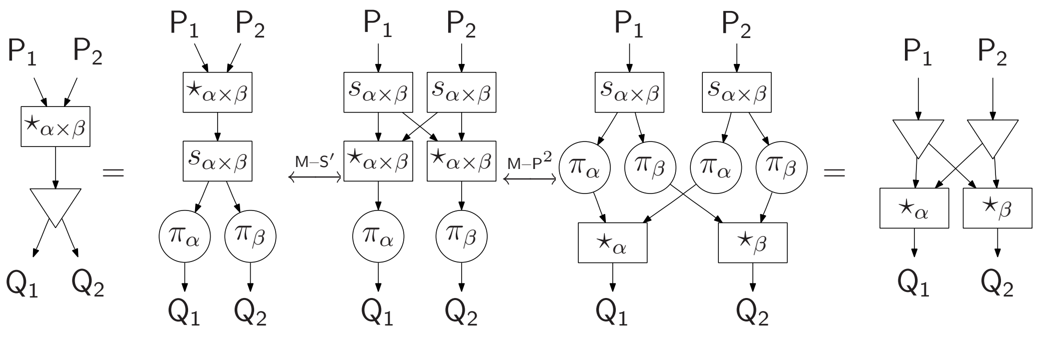


Figure J.21: Derivation of the Unpair–Merge transformation. The second step involves the use of the Merge–Split′ transformation.

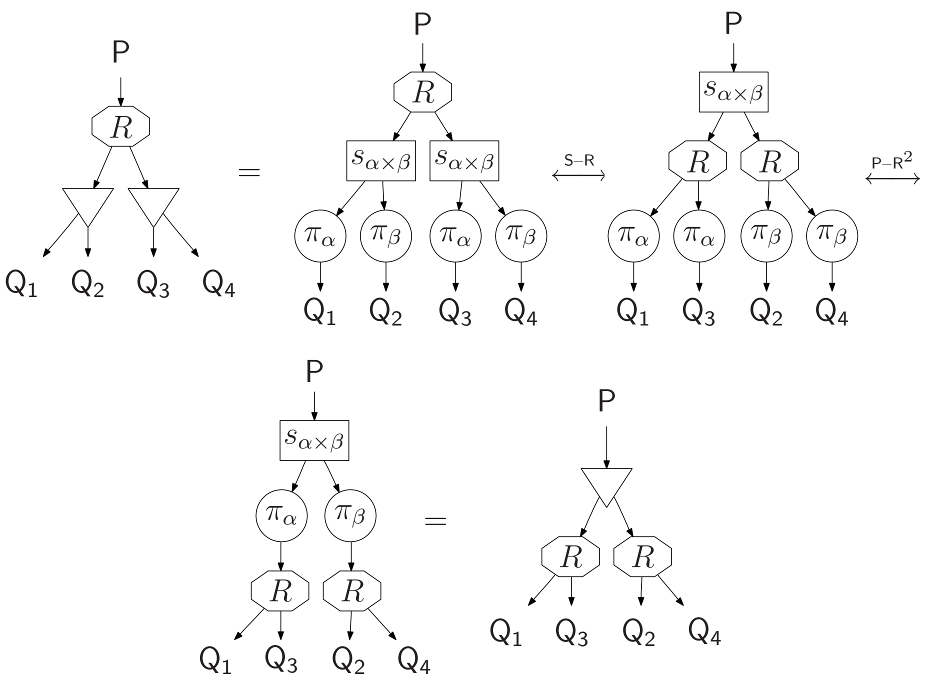


Figure J.22: Derivation of the Unpair–Replication transformation.

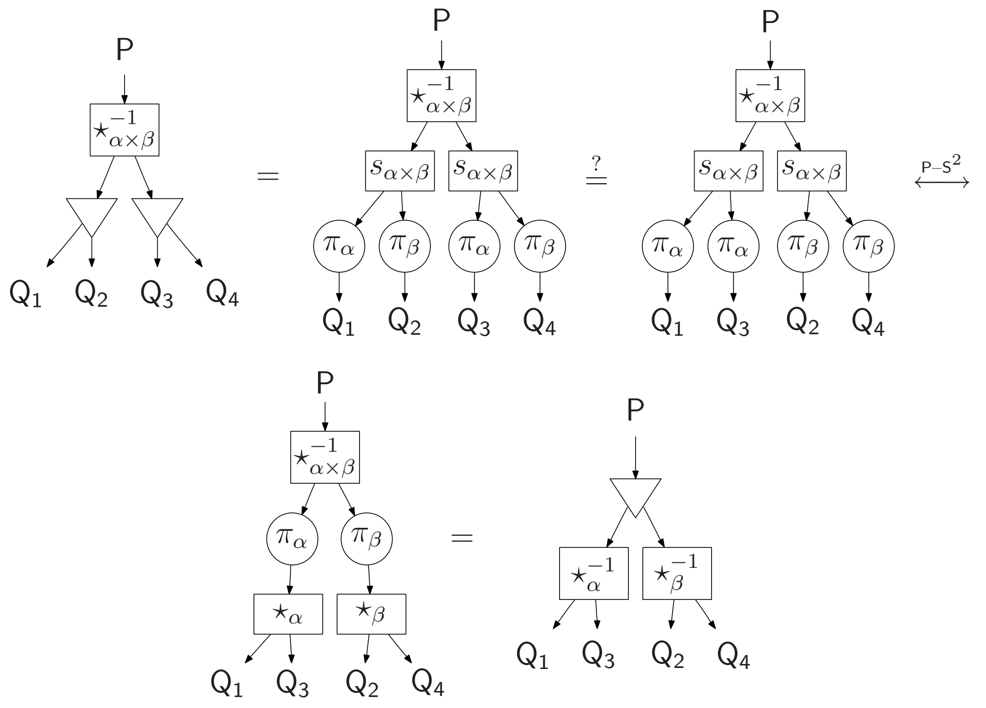


Figure J.23: Derivation of the Unpair–Split transformation. The validity of the second step, involving the exchange of different split tasks on a given datatype, is questionable.

References

[1] Mohamad Afshar. An Open Parallel Architecture for Data-intensive Applications. PhD thesis. Technical Report UCAM-CL-TR-459, University of Cambridge, Computer Laboratory, July 1999. (Cited on pages 109 and 115.)

[2] Kemal Akkaya and Mohamed Younis. A survey on routing protocols for wireless sensor networks. *Ad Hoc Networks*, 3(3):325–349, May 2005. (Cited on pages 198 and 200.)

[3] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: A survey. *Computer Networks*, 38:393–422, 2002. (Cited on pages 100, 197 and 198.)

[4] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services: Concepts, Architectures and Applications*. Springer-Verlag, Berlin / Heidelberg, 2004. (Cited on pages 35 and 100.)

[5] AMI-C. AMI-C use cases. Available from `http://www.ami-c.org/`, January 2003. (Cited on page 16.)

[6] Khalil Amiri, David Petrou, Gregory R. Ganger, and Garth A. Gibson. Dynamic function placement for data-intensive cluster computing. In *Proceedings of the USENIX Annual Technical Conference*, pages 307–322, San Diego, CA, USA, June 2000. (Cited on page 45.)

[7] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996. (Cited on page 217.)

[8] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic, and Sanjiva Weerawarana. Business Process Execution Language for Web Services, version 1.1. `http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/`, May 2003. (Cited on page 35.)

[9] Mario Antonioletti, Malcolm Atkinson, Rob Baxter, Andrew Borley, Neil P. Chue Hong, Brian Collins, Neil Hardman, Alastair C. Hume, Alan Knox, Mike Jackson, Amy Krause, Simon Laws, James Magowan, Norman W. Paton, Dave Pearson, Tom Sugden, Paul Watson, and Martin Westhead. The design and implementation of grid database services in OGSA-DAI. *Concurrency and Computation: Practice and Experience*, 17:357–376, Feb–Apr 2005. (Cited on pages 40 and 100.)

[10] Apple Inc. Bonjour overview. Available from `http://developer.apple.com/documentation/Cocoa/Conceptual/NetServices/NetServices.pdf`, May 2006. (Cited on page 199.)

[11] Joe Armstrong. *Programming Erlang: Software for a Concurrent World.* The Pragmatic Bookshelf, Raleigh, North Carolina, USA, July 2007. (Cited on page 36.)

