∼ Topic VI ∼

Languages for Concurrency and Parallelism

This lecture will take place out-of-order on 15 May 2014.

These are the additional slides for it.

Sources of parallel computing

Five main sources:

1. Theoretical models: PRAM, BSP (complexity theory),
   CSP, CCS, π-calculus (semantic theory),
   Actors (programming model).
2. Multi-core CPUs (possibly heterogeneous—mobile phones).
3. Graphics cards (just unusual SIMD multi-core CPUs).
4. Supercomputers (mainly for scientific computing).
5. Cluster Computing, Cloud Computing.

NB: Items 2–5 conceptually only differ in processor-memory communication.

Language groups

1. Theoretical models (PRAM, π-calculus, Actors, etc.).
2. C/C++ and roll-your-own using pthreads.
3. Pure functional programming ('free' distribution).
4. [Multi-core CPUs] Open/MP, Java (esp. Java 8), Open/MP, Cilk, X10.
5. [Graphics cards] CUDA (Nvidia), OpenCL (open standard).
6. [Supercomputers] MPI.
7. [Cloud Computing] MapReduce, Hadoop, Skywriting.
8. [On Chip] Verilog, Bluespec.

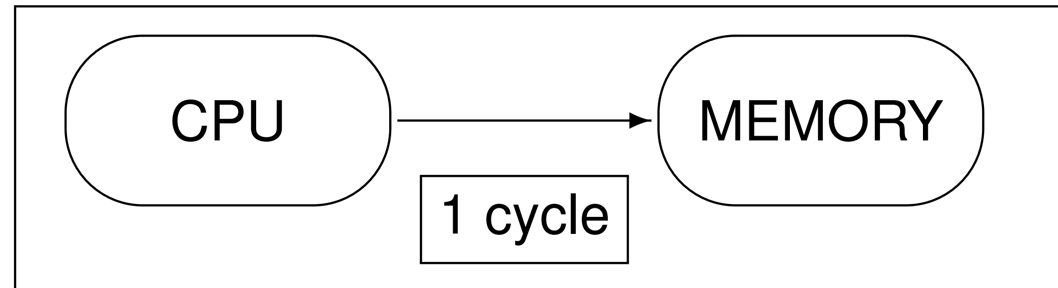
NB: Language features may fit multiple architectures.

Painful facts of parallel life

1. Single-core clock speeds have stagnated at around 3GHz for the last ten years. Moore's law continues to give more transistors (hence multi-core, many-core, giga-core).
2. Inter-processor *communication* is far far far more expensive than *computation* (executing an instruction).
3. Can't the compiler just take my old C/Java/Fortran (or ML/Haskell) program and, *you know*, parallelise it? Just another compiler optimisation? NO! (Compiler researchers' pipe-dream/elephants' graveyard.)

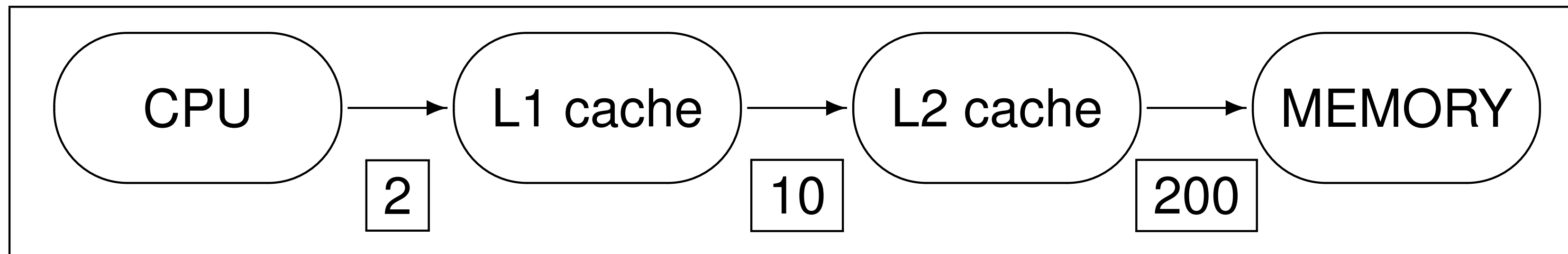
Takeaway: optimising performance requires exploiting parallelism, you'll have to program this yourself, and getting it wrong gives slow-downs and bugs due to races.

