3 • Parallel Algorithms

Chip Multiprocessors (ACS MPhil)
Robert Mullins
Books

- Introduction to Parallel Computing
  Grama/Gupta/Karypis

- Patterns for Parallel Programming
  Mattson/Sanders/Massingill

- Parallel Computer Architecture
  Culler/Singh
Introduction

• How might we exploit our chip-multiprocessor?
  – Use it to improve the performance of a single program
  – Allow us to solve larger and larger problems (while keeping running time constant)
  – Introduce completely new applications
    • Those that were not feasible in the past
  – Run multiple programs or processes in parallel
    • Workstation applications
    • Server applications
      – Throughput computing
• The focus today is on developing explicitly parallel algorithms and programs
Introduction

• Goals:
  – Correctness
    • May require equivalence to sequential version
  – Simplicity, low-development time, maintainability
    • Algorithm is apparent from source code
    • Easy to debug, verify and modify
  – Performance, scalability and portability
  – Low power consumption
    • Energy and power consumption will increasingly limit performance
Top-down influences

• How do we develop a parallel program?
  – 1. Identify concurrency in the problem
    • Decomore our problem into subproblems (tasks) that can safely execute at the same time
      – Task dependency graph
      – Critical path
      – Degree of concurrency
    • There may be many different ways in which we can achieve this decomposition, which is best?
      – Different decompositions imply different algorithms and implementations with different characteristics and costs
Bottom-up influences

2. Developing a parallel algorithm and program

- Need to ensure that the concurrency we have discovered is *exploitable*
  - Need to meet our goals (slide 4)
  - Ensure our algorithm maps well onto our **target architecture**
    » memory, communication and computation considerations
    » Ability to exploit locality, load-balance, *etc*..
  - We will also have to consider the constraints imposed by our **parallel programming** and **run-time environment**
    » *e.g.* availability, implementation cost and overheads of different approaches
Parallel speedup

- **Speedup** refers to how many times faster the parallel (or enhanced) solution is to the original:
  \[ \text{Speedup} = \frac{\text{total execution time without enhancement}}{\text{total execution time with enhancement}} \]

- Amdahl's Law

\[
\text{Speedup}^{\text{enhanced}}(f, S) = \frac{1}{(1-f) + \frac{f}{S}}
\]

Originally defined for parallel computers by Gene Amdahl. Here he assumed the speedup $S$ is equal to the number of cores or processors ($N$) and $f$ is the fraction of the program that was infinitely parallelisable. $(1-f)$ represented the totally sequential part.
Parallel speedup

- In the limit, speedup is limited by the fraction of the execution time that cannot be enhanced by parallel execution.
  - As the number of cores, $n$, goes to infinity: \( \text{speedup} = \frac{1}{1-f} \)

- If 10% of the program is serial (fairly common) is it worth developing a complex scalable parallel solution?
  - We need to be careful of diminishing returns
  - We'll return to how this applies to chip multiprocessors in the reading group
Parallel speedup

Amdahl’s Law

Parallel Portion
- 50%
- 75%
- 90%
- 95%

Speedup vs. Number of Processors

Chip Multiprocessors (ACS MPhil)
Parallel speedup – Gustafson's Law

- John Gustafson argued that it is overly pessimistic to assume that the serial execution time increases with problem size, i.e. that the serial fraction remains constant.
- He assumed that the time dedicated to executing the serial part of the program was constant as the problem size grew.
- If we assume this, keep overall execution time constant and increase the problem size, speedup can be approx. linear in \( N \) (number of processors).
- Here we are assuming the serial fraction reduces as problem size increases. This is often a reasonable assumption as the overheads due to parallelism generally decrease with problem size.
Parallel speedup

• In reality, performance can be even worse than predicted by Amdahl's law, e.g. due to:
  – Load balancing, scheduling, communication, I/O overheads

• Or even better than both Amdhal's or Gustafson's law predict, e.g. due to:
  – Cache memory provided by additional cores
  – Helper threads (non-traditional parallelism)
    • e.g. inter-core prefetching. Here we have one compute thread and many prefetching threads. Compute thread migrates between cores
Parallel Efficiency

- Parallel Efficiency, $E(N) = \frac{\text{Speedup}(N)}{N}$
  - Efficiency is a measure of the fraction of time for which each processor is doing useful work
  - **Perfect linear speedup** is the case where speedup is equal to the number of processors and $E(N)=1$
Decomposition

• Aims
  – Expose parallelism
  – Number of tasks should grow with problem size
  – Identifying tasks of a uniform size is often beneficial
  – Aim to decompose the problem in a way that minimises computation and communication
  • Think about CMP memory hierarchy
    – Caches, working set size
    – Ability to localise communications?
    – Trade-offs between recomputing intermediate results, memory usage, communication etc.
Decomposition Design Space

Structure approach around parallel tasks or decomposition of data?