[12] A. Arora, P. Dutta, S. Bapat, V. Kulathumani, H. Zhang, V. Naik, V. Mittal, H. Cao, M. Demirbas, M. Gouda, Y. Choi, T. Herman, S. Kulkarni, U. Arumugam, M. Nesterenko, A. Vora, and M. Miyashita. A line in the sand: A wireless sensor network for target detection, classification and tracking. *Computer Networks*, 46(5):605–634, December 2004. (Cited on page 198.)

[13] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David Patterson, and Kathy Yelick. Cluster I/O with River: Making the fast case common. In *Proceedings of the 6th Workshop on Input/Output in Parallel and Distributed Systems (IOPADS '99)*, pages 10–22, Atlanta, GA, USA, May 1999. (Cited on page 39.)

[14] Volkan Arslan, Patrick Eugster, Piotr Nienaltowski, and Sebastien Vaucouleur. *SCOOP – Concurrency Made Easy*, volume 4028 of *LNCS*, pages 82–102. Springer-Verlag, September 2006. (Cited on page 206.)

[15] Franz Aurenhammer. Voronoi diagrams – a survey of a fundamental geometric data structure. *ACM Computing Surveys*, 23(3):345–405, September 1991. (Cited on pages 78 and 79.)

[16] Jean Bacon, Ken Moody, John Bates, Richard Hayton, Chaoying Ma, Andrew McNeil, Oliver Seidel, and Mark Spiteri. Generic support for distributed applications. *IEEE Computer*, 33(3):68–76, March 2000. (Cited on page 38.)

[17] J. C. M. Baeten. A brief history of process algebra. *Theoretical Computer Science*, 335:131–146, 2005. (Cited on page 210.)

[18] Henri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992. (Cited on page 218.)

[19] François Bancilhon, Ted Briggs, Setrag Khoshafian, and Patrick Valduriez. FAD, a powerful and simple database language. In *VLDB '87: Proceedings of the 13th International Conference on Very Large Data Bases*, pages 97–105, Brighton, UK, September 1987. Morgan Kaufmann. (Cited on page 107.)

[20] Amnon Barak and Oren La'adan. The MOSIX multicomputer operating system for high performance cluster computing. *Future Generation Computer Systems*, 13(4–5):361–372, March 1998. (Cited on page 218.)

[21] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 164–177, Bolton Landing, NY, USA, October 2003. (Cited on page 33.)

[22] Rimon Barr, John C. Bicket, Daniel S. Dantas, Bowei Du, T. W. Danny Kim, Bing Zhou, and Emin Gün Sirer. On the need for system-level support for ad hoc and sensor networks. *ACM SIGOPS Operating Systems Review*, 36(2):1–5, 2002. (Cited on pages 46 and 223.)

[23] Marc Bechler, Walter J. Franz, and Lars Wolf. Mobile internet access in FleetNet. In *13. Fachtagung Kommunikation in Verteilten Systemen: Kurzbeiträge, Praxisberichte und Workshop E-Learning*, pages 107–118, Leipzig, Germany, April 2003. VDE Verlag. (Cited on page 193.)

[24] Alastair R. Beresford. Location privacy in ubiquitous computing. PhD thesis. Technical Report UCAM-CL-TR-612, University of Cambridge, Computer Laboratory, January 2005. (Cited on page 191.)

[25] Pierpaolo Bergamo, Daniela Maniezzo, Kung Yao, Matteo Cesana, Giovanni Pau, Mario Gerla, and Don Whiteman. IEEE802.11 wireless network under aggressive mobility scenarios. In *International Telemetry Conference ITC/USA2003*, Las Vegas, NV, USA, October 2003. (Cited on page 193.)

[26] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, 284(5):28–37, May 2001. (Cited on page 101.)

[27] Jan Beutel, Oliver Kasten, Friedemann Mattern, Kay Römer, Frank Siegemund, and Lothar Thiele. Prototyping wireless sensor network applications with BTnodes. In *Proceedings of the 1st European Workshop on Wireless Sensor Networks (EWSN 2004)*, volume 2920 of *LNCS*, pages 323–338, Berlin, Germany, January 2004. (Cited on page 195.)

[28] R. S. Bird. Lectures on constructive functional programming. In M. Broy, editor, *Constructive Methods in Computing Science: Proceedings of the NATO Advanced Study Institute*, volume 55 of *NATO ASI Series: Computer and Systems Sciences*, Marktoberdorf, Federal Republic of Germany, Jul–Aug 1988. Springer-Verlag. (Cited on page 216.)

[29] Roberto Bisiani and Mosur Ravishankar. PLUS: A distributed shared-memory system. *ACM SIGARCH Computer Architecture News*, 18:115–124, June 1990. (Cited on page 217.)

[30] Jeremy J. Blum, Azim Eskandarian, and Lance J. Hoffman. Challenges of intervehicle *Ad Hoc* networks. *IEEE Transactions on Intelligent Transportation Systems*, 5(4):347–351, December 2004. (Cited on page 25.)

[31] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. Querying the physical world. *IEEE Personal Communications*, 7(5):10–15, October 2000. (Cited on pages 48 and 197.)

[32] Philippe Bonnet, Johnannes Gehrke, and Praveen Seshadri. Towards sensor database systems. In *Proceedings of the 2nd International Conference on Mobile Data Management*, volume 1987 of *LNCS*, pages 3–14, 2001. (Cited on page 49.)

[33] Cristian Borcea, Chalermek Intanagonwiwat, Porlin Kang, Ulrich Kremer, and Liviu Iftode. Spatial programming using smart messages: Design and implementation. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS)*, pages 690–699, Tokyo, Japan, March 2004. IEEE Computer Society. (Cited on pages 51 and 53.)

[34] Athanassios Boulis, Chih-Chieh Han, and Mani B. Srivastava. Design and implementation of a framework for efficient and programmable sensor networks. In *Proceedings of MobiSys 2003: The First International Conference on Mobile Systems, Applications, and Services*, pages 187–200, San Francisco, CA, USA, May 2003. ACM. (Cited on pages 46, 47 and 52.)

[35] Athanassios Boulis and Mani B. Srivastava. A framework for efficient and programmable sensor networks. In *Proceedings of the IEEE Conference on Open Architectures and Network Programming*, pages 117–128, June 2002. (Cited on pages 51 and 198.)

[36] David Braginsky and Deborah Estrin. Rumor routing algorithm for sensor networks. In *Proceedings of the 1st ACM International Workshop on Wireless Sensor Networks and Applications (WSNA '02)*, pages 22–31, Atlanta, GA, USA, September 2002. (Cited on page 201.)

[37] Linda Briesemeister, Lorenz Schäfers, and Günter Hommel. Disseminating messages among highly mobile hosts based on inter-vehicle communication. In *IEEE Intelligent Vehicles Symposium*, pages 522–527, October 2000. (Cited on page 26.)

[38] R. M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977. (Cited on page 206.)

[39] V. Bychkovsky, K. Chen, M. Goraczko, H. Hu, B. Hull, A. Miu, E. Shih, Y. Zhang, H. Balakrishnan, and S. Madden. Data management in the CarTel mobile sensor

computing system. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 730–732, Chicago, IL, USA, 2006. (Cited on page 91.)

[40] Cambridgeshire County Council. The 2006 traffic monitoring report. Available from `http://www.cambridgeshire.gov.uk/transport/monitoring/network/traffic+monitoring+report.htm`, 2006. (Cited on page 89.)

[41] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989. (Cited on page 38.)

[42] Thomas L. Casavant and Jon G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering*, 14(2):141–154, February 1988. (Cited on page 158.)

[43] Alberto Cerpa, Jeremy Elson, Deborah Estrin, Lewis Girod, Michael Hamilton, and Jerry Zhao. Habitat monitoring: Application driver for wireless communications technology. *ACM SIGCOMM Computer Communication Review*, 31(2 supp.):20–41, April 2001. (Cited on page 197.)

[44] Fred B. Chambers, David A. Duce, and Gillian P. Jones, editors. *Distributed Computing*, volume 20 of *APIC Studies in Data Processing*. Academic Press, 1984. (Cited on pages 30, 277, 281 and 285.)

[45] Surajit Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the 17th ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems (PODS '98)*, pages 34–43, Seattle, WA, USA, 1998. (Cited on page 143.)

[46] Guangyu Chen, Byung-Tae Kang, Mahmut Kandemir, Narayanan Vijaykrishnan, Mary Jane Irwin, and Rajarathnam Chandramouli. Studying energy trade offs in offloading computation/compilation in Java-enabled mobile devices. *IEEE Transactions on Parallel and Distributed Systems*, 15(9):795–809, September 2004. (Cited on page 45.)