A programmer's view of memory



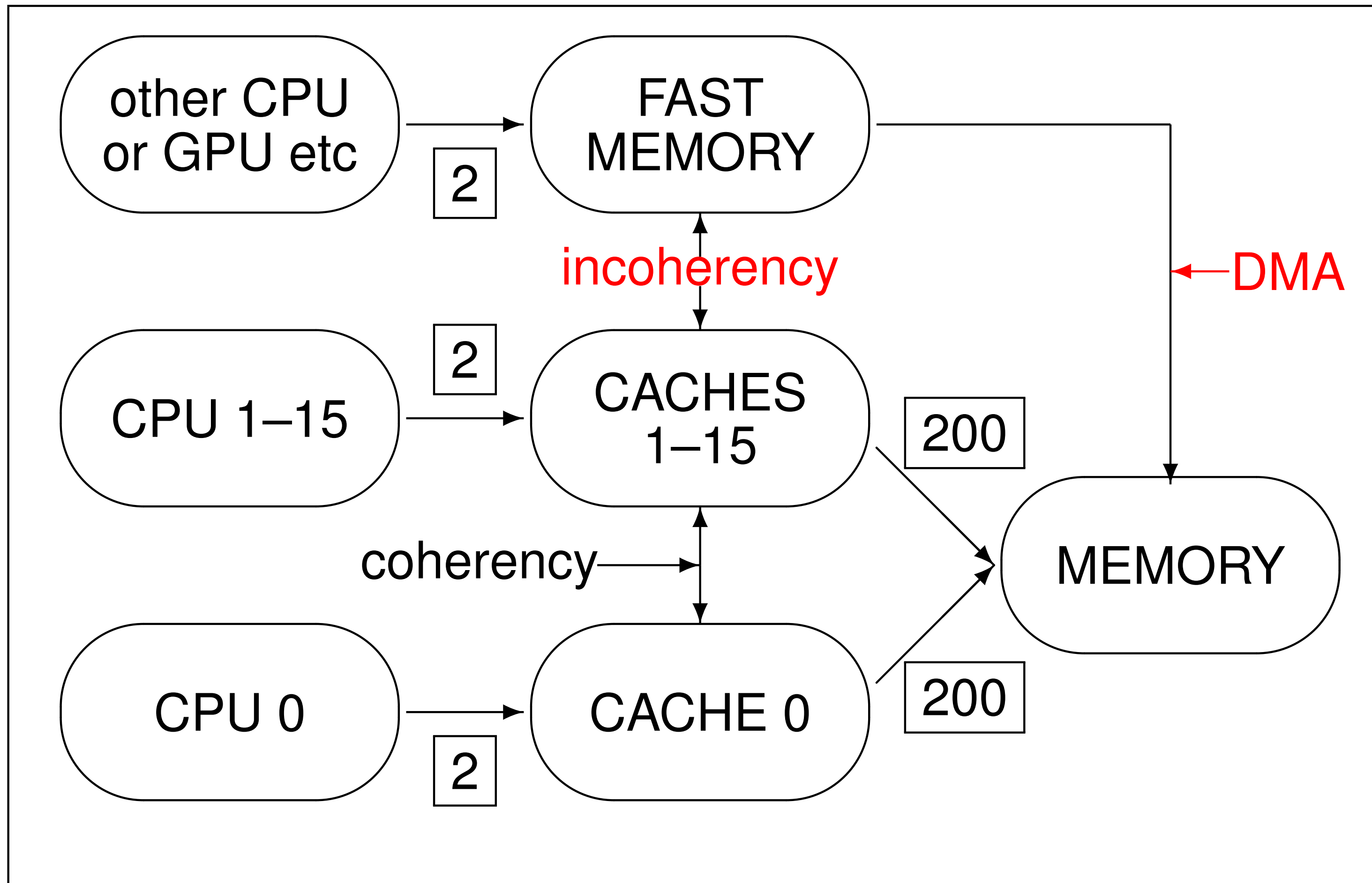
(This model was pretty accurate in 1985.)

