Advances in Programming Languages

APL11: Concurrency

David Aspinall
(including slides by Ian Stark)

School of Informatics
The University of Edinburgh

Monday 15 February 2010
Semester 2 Week 6

Techniques for concurrency

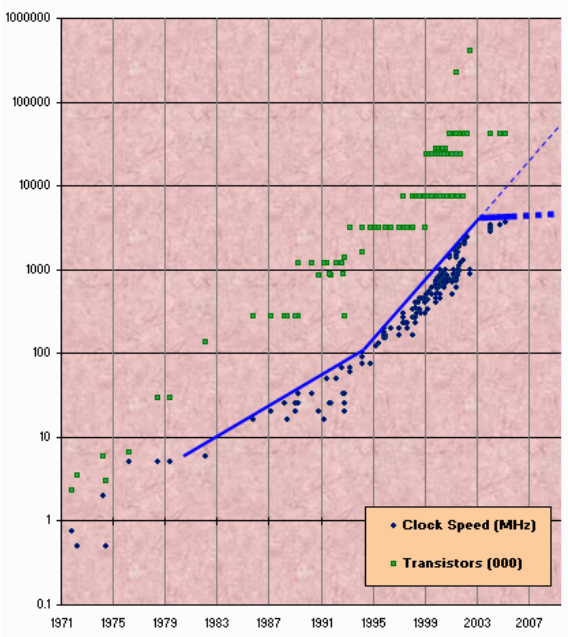
This is the first of a block of lectures looking at programming-language techniques for managing concurrency.

- Introduction, basic Java concurrency
- Concurrency abstractions
- Concurrency in some other languages
- Guest lecture(s) TBC