[47] Ioan Chisalita and Nahid Shahmehri. A peer-to-peer approach to vehicular communication for the support of traffic safety applications. In *IEEE 5th International Conference on Intelligent Transportation Systems*, pages 336–341, Singapore, September 2002. (Cited on page 27.)

[48] Howie Choset and Keiji Nagatani. Topological simultaneous localization and mapping (SLAM): toward exact localization without explicit localization. *IEEE Transactions on Robotics and Automation*, 17(2):125–137, April 2001. (Cited on page 79.)

[49] David N. Cottingham and Jonathan J. Davies. A vision for wireless access on the road network. In *Proceedings of the 4th International Workshop on Intelligent Transportation (WIT 2007)*, pages 25–30, Hamburg, Germany, March 2007. Technische Universität Hamburg-Harburg. (Cited on page 16.)

[50] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a realistic model of parallel computation. *ACM SIGPLAN Notices*, 28(7):1–12, July 1993. (Cited on page 207.)

[51] David Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware / Software Approach*. Morgan Kaufmann, 1997. (Cited on page 208.)

[52] Raymond Cunningham and Vinny Cahill. System support for smart cars. In *9th ACM SIGOPS European Workshop*, Kolding, Denmark, September 2000. ACM. (Cited on page 26.)

[53] Karl Czajkowski, Ian Foster, Nick Karonis, Carl Kesselman, Stuart Martin, Warren Smith, and Steven Tuecke. A resource management architecture for metacomputing systems. In *Job Scheduling Strategies for Parallel Processing (JSSPP '98)*, volume 1459 of *LNCS*, pages 62–82. Springer, 1998. (Cited on page 37.)

[54] Jonathan J. Davies, Alastair R. Beresford, and Andy Hopper. Scalable, distributed, real-time map generation. *IEEE Pervasive Computing*, 5(4):47–54, Oct–Dec 2006. (Cited on page 17.)

[55] Peter Day, Jianpang Wu, and Neil Poulton. Beyond real time. *ITS International*, 12(6):55–56, Nov–Dec 2006. (Cited on page 17.)

[56] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI '04)*, San Francisco, CA, USA, December 2004. (Cited on pages 40, 105 and 145.)

[57] Marios D. Dikaiakos, Tamer Nadeem, Saif Iqbal, and Liviu Iftode. VITP: An information transfer protocol for vehicular computing. In *Proceedings of the 2nd ACM International Workshop on Vehicular Ad Hoc Networks (VANET 2005)*, pages 30–39, Cologne, Germany, August 2005. (Cited on page 101.)

[58] Sandor Dornbush and Anupam Joshi. StreetSmart traffic: Discovering and disseminating automobile congestion using VANET's. In *Proceedings of the 65th IEEE Vehicular Technology Conference (VTC Spring 2007)*, pages 11–15, Dublin, Ireland, April 2007. (Cited on page 29.)

[59] Florian Dötzer, Florian Kohlmayer, Timo Kosch, and Markus Strassberger. Secure communication for intersection assistance. In *Proceedings of the 2nd International Workshop on Intelligent Transportation (WIT 2005)*, Hamburg, Germany, March 2005. Technische Universität Hamburg-Harburg. (Cited on page 17.)

[60] D. H. Douglas and T. K. Peucker. Algorithms for the reduction of the number of points required to represent a line or its caricature. *The Canadian Cartographer*, 10(2):112–122, 1973. (Cited on page 92.)

[61] Ralph Duncan. A survey of parallel computer architectures. *IEEE Computer*, 23(2):5–16, February 1990. (Cited on page 30.)

[62] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. Contiki – a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, pages 455–462, November 2004. (Cited on page 54.)

[63] Vladimir Dyo and Cecilia Mascolo. Efficient node discovery in mobile wireless sensor networks. In *Proceedings of the 4th International Conference on Distributed Computing in Sensor Systems (DCOSS 2008)*, volume 5067 of *LNCS*, pages 478–485, June 2008. (Cited on page 199.)

[64] S. B. Eisenman, E. Miluzzo, N. D. Lane, R. A. Peterson, G-S. Ahn, and A. T. Campbell. The BikeNet mobile sensing system for cyclist experience mapping. In *Proceedings of the 5th ACM Conference on Embedded Networked Sensor Systems (SenSys 07)*, pages 87–101, Sydney, Australia, November 2007. ACM Press, New York, NY, USA. (Cited on page 28.)

[65] Alberto Elfes. Using occupancy grids for mobile robot perception and navigation. *IEEE Computer*, pages 46–57, June 1989. (Cited on page 76.)

[66] Robert Ennals, Richard Sharp, and Alan Mycroft. Linear types for packet processing. In *Programming Languages and Systems: 13th European Symposium on Programming (ESOP 2004)*, volume 2986 of *LNCS*, pages 204–218, 2004. (Cited on pages 143 and 222.)

[67] Robert Ennals, Richard Sharp, and Alan Mycroft. Task partitioning for multi-core network processors. In *Proceedings of the 14th International Conference on Compiler Construction*, volume 3443 of *LNCS*, pages 76–90. Springer-Verlag, April 2005. (Cited on pages 143 and 222.)

[68] Jakob Eriksson, Lewis Girod, Bret Hull, Ryan Newton, Samuel Madden, and Hari Balakrishnan. The pothole patrol: Using a mobile sensor network for road surface monitoring. In *Proceedings of the 6th International Conference on Mobile Systems, Applications and Services (MobiSys '08)*, pages 29–39, Breckenridge, CO, USA, 2008. (Cited on page 28.)

[69] Deborah Estrin, Ramesh Govindan, John Heidemann, and Satish Kumar. Next century challenges: Scalable coordination in sensor networks. In *Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking*, pages 263–270, Seattle, WA, USA, 1999. (Cited on page 197.)

[70] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35(2):114–131, June 2003. (Cited on pages 36 and 38.)

[71] Mbou Eyole-Monono. *Energy-Efficient Sentient Computing*. PhD thesis, Cambridge University, July 2008. (Cited on page 202.)

[72] David Fernández-Baca. Allocating modules to processors in a distributed system. *IEEE Transactions on Software Engineering*, 15(11):1427–1436, November 1989. (Cited on page 158.)

[73] Jason Flinn, Dushyanth Narayanan, and M. Satyanarayanan. Self-tuned remote execution for pervasive computing. In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS)*, pages 61–66, Schloss Elmau, Oberbayern, Germany, May 2001. IEEE Press. (Cited on page 45.)

[74] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, September 1972. (Cited on page 30.)

[75] Chien-Liang Fok, Gruia-Catalin Roman, and Chenyang Lu. Rapid development and flexible deployment of adaptive wireless sensor network applications. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems (ICDCS '05)*, pages 653–662, Washington, DC, USA, 2005. IEEE Computer Society. (Cited on page 53.)

[76] Stephen Fortune. A sweepline algorithm for Voronoi diagrams. In *Proceedings of the Second Annual ACM SIGACT/SIGGRAPH Symposium on Computational Geometry*, pages 313–322. ACM Press, June 1986. (Cited on page 175.)

[77] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the 10th Annual ACM Symposium on Theory of Computing*, pages 114–118, San Diego, CA, USA, 1978. ACM. (Cited on page 207.)

[78] Ian Foster and Carl Kesselman. *The Grid: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann, 2nd edition, 2004. (Cited on pages 32 and 100.)

[79] Ian Foster, Carl Kesselman, Jeffrey M. Nick, and Steven Tuecke. The physiology of the grid. Technical report, The Globus Project, December 2004. (Cited on page 37.)

[80] Michael J. Franklin, Björn Thór Jónsson, and Donald Kossmann. Performance tradeoffs for client-server query processing. *ACM SIGMOD Record*, 25:149–160, June 1996. (Cited on page 44.)

[81] Eric Freeman, Susanne Hupfer, and Ken Arnold. *JavaSpaces Principles, Patterns and Practice.* Addison-Wesley, 1999. (Cited on page 38.)

[82] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, May 1998. (Cited on page 39.)

[83] Holger Füßler, Martin Mauve, Hannes Hartenstein, Michael Käsemann, and Dieter Vollmer. A comparison of routing strategies for vehicular ad hoc networks. Technical Report TR-02-003, Department of Computer Science, University of Mannheim, July 2002. (Cited on pages 25 and 26.)