A 2004-era single-core view of memory and timings

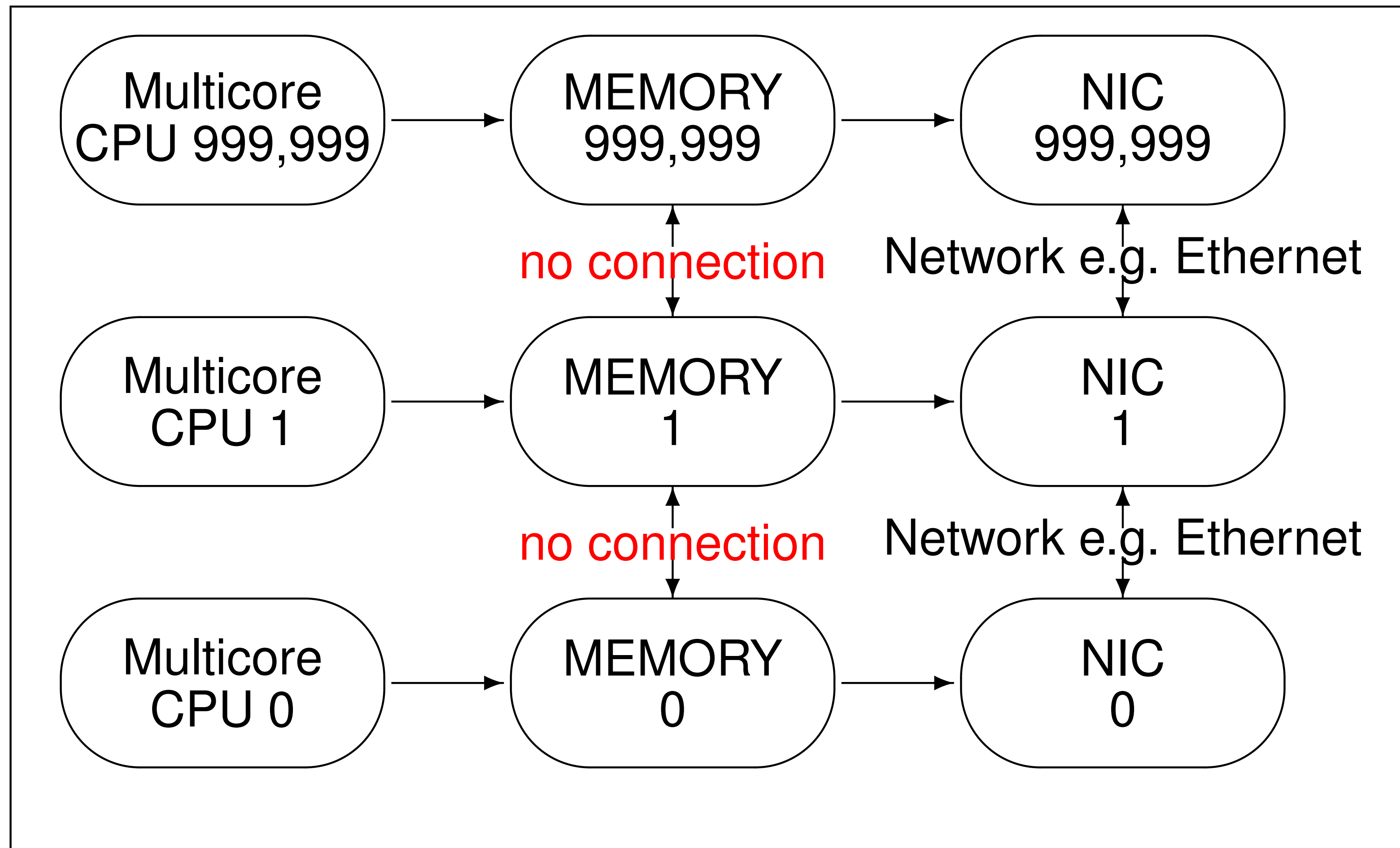


Multi-core-chip memory models

Today's model (cache simplified to one level):



A Compute Cluster or Cloud-Computing Server



(The sort of thing which Google uses.)

Lecture topic: what *programming abstractions*?