Start: Analyze problem, look for parallelism

**TASKS:**
Organise by tasks (functional decomposition)
- Linear (unstructured or flat)
- Recursive

**DATA:**
Organise by decomposition of data
- Linear
- Recursive
Decomposition Design Space

• Medical imaging
  – Positron Emission Tomography (PET scanner)
  – Need to model how radiation propagates through the body in order to correct images
  – Monte Carlo method
    • Select random starting points and track the trajectory of gamma rays as each ray passes through the body
Decomposition Design Space

• Possible approaches to parallelization:
  
• **Task decomposition**
  – Treat the calculations involved in each trajectory as a separate task

• **Data decomposition**
  – Partition the body into sections and assign different tasks to each section.
  – Trajectories need to be passed between regions at their boundaries
Decomposition Design Space

**TASKS**
- Linear
  - Independent (no interaction between tasks)
  - Data-flow between tasks?
- Recursive
  - Regular [2]
  - Irregular
    - Event-based coordination [3]
    - Repository [4]
  - Divide-and-Conquer [5]
  - Exploratory [6]

**DATA**
- Linear
- Recursive
- Amorphous
  - Geometric decomposition [7]
  - Recursive data structures [8]
  - Amorphous data parallelism [9]

See Mattson book for a similar algorithm structure decision tree, sec 4.2.3
Decomposition Design Space

- *This is not meant to be a definitive decision tree
  - Just meant as a helpful guide

- In practice we do not usually limit ourselves to a single decomposition
  - e.g. climate models
    - Task-driven decomposition into major components followed by data-driven decomposition of each component (models of ocean, atmosphere, land etc.)
    - May also consider transforming our data into periodic or spectral domain first
1. Independent tasks

• Tasks are completely independent
  – Little or no communication is required between tasks, sharing of data is read-only
  – So called *embarrassingly parallel* problems
  – Many problems fall into this category
    • Monte-Carlo techniques, ray-tracing, rendering individual frames of an animation and many other graphics problems, simple flat brute-force searches, systematic evaluation of large design/problem spaces
1. Independent tasks

- In general, such problems may initially require some partitioning of the input data and collecting of results at the end of the computation.
  - In some cases we may initially replicate the global data structure to allow the tasks to execute in parallel
  - The final result is then often computed using a reduction operation
2. Regular data-flow

- An application will often have regular data-flow at a higher level, e.g. a simple linear pipeline, where each stage or task in the pipeline executes in parallel.
  - Signal processing (wireless, radio, radar, ODFM, UMTS, real-time beam former), graphics pipelines, multimedia compression and decompression algorithms, ...
  - More generally the pipeline may fork/join (non-linear pipelines) or simply be a network of components with predictable/static data-flow
  - Wavefront and streaming organisations
2. Regular data-flow

- **Streaming Applications**
  - Process large streams of data
    - Possibly continuous input, but data has limited lifetime
  - Processing consists of a sequence of data transformations
    - Independent filters connected in a stream graph
    - The stream graph is fixed (and structured)
    - Filters are applied in a regular, predictable order
  - Occasional modification of stream structure
    - Dynamic modifications can occur on occasion
    - *e.g.* wireless network may add extra filters in noisy environment to clean up signal
  - Small amount of control information sent between filters
  - High-performance requirements, real-time constraints

[Thies'02]
2. Regular data-flow

- Variable length decoding
- Spatial decoding
  - block decoding in parallel with motion vector decoding
- Temporal decoding
  - all color channels motion compensated in parallel
- Color space conversion and data ordering
3. Event-based co-ordination

• The problem can be decomposed into groups of semi-independent tasks that interact in an irregular fashion
  – Unpredictable timing and data-flow
  – A commonly cited example is discrete-event simulation

Initialise
while (not done)
{
  receive event
  process event
  send events
}
finalize

Event-based co-ordination pattern
[Mattson'04]
Speculative decomposition

- Speculative decomposition exposes parallelism by speculating beyond control dependencies
  - An analogy in the sequential world might be to evaluate all branches of a switch statement in C in parallel before waiting for the switch condition to be resolved
  - Example: Discrete event simulation
    - Guess inputs so we can start processing (try to follow the most promising paths)
    - or just assume we won't receive an input from another part of the simulated system
      - If we make a mistake, we will need to rollback
4. Repository

• Tasks concurrently update (read and write) a centralised data structure in a non-deterministic way
  – We need to provide controls to enforce the atomicity of updates as multiple tasks may attempt to update the same element of the data-structure simultaneously
  – VLSI routing algorithms, databases – e.g. client/server travel reservation system, Delaunay mesh refinement (Ruppert's algorithm)
5. Divide-and-conquer

- The problem is naturally expressed using a recursive divide-and-conquer approach
  - Split provide into smaller subproblems, solving them independently and merging the subsolutions into a solution for the whole problem. Each subproblem can solved directly, or they can be solved by the same divide-and-conquer strategy
  - Common examples: FFT, Cholesky decomposition (computational linear algebra), Quicksort, Mergesort, matrix diagonalisation, computational geometry problems (convex hull and nearest neighbour)
5. Divide-and-conquer

Reproduced from “Patterns for Parallel Programming”, Mattson'04
6. Exploratory

• Here the computation involves searching for a solution or the best solution to a problem
  – We recursively partition the search space and evaluate different configurations until a solution is found, e.g. finding a solution for the peg solitaire game (tree search technique)