[84] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software: Practice and Experience*, 30(11):1203–1233, September 2000. (Cited on page 160.)

[85] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI 2003)*, pages 1–11, San Diego, CA, USA, June 2003. (Cited on page 46.)

[86] Johannes Gehrke and Samuel Madden. Query processing in sensor networks. *IEEE Pervasive Computing*, 3(1):46–55, Jan–Mar 2004. (Cited on page 49.)

[87] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. A users' guide to PICL: A portable instrumented communication library. Technical Report ORNL/TM-11616, Oak Ridge National Laboratory, TN, USA, October 1990. (Cited on page 36.)

[88] Andrew S. Glassner, editor. *An Introduction to Ray Tracing*. Morgan Kaufmann, San Francisco, CA, USA, 1989. (Cited on page 163.)

[89] J. R. W. Glauert. High level dataflow programming. In Chambers et al. [44], chapter 4, pages 43–53. (Cited on page 214.)

[90] Michael I. Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2006)*, pages 151–162, San Jose, CA, USA, October 2006. ACM. (Cited on page 143.)

[91] Sergei Gorlatch. *Stages and Transformations in Parallel Programming*, pages 147–162. IOS Press, 1996. (Cited on page 216.)

[92] Sergei Gorlatch. Systematic efficient parallelization of scan and other list homomorphisms. In *Proceedings of the 2nd International Euro-Par Conference on Parallel Processing–Volume II*, volume 1124 of *LNCS*, pages 401–408. Springer-Verlag, 1996. (Cited on page 216.)

[93] Andrew S. Grimshaw. Easy-to-use object-oriented parallel processing with Mentat. Technical Report CS-92-32, University of Virginia, October 1992. (Cited on page 219.)

[94] Marco Gruteser and Dirk Grunwald. Anonymous usage of location-based services through spatial and temporal cloaking. In *Proceedings of 1st ACM/USENIX International Conference on Mobile Systems, Applications and Services (MobiSys)*, pages 31–42, San Francisco, CA, USA, May 2003. ACM Press. (Cited on page 17.)

[95] Xiaohui Gu, Klara Nahrstedt, Alan Messer, Ira Greenberg, and Dejan Milojicic. Adaptive offloading for pervasive computing. *IEEE Pervasive Computing*, 3(3):66–73, July–September 2004. (Cited on page 224.)

[96] Alexander Hagin, Gabriel Dermler, Kurt Rothermel, and Gennadij Shchemelev. Distributed multimedia application configuration management. *IEEE Transactions on Parallel and Distributed Systems*, 11(7):669–682, July 2000. (Cited on page 44.)

[97] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991. (Cited on page 214.)

[98] Griffith Hamlin, Jr and James D. Foley. Configurable applications for graphics employing satellites (CAGES). *ACM SIGGRAPH Computer Graphics*, 9(1):9–19, Spring 1975. (Cited on page 221.)

[99] Kevin Hammond. Parallel functional programming: An introduction. In *Proceedings of the International Symposium on Parallel Symbolic Computation (PASCO 94)*, pages 181–193, Hagenberg/Linz, Austria, September 1994. World Scientific. (Cited on page 207.)

[100] Chih-Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler, and Mani Srivastava. A dynamic operating system for sensor nodes. In *Proceedings of the 3rd International Conference on Mobile Systems, Applicaions and Services (MobiSys '05)*, pages 163–176, Seattle, WA, USA, 2005. ACM. (Cited on page 54.)

[101] Robert Harle and Alastair Beresford. Keeping big brother off the road. *IEE Review*, 51(10):34–37, October 2005. (Cited on page 17.)

[102] Robert K. Harle. *Maintaining World Models In Context-Aware Environments*. PhD thesis, Laboratory for Communication Engineering, Department of Engineering, University of Cambridge, 2004. (Cited on pages 74 and 87.)

[103] Hannes Hartenstein, Bernd Bochow, André Ebner, Matthias Lott, Markus Radimirsch, and Dieter Vollmer. Position-aware ad hoc wireless networks for inter-vehicle communications: the Fleetnet project. In *MobiHoc '01: Proceedings of the 2nd ACM international symposium on Mobile ad hoc networking & computing*, pages 259–262, Long Beach, CA, USA, 2001. ACM Press. (Cited on page 25.)

[104] John Heidemann, Fabio Silva, Chalermek Intanagonwiwat, Ramesh Govindan, Deborah Estrin, and Deepak Ganesan. Building efficient wireless sensor networks with low-level naming. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 146–159, Banff, Alberta, Canada, 2001. (Cited on page 49.)

[105] Wendi B. Heinzelman, Amy L. Murphy, Hervaldo S. Carvalho, and Mark A. Perillo. Middleware to support sensor network applications. *IEEE Network*, 18(1):6–14, Jan/Feb 2004. (Cited on page 50.)

[106] Wendi Rabiner Heinzelman, Anantha Chandrakasan, and Hari Balakrishnan. Energy-efficient communication protocol for wireless microsensor networks. In *Proceedings of the 33rd Hawaii International Conference on System Sciences (HICSS)*, January 2000. (Cited on page 202.)

[107] Wendi Rabiner Heinzelman, Joanna Kulik, and Hari Balakrishnan. Adaptive protocols for information dissemination in wireless sensor networks. In *Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking (Mobicom '99)*, pages 174–185, Seattle, WA, USA, 1999. (Cited on page 201.)

[108] Eric H. Herrin II and Raphael A. Finkel. An implementation of service rebalancing. Technical Report 191-91, University of Kentucky, July 1991. (Cited on page 45.)

[109] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, pages 93–104, Cambridge, MA, USA, November 2000. (Cited on pages 46, 47, 195 and 198.)

[110] Jonathan M. D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, and Rob H. Bisseling. BSPlib: The BSP programming library. *Parallel Computing*, 24:1947–1980, 1998. (Cited on page 37.)

[111] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985. (Cited on page 211.)

[112] Steve Hodges, Lyndsay Williams, Emma Berry, Shahram Izadi, James Srinivasan, Alex Butler, Gavin Smyth, Narinder Kapur, and Ken Wood. SenseCam: A retrospective memory aid. In *Ubicomp 2006*, volume 4206 of *LNCS*, pages 177–193, Orange County, CA, USA, September 2006. Springer-Verlag. (Cited on page 64.)

[113] Karin Hogstedt, Doug Kimelman, V. T. Rajan, Tova Roth, and Mark Wegman. Graph cutting algorithms for distributed applications partitioning. *ACM SIGMETRICS Performance Evaluation Review*, 28(4):27–29, March 2001. (Cited on page 223.)

[114] Andy Hopper. Sentient computing (abridged and updated version of the Royal Society Clifford Paterson Lecture, 1999). In *Computer Systems: Theory, Technology, and Applications: A Tribute to Roger Needham*, Monographs in Computer Science, pages 125–131. Springer-Verlag, December 2003. (Cited on page 21.)

[115] Andy Hopper and Andrew Rice. Computing for the future of the planet. *Philosophical Transactions of the Royal Society A*, 366(1881):3685–3697, October 2008. (Cited on page 188.)

[116] John M. Howie. *Fundamentals of Semigroup Theory*. Oxford University Press Inc., New York, 1995. (Cited on page 108.)

[117] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the programming language Haskell: A non-strict, purely functional language. *ACM SIGPLAN Notices*, 27:1–164, May 1992. (Cited on page 206.)

[118] Barbara Hughes, René Meier, Raymond Cunningham, and Vinny Cahill. Towards real-time middleware for vehicular ad hoc networks. In *Proceedings of the 1st ACM Workshop on Vehicular Ad Hoc Networks*, pages 95–96, Philadephia, US, October 2004. (Cited on page 27.)

[119] Bret Hull, Vladimir Bychkovsky, Yang Zhang, Kevin Chen, Michel Goraczko, Allen Miu, Eugene Shih, Hari Balakrishnan, and Samuel Madden. CarTel: A distributed mobile sensor computing system. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems (SenSys)*, pages 125–138, Boulder, CO, USA, November 2006. (Cited on page 28.)

[120] Galen C. Hunt and Michael L. Scott. The Coign automatic distributed partitioning system. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*, pages 187–200, New Orleans, LA, USA, February 1999. (Cited on page 222.)