- We've got a large (and increasing) number of processors available for use within each 'device'
- This holds at multiple levels of scale (from on-chip to on-cloud). "Fractal"
- Memory is local to processor units (at each scale)
- Communication (message passing) between units is much slower than computation.

Question: what are good programming abstractions for a system containing lots of processors?

Answer: rest of this lecture.

What hardware architecture tells us

- Communication latency is far higher than instruction execution time (2–6 orders of magnitude)
- So, realistically a task needs to have need at least 10^4 instructions for it to be worth moving to another CPU.
- *Long-running independent computations* fit the hardware best.
- “Shared memory” is an illusion. At the lowest level it is emulated by message passing in the cache-coherency protocol.
- Often best to think of multi-core processors as distributed systems.

Communication abstractions for programming

- “Head in sand”: What communication – I’m just using a multi-core CPU?
- “Principled head in sand”: the restrictions in my programming language means I can leave this to someone else (or even the compiler).
- Just use TCP/IP.
- Shared memory, message passing, RMI/RPC?
- Communication is expensive, expose it to programmer (no lies about ‘shared memory’).

Ask: language $\Rightarrow$ programmer model of communication?

Concurrent, Parallel, Distributed

These words are often used informally as near synonyms.

- Distributed systems have separate processors connected by a network, perhaps on-chip (multi-core)?
- ‘Parallel’ suggests multiple CPUs or even SIMD, but “parallel computation” isn’t clearly different from “concurrency”.
- Concurrent behaviour *can* happen on a single-core CPU (e.g. Operating System and threads), Theorists often separate ‘true concurrency’ (meaning parallel behaviour) from ‘interleaving concurrency’.

SIMD, MIMD

Most parallel systems nowadays are MIMD.

GPUs (graphical processor units) are a bit of an exception; several cores execute the same instructions, perhaps conditionally based on a previous test which sets per-processor condition codes.

Programming Languages for GPUs (OpenCL, CUDA) emphasise the idea of a single program which is executed by many tasks. A program can enquire to find out the numerical value of its task identifier, originally its (x, y) co-ordinate, to behave differently at different places (in addition to having separate per-task pixel data).

Theoretical model – process algebra

CCS, CSP, Pi-Calculus (calculus = "simple programming language"). E.g.

Atomic actions α, $\overline{\alpha}$, can communicate with each other or the world (non-deterministically if multiple partners offered). Internal communication gives special internal action τ.

Behaviour $p ::= 0 \mid \alpha.p \mid p + p \mid p|p \mid X \mid \mathit{rec}\ X.p$
(Deadlock, prefixing, non-determinism, parallelism, recursive definitions, also (not shown) parameterisation/hiding and value-passing.)

Typical questions: "is $\alpha.0|\beta.0$ the same as $\alpha.\beta.0 + \beta.\alpha.0$" and "what does it mean for two behaviours to be equal"

Part II course.

Theoretical model – PRAM model

PRAM: parallel random-access machine.

N shared memory locations and P processors (both unbounded); each processor can access any location in one cycle.

Execute instructions in lock-step (often SIMD, but MIMD within the model): fetch data, do operation, write result.

Typical question: "given n items can we sort them in $O(n)$ time, or find the maximum in $O(1)$ time"

BSP (bulk-synchronous parallel) model refines PRAM by adding costs for communication and synchronisation.

New Part II course in 2014/15.

Oldest idea: Threads

Java threads – either extend Thread or implement Runnable:

```
class PrimeRun implements Runnable {
    long minPrime;
    PrimeRun(long m) { minPrime = m; }
    public void run() {
        // compute primes larger than minPrime
    }
}
...
p = new PrimeRun(143);  // create a thread
new Thread(p).start();  // run it
```

Posix's pthreads are similar.

Threads, and what's wrong with them

- Need explicit synchronisation. Error prone.
- Because they're implemented as library calls, the compiler (and often users) cannot work out where they start and end.
- pthreads as OS-level threads. Need context switch. Heavyweight.
- Various lightweight-thread systems. Often non-preemptive. Blocking operations can block all lightweight operations sharing the same OS thread.
- Number of threads pretty hard-coded into your program.