• Applications: Discrete optimisation problems (VLSI floorplanning, robot motion planning, test-pattern generation for digital circuits), game playing – chess (e.g. IBM's Deep Blue)
6. Exploratory

• Differences to divide-and-conquer:
  – Not all tasks contribute to the answer
  – Unfinished tasks may be terminated as soon as a solution is found
  – Poor partial solution paths may be abandoned when a better solution has already been found
    • parallel depth-first branch-and-bound search, Grama p.495
  – Search space may be unstructured

Reproduced from Grama book, p.481
Data Decomposition

• 7 - Geometric Decomposition
• 8 - Recursive Data Structures
• 9 - Amorphous Data Parallelism
7. Geometric decomposition

- The data structure provides inspiration for finding parallelism. It can be decomposed into “blocks” that can be updated concurrently.
  - Tasks are associated with each block (or more generally “chunk”) of data.
  - If the tasks only require local data, we have an embarrassingly parallel program (see slide 18.).
- Geometric decomposition is not restricted to purely independent tasks. We associate tasks with chunks or blocks of data, but tasks may require access to non-local points to complete their computation.
7. Geometric decomposition

- Decompose what data precisely?
  - Input, intermediate or output data (Graman p.98)
- Mesh ghost copies
  - (Mattson p.83)
- Techniques for distributing arrays
  - (Graman p.117)
  - Block distributions
  - Cyclic and Block-cyclic distributions
  - Randomized block distributions
7. Geometric decomposition

Figure 3.36 A distribution of the mesh elements to eight processes, by using a graph-partitioning algorithm.

Reproduced from Grama book
8. Recursive data structures

- The problem involves operations on a recursive data structure (e.g. list, tree, graph).
  - Use a divide-and-conquer approach if possible
  - In other cases scope for parallel processing may appear to be limited. In these cases, it is necessary to think of a strategy that may be completely different to a sequential approach.
    - This often involves trading total work performed for additional concurrency.
    - Pointer jumping (JaJa, p.52, Mattson p.97)
    - Euler-tour technique, ear decomposition, and graph contraction
Pointer jumping example

9. Amorphous Data Parallelism

- These algorithms typically operate on a graph
- Any at one time there is a set of active nodes where computations may be performed
  - These computations can often be performed in any order (and with some help, in parallel)
  - The “worklist” may be added to during the computation

Reproduced from "Amorphous Data-Parallelism", Pingali et al.
9. Amorphous Data Parallelism

- Parallelism is limited by the dependencies (overlapping neighborhoods) that exist between elements
  - Can use optimistic parallelisation techniques
    - Speculate that there are no dependencies and roll back if we detect conflicts
    - e.g. Galois system

Reproduced from "Amorphous Data-Parallelism", Pingali et al.
9. Amorphous Data Parallelism

- Examples: Delaunay mesh refinement, Delaunay triangulation, agglomerative clustering, Barnes-Hut, survey propagation
  - See Lonestar benchmarks
Towards an implementation

Now we have exposed some parallelism, what common methods, models or patterns can be used to structure our parallel program?

– Program structuring patterns
  • SPMD
  • Master/Worker
  • Task and Loop parallelism
  • Fork/Join
  • Pipeline or producer-consumer model
  • ...

– Concurrent data structures, e.g. distributed arrays and concurrent FIFOs
Parallel skeletons

• Can we package up implementations of useful patterns of parallel computation and interaction to help programmers?
  – Provide a framework or template (higher order function) parameterised by pieces of code (incl. skeletons) the programmer provides
    • *i.e.* map-reduce, pipeline, grid-structured problems, master-worker (farm), divide and conquer...
  – Programmer just specifies skeleton + functions
    • Interaction, communication *etc.* is handled by skeleton
Performance considerations

- Scientific supercomputing applications typically run in a very controlled environment
  - Fixed number of processors
  - One application runs at a time
- Chip Multiprocessors
  - Number of cores varies between platforms
  - Applications run in a multiprogrammed environment
- Achieving a speedup for each additional available core requires finer-grain tasks to be exposed
Performance considerations

A problem broken into 64 tasks and run on 32 or 48 cores will take 2 time units in both cases.

Reproduced from “Carbon: Architectural Support for Fine-Grained Parallelism on Chip Multiprocessors”, Kumar/Hughes/Nguyen, ISCA'07
Performance results, don't be fooled!

- Misleading ways to improve your results:
  - Report the performance of only the parallel kernel rather than the entire application
  - Scale up the problem size with the number of processors but omit any mention of this fact
  - Simply scale the performance of a smaller parallel system linearly as a prediction of the performance of a larger system
  - Use a poor base case: disregard best scalar solutions, compare to an old architecture or program. If all else fails, compare to a restricted version of your new architecture or program
  - Don't report performance, report utilisation!
  - Don't provide any quantitative numbers, just talk authoritatively and be careful not to take questions.
  - And .... normalise results to hide details, omit any mention of power consumption, assume unrealistic memory bandwidth......

see David Bailey's article (wiki)