[121] Tomasz Imielinski and Samir Goel. DataSpace: Querying and monitoring deeply networked collections in physical space. *IEEE Personal Communications*, 7(5):4–9, October 2000. (Cited on page 50.)

[122] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *MobiCom 2000: Proceedings of the 6th Annual International Conference on Mobile Computing and Networking*, pages 56–67, Boston, MA, USA, August 2000. ACM Press. (Cited on page 201.)

[123] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2007 EuroSys Conference*, pages 59–72, Lisbon, Portugal, March 2007. ACM. (Cited on pages 42, 43, 143 and 161.)

[124] Chaiporn Jaikaeo, Chavalit Srisathapornphat, and Chien-Chung Shen. Querying and tasking in sensor networks. In *SPIE's 14th Annual International Symposium on Aerospace/Defense Sensing, Simulation, and Control*, Orlando, FL, USA, April 2000. (Cited on pages 51 and 53.)

[125] David B. Johnson and David A. Maltz. *Dynamic Source Routing in Ad Hoc Wireless Networks*, volume 353, pages 153–181. Springer US, 1996. (Cited on page 25.)

[126] Kirk L. Johnson, M. Frans Kaashoek, and Deborah A. Wallach. CRL: High-performance all-software distributed shared memory. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SIGOPS '95)*, pages 213–226, Copper Mountain, CO, USA, 1995. ACM. (Cited on page 218.)

[127] Anthony D. Joseph, Joshua A. Tauber, and M. Frans Kaashoek. Mobile computing with the Rover toolkit. *IEEE Transactions on Computers*, 46(3):337–352, March 1997. (Cited on page 40.)

[128] Philo Juang, Hidekazu Oki, Yong Wang, Margaret Martonosi, Li-Shiuan Peh, and Daniel Rubenstein. Energy-efficient computing for wildlife tracking: Design trade-offs and early experiences with ZebraNet. *SIGPLAN Notices*, 37(10):96–107, October 2002. (Cited on pages 54 and 197.)

[129] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988. (Cited on page 40.)

[130] Silke Jung. HGV tolls in Germany: Innovative, environmentally friendly and fair. *The IEE Road Transport Symposium*, pages 5/1–7, December 2005. (Cited on page 22.)

[131] Muhammad Kafil and Ishfaq Ahmad. Optimal task assignment in heterogeneous distributed computing systems. *IEEE Concurrency*, 8(3):42–51, July–September 1998. (Cited on page 158.)

[132] Porlin Kang, Cristian Borcea, Gang Xu, Akhilesh Saxena, Ulrich Kremer, and Liviu Iftode. Smart messages: A distributed computing platform for networks of embedded systems. *The Computer Journal*, 47:475–494, January 2004. (Cited on page 53.)

[133] Eiman Kanjo and Peter Landshoff. Mobile phones to monitor pollution. *IEEE Distributed Systems Online*, 8(7), 2007. (Cited on page 197.)

[134] Hillol Kargupta, Ruchita Bhargava, Kun Liu, Michael Powers, Patrick Blair, Samuel Bushra, James Dull, Kakali Sarkar, Martin Klein, Mitesh Vasa, and David Handy. VEDAS: A mobile and distributed data stream mining system for real-time vehicle monitoring. In Michael W. Berry, Umeshwar Dayal, Chandrika Kamath, and David B. Skillicorn, editors, *Proceedings of the SIAM International Data Mining Conference*, number 117 in Proceedings in Applied Mathematics, pages 300–311, Orlando, FL, USA, 2004. Cambridge University Press. (Cited on pages 29 and 91.)

[135] Brad Karp and H. T. Kung. GPSR: Greedy perimeter stateless routing for wireless networks. In *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking (MOBICOM '00)*, pages 243–254, Boston, MA, USA, 2000. (Cited on page 25.)

[136] Richard M. Karp and Vijaya Ramachandran. A survey of parallel algorithms for shared-memory machines. Technical Report CSD-88-408, UC Berkeley, March 1988. (Cited on page 207.)

[137] J. R. Kennaway and M. R. Sleep. The 'language first' approach. In Chambers et al. [44], chapter 7, pages 111–124. (Cited on page 30.)

[138] Hong Bong Kim, Marc Emmelmann, Berthold Rathke, and Adam Wolisz. A radio over fiber network architecture for road vehicle communication systems. In *Proceedings of the 61st IEEE Vehicular Technology Conference (VTC Spring 2005)*, volume 5, pages 2920–2924, May 2005. (Cited on page 193.)

[139] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary E. Zosel. *The High Performance Fortran Handbook*. Scientific and Engineering Computation Series. MIT Press, Cambridge, MA, USA, 1994. (Cited on page 218.)

[140] Donald Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys (CSUR)*, 32(4):422–469, December 2000. (Cited on pages 48 and 143.)

[141] Ulrich Kremer, Jamey Hicks, and James H. Rehg. A compilation framework for power and energy management on mobile computers. Technical Report DCS-TR-446, Rutgers University, June 2001. (Cited on page 43.)

[142] Rajnish Kumar, Matthew Wolenetz, Bikash Agarwalla, JunSuk Shin, Phillip Hutto, Arnab Paul, and Umakishore Ramachandran. DFuse: A framework for distributed data fusion. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems (SenSys '03)*, pages 114–125, Los Angeles, CA, USA, November 2003. ACM Press, New York, NY, USA. (Cited on page 46.)

[143] Ram Kumar, Vlasios Tsiatsis, and Mani B. Srivastava. Computation hierarchy for in-network processing. In *WSNA '03: Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, pages 68–77, San Diego, CA, USA, September 2003. ACM Press. (Cited on page 222.)

[144] Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys (CSUR)*, 31(4):406–471, December 1999. (Cited on pages 99 and 158.)

[145] Uichin Lee, Eugenio Magistretti, Biao Zhou, Mario Gerla, Paolo Bellavista, and Antonio Corradi. MobEyes: Smart mobs for urban monitoring with vehicular sensor networks. *IEEE Wireless Communications*, 13(5):52–57, October 2006. (Cited on pages 27 and 28.)

[146] Gabriel Leen and Donal Heffernan. Expanding automotive electronic systems. *IEEE Computer*, 35(1):88–93, January 2002. (Cited on page 23.)

[147] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The Stanford Dash multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992. (Cited on pages 33 and 217.)

[148] Philip Levis. The TinyScript language. Available from `http://www.cs.berkeley.edu/ pal/mate-web/files/tinyscript-manual.pdf`, July 2004. (Cited on page 52.)

[149] Philip Levis and David Culler. Maté: A tiny virtual machine for sensor networks. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2002)*, pages 85–95, San Jose, CA, USA, October 2002. (Cited on page 52.)

[150] Philip Levis, David Gay, and David Culler. Active sensor networks. In *Proceedings of the 2nd Symposium on Networked Systems Design & Implementation (NSDI '05)*, volume 2, pages 343–356, Berkeley, CA, USA, 2005. USENIX Association. (Cited on pages 51 and 52.)

[151] Jinyang Li, John Jannotti, Douglas S. J. De Couto, David R. Karger, and Robert Morris. A scalable location service for geographic ad hoc routing. In *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking (MOBICOM 2000)*, pages 120–130, Boston, MA, USA, August 2000. (Cited on page 26.)

[152] Shuoqi Li, Ying Lin, Sang H. Son, John A. Stankovic, and Yuan Wei. Event detection services using data service middleware in distributed sensor networks. *Telecommunication Systems*, 26(2–4):351–368, June 2004. (Cited on page 50.)

[153] Wei-yi Li. Design and implementation of digital radio communications link for platoon control. Technical Report UCB-ITS-PRR-95-2, California Partners for Advanced Transit and Highways (PATH), Institute of Transportation Studies, University of California, Berkeley, January 1995. (Cited on page 26.)

[154] Zhiyuan Li, Cheng Wang, and Rong Xu. Computation offloading to save energy on handheld devices: A partition scheme. In *CASES '01: Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 238–246, Atlanta, GA, USA, November 2001. ACM Press. (Cited on page 223.)

[155] Stephanie Lindsey and Cauligi S. Raghavendra. PEGASIS: Power-efficient gathering in sensor information systems. In *Proceedings of the 2002 IEEE Aerospace Conference*, volume 3, pages 1125–1130, March 2002. (Cited on page 202.)