Language support: Cilk

Cilk [example from Wikipedia]

```
cilk int fib (int n)
{  int x,y;
   if (n < 2) return n;
   x = spawn fib (n-1);
   y = spawn fib (n-2);
   sync;
   return x+y;
}
```

Compiler/run-time library can manage threads. Neat implementation by “work stealing”. Can adapt to hardware.

X10 (IBM) adds support for partitioned memory.

Language support: OpenMP

OpenMP [example from Wikipedia]

```
int main(int argc, char *argv[]) {
    const int N = 100000;
    int i, a[N];
    #pragma omp parallel for
    for (i = 0; i < N; i++)
        a[i] = 2 * i;
    return 0;
}
```

The directive “omp parallel for” tells the compiler “it is safe to do the iterations in parallel”.
Fortran “FORALL INDEPENDENT”.

Clusters/Cloud Computing

Memory support for threads, Cilk, OpenMP centres around a shared address space. (Even if secretly multi-core machines behave like distributed machines.)

What about clusters? Cloud Computing?

More emphasis on message-passing . . .

Software support for message passing: MPI

MPI = Message Passing Interface [nothing to do with OpenMP]

“de facto standard for communication among processes that model a parallel program running on a distributed memory system.” [no shared memory].

Standardised API calls for transferring data and synchronising iterations. Message passing is generally synchronous, suitable for repeated sweeps over scientific data.

Emphasis on message passing (visible and expensive-looking to user) means that MPI programs can work surprisingly well on multi-core, because they encourage within-core locality.

Software support for message passing: Erlang

Shared-nothing language based on the actor model (asynchronous message passing).

Dynamically typed, functional-style (no assignment).

Means tasks can just commit suicide if they feel there's a problem and someone else fixes things, including restarting them

Relatively easy to support hot-swapping of code.

Cloud Computing (1)

Can mean either “doing one computer’s worth of work on a server instead of locally”. Google Docs. Or . . .

Cloud Computing (2)

. . . massively parallel combinations of computing, e.g. MapReduce invoked by a search engine.

MapReduce can match a search term against many computers (Map) each holding part of Google index of words, and then combine these result (Reduce).

Reduce here means parallel reduce (tree-like, logarithmic cost), not *foldl* or *foldr* from ML.

Functional style (idempotency) useful for error resilience (errors happen often in big computations). Try to ensure computation units are larger than cost of transmitting arguments and results. (also: Skywriting project in Cambridge)

Embarrassingly Parallel

Program having many separate sub-units of work (typically more than the number of processors) which

- do not interact (no communication between them, not even via shared memory)
- are large

Example: the map part of MapReduce.

Functional Programming

In pure functional programming every tuple (perhaps an argument list to an application) can be evaluated in parallel.

So functional programming is embarrassingly parallel?

Not in general (i.e. not enough for compilers to be able to choose the parallelism for you). Need to find sub-executions with X

- little data to transfer at spawn time (because it needs copying, even if memory claims to be shared);
- a large enough unit of work to be done before return

Probably only certain stylised code.

Garbage Collection

While we're talking about functional programming, and as garbage collection has previously been mentioned . . .

Just how do we do garbage collection across multiple cores?

- Manage data so that data structures do not move from one processor to another?
- "Stop the world" GC with one big lock doesn't look like it will work.
- *Parallel GC:* use multiple cores for GC). *Concurrent GC:* do GC which the mutator (user's program) is running. Hard?
- Incremental? Track imported/exported pointers?

Java 8: Internal vs External iteration

Can't trust users to iterate over data. They start with

```
for (i : collection)
{   // whatever
}
```

and then get lazy. Do we want to write this?

```
for (k = 0; k<NUMPROCESSORS; k++}
{   spawn for (i : subpart(collection,k))
    {   // whatever
    }
}
sync;
// combine results from sub-parts here
```

Internal vs External iteration (2)

Previous slide was *external iteration*. It's hard to parallelise (especially in Java where iterators have shared mutable state).

The Java 8 Streams library encourages *internal iteration* – keep the iterator in the library, and use ML-like stream operation to encode the body of the loop

```
maxeven = collection.toStream().parallel()
                               .filter(x -> x%2 == 0)
                               .max();
```

The library can optimise the iteration based on the number of threads available (and do a better job than users make!). The Java 8 API ensures that a Stream pipeline like the above only traverses the data once.