[156] Nikitas Liogkas, Blair MacIntyre, Elizabeth D. Mynatt, Yannis Smaragdakis, Eli Tilevich, and Stephen Voida. Automatic partitioning for prototyping ubiquitous computing applications. *IEEE Pervasive Computing*, 3(3):40–47, July–September 2004. (Cited on pages 34 and 223.)

[157] Todd Litman. Distance-based vehicle insurance as a TDM strategy. Available online at `http://www.vtpi.org/dbvi.pdf`, December 2004. (Cited on page 22.)

[158] Genping Liu, Bu-Sung Lee, Boon-Chong Seet, Chuan-Heng Foh, Kai-Juan Wong, and Keok-Kee Lee. A routing strategy for metropolis vehicular communications. In *Proceedings of the 18th International Conference on Information Networking*, pages 134–143. Springer, February 2004. (Cited on pages 24 and 26.)

[159] Ting Liu and Margaret Martonosi. Impala: A middleware system for managing autonomic, parallel sensor systems. In *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 107–118, San Diego, CA, USA, June 2003. (Cited on page 54.)

[160] Clemens Lombriser, Daniel Roggen, Mathias Stäger, and Gerhard Tröster. Titan: A tiny task network for dynamically reconfigurable heterogeneous sensor networks. In *Kommunikation in Verteilten Systemen (KiVS)*, pages 127–138, Berlin / Heidelberg, February 2007. Springer. (Cited on page 44.)

[161] William E. Lorensen and Harvey E. Cline. Marching cubes: a high resolution 3D surface construction algorithm. *SIGGRAPH Computer Graphics*, 21(4):163–169, July 1987. (Cited on page 85.)

[162] Honghui Lu, Sandhya Dwarkadas, Alan L. Cox, and Willy Zwaenepoel. Message passing versus distributed shared memory on networks of workstations. In *Proceedings of the ACM/IEEE SC 1995 Conference: Conference on High Performance Networking and Computing (SC '95)*, San Diego, CA, USA, December 1995. (Cited on page 33.)

[163] Jason Maassen, Rob van Nieuwpoort, Ronald Veldema, Henri E. Bal, and Aske Plaat. An efficient implementation of Java's remote method invocation. In *Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 173–182, Atlanta, GA, USA, 1999. ACM. (Cited on page 35.)

[164] David B. MacQueen. Models for distributed computing. Technical Report 351, Institut de Recherche d'Informatique et d'Automatique (IRIA), April 1979. (Cited on pages 209, 213 and 214.)

[165] Samuel Madden, Michael J. Franklin, Joseph Hellerstein, and Wei Hong. TAG: a Tiny AGgregation service for ad-hoc sensor networks. *ACM SIGOPS Operating Systems Review*, 36:131–146, 2002. (Cited on pages 48, 49, 107 and 225.)

[166] Samuel Madden, Robert Szewczyk, Michael J. Franklin, and David Culler. Supporting aggregate queries over ad-hoc wireless sensor networks. In *Proceedings of the 4th IEEE Workshop on Mobile Computing Systems and Applications*, pages 49–58, 2002. (Cited on page 49.)

[167] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TinyDB: An acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems*, 30(1):122–173, March 2005. (Cited on page 49.)

[168] Geoff Mainland, Laura Kang, Sebastien Lahaie, David C. Parkes, and Matt Welsh. Using virtual markets to program global behavior in sensor networks. In *Proceedings of the 11th ACM SIGOPS European Workshop*, pages 1–6, Leuven, Belgium, 2004. ACM. (Cited on page 55.)

[169] Alan Mainwaring, Joseph Polastre, Robert Szewczyk, David Culler, and John Anderson. Wireless sensor networks for habitat monitoring. In *Proceedings of the 1st ACM International Workshop on Wireless Sensor Networks and Applications (WSNA '02)*, pages 88–97, Atlanta, GA, USA, September 2002. (Cited on page 197.)

[170] Arati Manjeshwar and Dharma P. Agrawal. TEEN: A routing protocol for enhanced efficiency in wireless sensor networks. In *Proceedings of the 15th International Parallel & Distributed Processing Symposium*, San Francisco, CA, USA, April 2001. (Cited on page 202.)

[171] Kieran Mansley, David Scott, Alastair Tse, and Anil Madhavapeddy. Feedback, latency, accuracy: Exploring tradeoffs in location-aware gaming. In *SIGCOMM 2004 Workshops: Proceedings of ACM SIGCOMM 2004 workshops on NetGames '04*, pages 93–97, Portland, Oregon, USA, August 2004. ACM Press. (Cited on page 17.)

[172] Keith D. McDonald and Christopher Hegarty. Post-modernization GPS performance capabilities. In *Proceedings of the IAIN World Congress and the ION 56th Annual Meeting*, pages 242–249, San Diego, CA, USA, June 2000. Institute of Navigation. (Cited on page 82.)

[173] Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: Reconciling objects, relations and XML in the .NET framework. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, page 706, Chicago, IL, USA, 2006. (Cited on page 150.)

[174] Daniel A. Menascé, Stella C. da Silva Porto, and Satish K. Tripathi. Static heuristic processor assignment in heterogeneous multiprocessors. *International Journal of High Speed Computing*, 6(1):115–137, March 1994. (Cited on page 159.)

[175] Janet Michel and Andries van Dam. Experience with distributed processing on a host/satellite graphics system. *ACM SIGGRAPH Computer Graphics*, 10(2):190–195, Summer 1976. (Cited on page 222.)

[176] A. J. R. G. Milner. Using algebra for concurrency. In Chambers et al. [44], chapter 21, pages 291–305. (Cited on page 210.)

[177] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer-Verlag, 1980. (Cited on pages 210 and 211.)

[178] Robin Milner. A proposal for Standard ML. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, pages 184–197, Austin, TX, USA, 1984. ACM. (Cited on page 206.)

[179] Robin Milner. *Communicating and Mobile Systems: The π-Calculus*. Cambridge University Press, 1999. (Cited on pages 211 and 212.)

[180] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, April 1965. (Cited on pages 22 and 195.)

[181] Hans P. Moravec and Alberto Elfes. High resolution maps from wide angle sonar. In *Proceedings of the 1985 IEEE International Conference on Robotics and Automation*, pages 116–121, March 1985. (Cited on page 76.)

[182] Robert Morris, John Jannotti, Frans Kaashoek, Jinyang Li, and Douglas Decouto. CarNet: A scalable ad hoc wireless network system. In *Proceedings of the 9th ACM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating System*, pages 61–65, Kolding, Denmark, September 2000. (Cited on page 26.)

[183] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989. (Cited on page 212.)

[184] Patrick Murphy, Erik Welsh, and J. Patrick Frantz. Using Bluetooth for short-term ad hoc connections between moving vehicles: A feasibility study. In *Proceedings of the 55th IEEE Vehicular Technology Conference (VTC Spring 2002)*, volume 1, pages 414–418, 2002. (Cited on page 193.)

[185] Peter Muszynski and Harri Holma. *Introduction to WCDMA*, chapter 3. John Wiley & Sons, Ltd, Chichester, England, 2nd edition, April 2004. (Cited on page 28.)

[186] Tamer Nadeem, Sasan Dashtinezhad, Chunyuan Liao, and Liviu Iftode. TrafficView: Traffic data dissemination using car-to-car communication. *Mobile Computing and Communications Review*, 8(3):6–19, July 2004. (Cited on pages 24 and 29.)

[187] Suman Nath, Yan Ke, Phillip B. Gibbons, Brad Karp, and Srinivasan Seshan. IrisNet: An architecture for enabling sensor-enriched internet services. Technical Report IRP-TR-02-10, Intel Research Pittsburgh, December 2002. (Cited on page 50.)

[188] Maziar Nekovee. Sensor networks on the road: The promises and challenges of vehicular ad hoc networks and grids. In *Workshop on Ubiquitous Computing and e-Research*, National e-Science Centre, Edinburgh, Scotland, May 2005. (Cited on pages 24 and 25.)

[189] Yang Ni, Ulrich Kremer, Adrian Stere, and Liviu Iftode. Programming ad-hoc networks of mobile and resource-constrained devices. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 249–260, Chicago, IL, USA, June 2005. (Cited on pages 53, 101, 146, 147 and 149.)

[190] David A. Nichols. Using idle workstations in a shared computing environment. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP '87)*, pages 5–12, Austin, TX, USA, November 1987. (Cited on page 37.)

[191] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In *Proc. 12th International Conference on Compiler Construction*, volume 2622 of *LNCS*, pages 138–152, Warsaw, Poland, April 2003. Springer-Verlag. (Cited on page 160.)

[192] Salim Omar, Xinan Zhou, and Thomas Kunz. Mobile code, adaptive mobile applications, and network architectures. In *Mobile Agents for Telecommunication Applications (MATA 2000)*, volume 1931 of *LNCS*, pages 319–330. Springer-Verlag, 2000. (Cited on page 45.)

[193] Åke Östmark, Per Lindgren, Aart van Halteren, and Lianne Meppelink. Service and device discovery of nodes in a wireless sensor network. In *IEEE Consumer Communications and Networking Conference (CCNC 2006)*, pages 218–222, Las Vegas, NV, USA, January 2006. (Cited on page 199.)

[194] Jörg Ott and Dirk Kutscher. Drive-Thru Internet: IEEE 802.11b for "automobile" users. In *Proceedings of the 23rd Annual Joint Conference of the IEEE Computer and Communication Societies (INFOCOM 2004)*, volume 1, March 2004. (Cited on page 193.)

[195] Shumao Ou, Kun Yang, and Antonio Liotta. An adaptive multi-constraint partitioning algorithm for offloading in pervasive systems. In *Fourth IEEE International Conference on Pervasive Computing and Communications (PERCOM 2006)*, pages 116–125, Pisa, Italy, March 2006. (Cited on page 223.)

[196] Peter S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann, 1996. (Cited on page 37.)

[197] Markos Papageorgiou and Apostolos Kotsialos. Freeway ramp metering: An overview. *IEEE Transactions on Intelligent Transportation Systems*, 3(4):271–281, December 2002. (Cited on page 17.)

[198] Charles E. Perkins and Elizabeth M. Royer. Ad-hoc on-demand distance vector routing. In *Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications (WMCSA '99)*, pages 90–100, New Orleans, LA, USA, February 1999. (Cited on page 25.)

[199] Emil M. Petriu, Nicolas D. Georganas, Dorina C. Petriu, Dimitrios Makrakis, and Voicu Z. Groza. Sensor-based information appliances. *IEEE Instrumentation & Measurement Magazine*, 3(4):31–35, December 2000. (Cited on page 197.)

[200] G. J. Pottie and W. J. Kaiser. Wireless integrated network sensors. *Communications of the ACM*, 43(5):51–58, May 2000. (Cited on page 48.)

[201] Michael L. Powell and Barton P. Miller. Process migration in DEMOS/MP. *ACM SIGOPS Operating Systems Review*, 17(5):110–119, October 1983. (Cited on page 40.)

[202] Ramjee Prasad and Marina Ruggieri. *Applied Satellite Navigation using GPS, GALILEO, and Augmentation Systems*. Artech House, London, UK, 2005. (Cited on page 82.)

[203] Hairong Qi, Xiaoling Wang, S. Sitharama Iyengar, and Krishnendu Chakrabarty. Multisensor data fusion in distributed sensor networks using mobile agents. In *Proceedings of the 4th International Conference on Information Fusion*, pages TuC2:11–16, Montreal, Canada, August 2001. (Cited on page 51.)

[204] Connie Ribeiro. Bringing wireless access to the automobile: A comparison of Wi-Fi, WiMAX, MBWA and 3G. In *Proceedings of the 21st Annual Rensselaer at Hartford Computer Science Conference*, April 2005. (Cited on page 194.)

[205] Andrew C. Rice, Alastair R. Beresford, and Robert K. Harle. Cantag: An open source software toolkit for designing and deploying marker-based vision systems. In *Proceedings of the 4th Annual IEEE International Conference on Pervasive Computing and Communications (PERCOM)*, pages 12–21, March 2006. (Cited on page 175.)

[206] Golden G. Richard, III. *Service and Device Discovery Protocols and Programming*, chapter 4. McGraw-Hill, 2002. (Cited on page 199.)

[207] Martin C. Rinard, Daniel J. Scales, and Monica S. Lam. Jade: A high-level, machine-independent language for parallel programming. *IEEE Computer*, 26(6):28–38, June 1993. (Cited on page 218.)

[208] Volkan Rodoplu and Teresa H. Meng. Minimum energy mobile wireless networks. *IEEE Journal on Selected Areas in Communications*, 17(8):1333–1344, August 1999. (Cited on page 202.)

[209] Alexey Rudenko, Peter Reiher, Gerald J. Popek, and Geoffrey H. Kuenning. Saving portable computer battery power through remote process execution. *ACM SIGMOBILE Mobile Computing and Communications Review*, 2(1):19–26, January 1998. (Cited on page 223.)

[210] Alexey Rudenko, Peter Reiher, Gerald J. Popek, and Geoffrey H. Kuenning. The remote processing framework for portable computer power saving. In *Proceedings of the 1999 ACM Symposium on Applied Computing*, pages 365–372, San Antonio, TX, USA, 1999. ACM. (Cited on page 45.)

[211] Vivek Sarkar and John Hennessy. Compile-time partitioning and scheduling of parallel programs. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, pages 17–26, Palo Alto, CA, USA, 1986. ACM. (Cited on page 223.)

[212] M. Satyanarayanan and Dushyanth Narayanan. Multi-fidelity algorithms for interactive mobile applications. *Wireless Networks*, 7:601–607, November 2001. (Cited on page 45.)

[213] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '96)*, pages 174–185, Cambrigde, MA, USA, 1996. ACM. (Cited on page 218.)

[214] Curt Schurgers and Mani B. Srivastava. Energy efficient routing in wireless sensor networks. In *Proceedings of the Military Communications Conference (MILCOM 2001)*, volume 1, pages 357–361, 2001. (Cited on page 201.)

[215] Rahul C. Shah and Jan M. Rabaey. Energy aware routing for low energy ad hoc sensor networks. In *Proceedings of the Wireless Communications and Networking Conference (WCNC 2002)*, volume 1, pages 350–355, March 2002. (Cited on page 201.)

[216] Mary Sheeran. muFP, a language for VLSI design. In *Proc. 1984 ACM Symposium on LISP and Functional Programming*, pages 104–112, Austin, TX, USA, August 1984. ACM. (Cited on page 147.)

[217] Doug Simon, Cristina Cifuentes, Dave Cleal, John Daniels, and Derek White. Java$^{\mathrm{TM}}$ on the bare metal of wireless sensor devices: The Squawk Java virtual machine. In *Proceedings of the 2nd ACM/USENIX International Conference on Virtual Execution Environments (VEE '06)*, pages 78–88, Ottawa, Ontario, Canada, 2006. (Cited on page 195.)

[218] Thirunavukkarasu Sivaharan, Gordon Blair, Adrian Friday, Maomao Wu, Hector Duran-Limon, Paul Okanda, and Carl-Fredrik Sørensen. Cooperating sentient vehicles for next generation automobiles. In *Proceedings of the ACM/USENIX MobiSys 2004 International Workshop on Applications of Mobile Embedded Systems (WAMES 2004)*, Boston, MA, USA, June 2004. (Cited on page 29.)

[219] D. B. Skillicorn. *Parallelism and the Bird-Meertens Formalism*. Department of Computing and Information Science, Queen's University, Kingston, Ontario, April 1992. (Cited on pages 215 and 216.)

[220] D. B. Skillicorn, Jonathan M. D. Hill, and W. F. McColl. Questions and answers about BSP. *Scientific Programming*, 6(3):249–274, 1997. (Cited on page 32.)

[221] David Skillicorn. *Foundations of Parallel Programming*, volume 6 of *Cambridge International Series on Parallel Computation*. Cambridge University Press, 1994. (Cited on page 215.)

[222] Siarhei Smolau and Ronald Beaubrun. State-oriented programming for TinyOS. In *Proceedings of the 2007 Summer Computer Simulation Conference*, pages 766–771, San Diego, CA, USA, 2007. Society for Computer Simulation International. (Cited on page 47.)

[223] Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-typed actors for Java. In *ECOOP 2008: 22nd European Conference on Object-Oriented Programming*, volume 5142 of *LNCS*, pages 104–128, Paphos, Cyprus, July 2008. Springer. (Cited on page 206.)

[224] Chavalit Srisathapornphat, Chaiporn Jaikaeo, and Chien-Chung Shen. Sensor information networking architecture. In *Proceedings of the 2000 International Workshop on Parallel Processing (ICPP '00)*, pages 23–30, Toronto, Ontario, Canada, 2000. (Cited on page 49.)

[225] Phillip Stanley-Marbell and Liviu Iftode. Scylla: A smart virtual machine for mobile embedded systems. In *Proceedings of the 3rd IEEE Workshop on Mobile Computing Systems and Applications (WMCSA '00)*, pages 41–50, Monterey, CA, USA, December 2000. (Cited on page 52.)

[226] Robert Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, July 1997. (Cited on pages 97 and 213.)

[227] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990. (Cited on pages 36 and 218.)

[228] James E. Tate. A novel research tool – presenting the highly instrumented car. *Traffic Engineering and Control*, 46(7):262–265, July 2005. (Cited on page 58.)

[229] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: The Condor experience. *Concurrency and Computation: Practice and Experience*, 17:323–356, Feb–Apr 2005. (Cited on page 37.)

[230] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction (CC 2002)*, volume 2304 of *LNCS*, pages 179–196, Grenoble, France, April 2002. Springer. (Cited on page 215.)

[231] Sebastian Thrun. Robotic mapping: A survey. In Gerhard Lakemeyer and Bernhard Nebel, editors, *Exploring Artificial Intelligence in the New Millennium*, chapter 1, pages 1–36. Morgan Kaufmann, July 2002. (Cited on page 76.)

[232] Gilman Tolle, Joseph Polastre, Robert Szewczyk, David Culler, Neil Turner, Kevin Tu, Stephen Burgess, Todd Dawson, Phil Buonadonna, David Gay, and Wei Hong. A macroscope in the redwoods. In *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems (SenSys '05)*, pages 51–63, San Diego, CA, USA, November 2005. (Cited on page 197.)

[233] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990. (Cited on page 32.)

[234] R. J. van Glabbeek. Notes on the methodology of CCS and CSP. *Theoretical Computer Science*, 177:329–349, 1997. (Cited on page 211.)

[235] Upkar Varshney. Vehicular mobile commerce. *IEEE Computer*, 37(12):116–118, December 2004. (Cited on page 16.)

[236] R. Veldema, R. F. H. Hofman, R. A. F. Bhoedjang, C. J. H. Jacobs, and H. E. Bal. Source-level global optimizations for fine-grain distributed shared memory systems. *ACM SIGPLAN Notices*, 36(7):83–92, July 2001. (Cited on page 218.)

[237] Alexander Verbraeck and Corné Versteegt. A bridge between the design and implementation of complex transportation systems. In Dietmar P. F. Möller, editor, *Proceedings of the 12th European Simulation Symposium – Simulation in Industry (ESS2000)*, pages 238–243, Hamburg, Germany, September 2000. SCS Publications, Ghent. (Cited on page 29.)

[238] John von Neumann. First draft of a report on the EDVAC. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993. (Cited on page 30.)

[239] William W. Wadge. An extensional treatment of dataflow deadlock. In Gilles Kahn, editor, *Semantics of Concurrent Computation*, volume 70 of *LNCS*, pages 285–299, Evian, France, 1979. Springer-Verlag. (Cited on pages 141 and 142.)

[240] William W. Wadge and Edward A. Ashcroft. *Lucid, the Dataflow Programming Language*, volume 22 of *APIC Studies in Data Processing*. Academic Press, 1985. (Cited on page 214.)

[241] Jim Waldo. The Jini architecture for network-centric computing. *Communications of the ACM*, 42(7):76–82, July 1999. (Cited on page 199.)

[242] David W. Walker. The design of a standard message passing interface for distributed memory concurrent computers. Technical Report ORNL/TM-12512, Oak Ridge National Laboratory, TN, USA, October 1993. (Cited on pages 36, 37 and 105.)

[243] Mark Weiser. The computer for the twenty-first century. *Scientific American*, 265(3):94–104, September 1991. (Cited on pages 21 and 100.)

[244] Matt Welsh and Geoff Mainland. Programming sensor networks using abstract regions. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation (NSDI 2004)*, pages 29–42, San Francisco, CA, USA, March 2004. (Cited on pages 54 and 105.)

[245] Kamin Whitehouse, Cory Sharp, Eric Brewer, and David Culler. Hood: A neighborhood abstraction for sensor networks. In *Proceedings of the 2nd International Conference on Mobile Systems, Applications, and Services (MobiSys 2004)*, pages 99–110, Boston, MA, USA, June 2004. (Cited on page 54.)

[246] Lars Wischhof, André Ebner, and Hermann Rohling. Information dissemination in self-organizing intervehicle networks. *IEEE Transactions on Intelligent Transportation Systems*, 6(1):90–101, March 2005. (Cited on page 27.)

[247] Lars Wischhof, André Ebner, Hermann Rohling, Matthias Lott, and Rüdiger Halfmann. SOTIS – a self-organizing traffic information system. In *The 57th IEEE Semi-Annual Vehicular Technology Conference (VTC 2003-Spring)*, volume 4, pages 2442–2446, April 2003. (Cited on pages 27 and 29.)

[248] Hao Wu, Richard Fujimoto, Randall Guensler, and Michael Hunter. MDDV: A mobility-centric data dissemination algorithm for vehicular networks. In *Proceedings of the 1st ACM International Workshop on Vehicular Ad Hoc Networks (VANET '04)*, pages 47–56, Philadelphia, PA, USA, October 2004. (Cited on pages 25 and 26.)

[249] Hao Wu, Jaesup Lee, Michael Hunter, Richard Fujimoto, Randall L. Guensler, and Joonho Ko. Efficiency of simulated vehicle-to-vehicle message propagation efficiency on Atlanta's I-75 corridor. *Transportation Research Record: Journal of the Transportation Research Board*, 1910:82–89, 2005. (Cited on page 26.)

[250] Bo Xu, Aris Ouksel, and Ouri Wolfson. Opportunistic resource exchange in intervehicle ad-hoc networks. In *Proceedings of the IEEE International Conference on Mobile Data Management (MDM 2004)*, pages 4–12, 2004. (Cited on page 28.)

[251] Ning Xu, Sumit Rangwala, Krishna Kant Chintalapudi, Deepak Ganesan, Alan Broad, Ramesh Govindan, and Deborah Estrin. A wireless sensor network for

structural monitoring. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems (SenSys '04)*, pages 13–24, Baltimore, MD, USA, November 2004. (Cited on page 198.)

[252] Ya Xu, John Heidemann, and Deborah Estrin. Geography-informed energy conservation for ad hoc routing. In *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking*, pages 70–84, Rome, Italy, 2001. (Cited on page 202.)

[253] Yong Yao and Johannes Gehrke. Query processing for sensor networks. In *Proceedings of the 1st Biennial Conference on Innovative Data Systems Research (CIDR 2003)*, Asilomar, CA, USA, January 2003. (Cited on page 48.)

[254] Jijun Yin, Tamer ElBatt, Gavin Yeung, Bo Ryu, Stephen Habermas, Hariharan Krishnan, and Timothy Talty. Performance evaluation of safety applications over dsrc vehicular ad hoc networks. In *VANET '04: Proceedings of the first ACM workshop on Vehicular ad hoc networks*, pages 1–9, Philadelphia, PA, USA, 2004. ACM Press. (Cited on page 193.)

[255] Shigeki Yokoi, Jun-Ichiro Toriwaki, and Teruo Fukumura. An analysis of topological properties of digitized binary pictures using local features. *Computer Graphics and Image Processing*, 4(1):63–73, March 1975. (Cited on page 78.)

[256] Weimin Yu and Alan Cox. Java/DSM: A platform for heterogeneous computing. *Concurrency: Practice and Experience*, 9(11):1213–1224, November 1997. (Cited on page 217.)

[257] Yan Yu, Ramesh Govindan, and Deborah Estrin. Geographical and energy aware routing: A recursive data dissemination protocol for wireless sensor networks. Technical Report CSD-TR-01-0023, UCLA Computer Science Department, May 2001. (Cited on page 203.)

[258] Yang Zhang, Bret Hull, Hari Balakrishnan, and Samuel Madden. ICEDB: Intermittently-connected continuous query processing. In *Proceedings of the 23rd International Conference on Data Engineering (ICDE 2007)*, pages 166–175, Istanbul, Turkey, April 2007. IEEE. (Cited on page 49.)

[259] Jing Zhao and Guohong Cao. VADD: Vehicle-assisted data delivery in vehicular ad hoc networks. *IEEE Transactions on Vehicular Technology*, 57(3):1910–1922, May 2008. (Cited on page 26.)