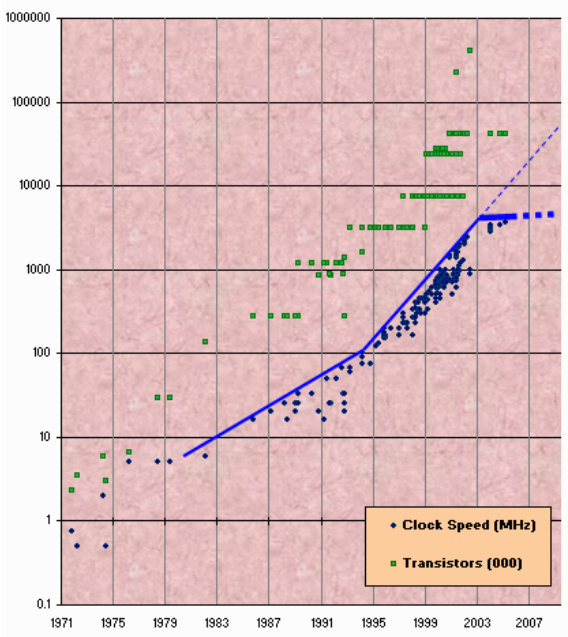
Outline

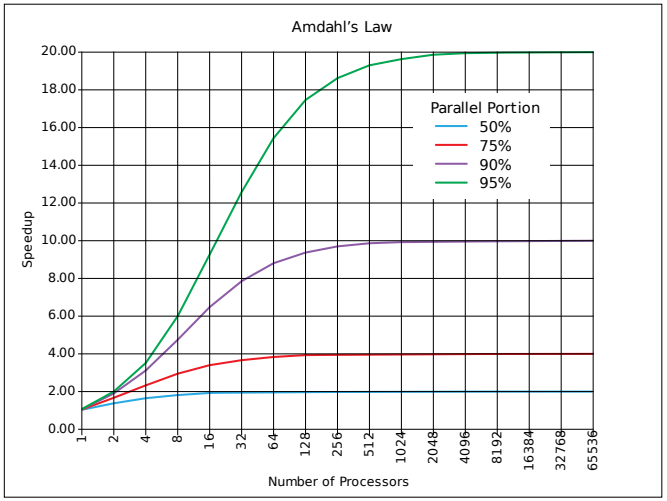
Outline

The free lunch is over



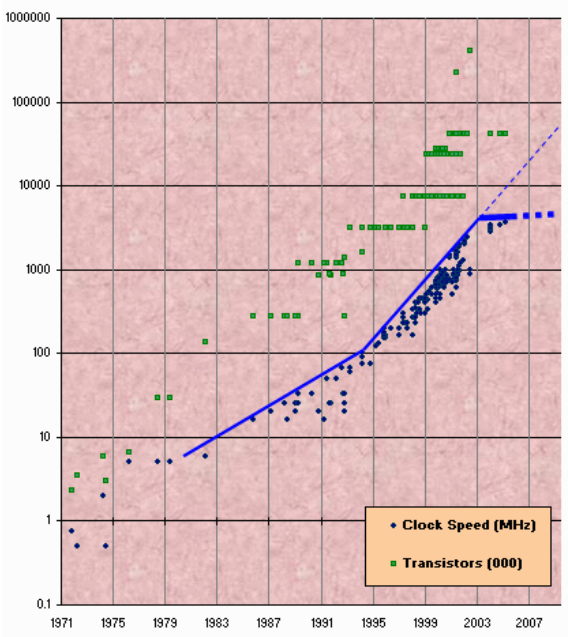
The free lunch is over

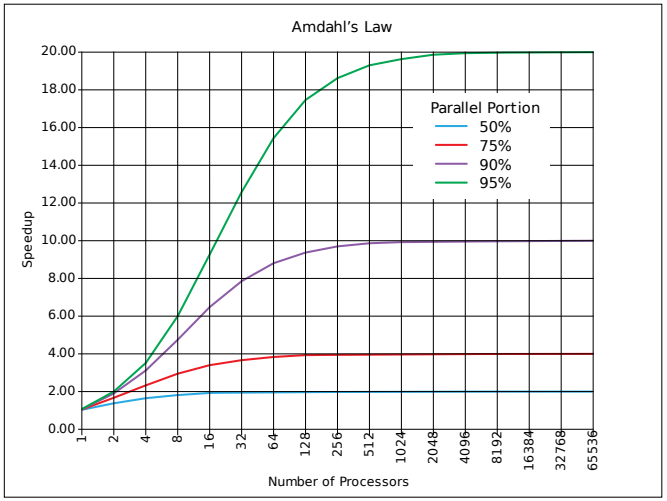




For the past 30 years, computer performance has been driven by Moore's Law; from now on, it will be driven by Amdahl's Law. (Doron Rajwan, Intel Research Scientist).

The free lunch is over





For the past 30 years, computer performance has been driven by Moore's Law; from now on, it will be driven by Amdahl's Law. (Doron Rajwan, Intel Research Scientist).

Concurrency is the next major revolution in how we write software. . . . The vast majority of programmers today don't grok concurrency, just as the vast majority of programmers 15 years ago didn't yet grok objects. (Herb Sutter, Microsoft, C++ ISO chair).

Why write concurrent programs?

Concurrent programming is about writing code that can handle doing more than one thing at a time. There are various motivations, such as:

- Separation of duties (fetch data, render images)
- Efficient use of mixed resources (disk, memory, network)
- Responsiveness (GUI, hardware interrupts, managing mixed resources)
- Speed (multiprocessing, hyperthreading, multi-core, many-core)
- Multiple clients (database engine, web server)

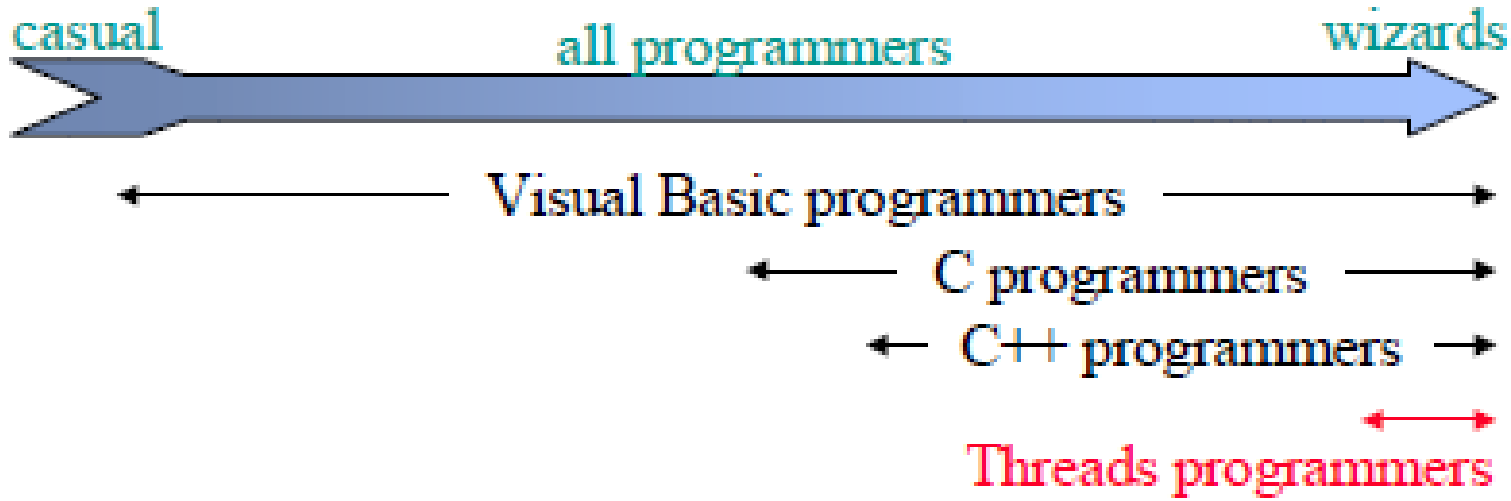
Why write concurrent programs?

Concurrent programming is about writing code that can handle doing more than one thing at a time. There are various motivations, such as:

- Separation of duties (fetch data, render images)
- Efficient use of mixed resources (disk, memory, network)
- Responsiveness (GUI, hardware interrupts, managing mixed resources)
- Speed (multiprocessing, hyperthreading, multi-core, many-core)
- Multiple clients (database engine, web server)

Note that the aims here are different to *parallel programming*, which is generally about the efficient (and speedy) processing of large sets of data.

What's Wrong With Threads?



- **Too hard for most programmers to use.**
- **Even for experts, development is painful.**

John Ousterhout: *Why Threads Are A Bad Idea (for most purposes)*
USENIX Technical Conference, invited talk, 1996

It's hard to walk and chew gum

Concurrent programming offers much, including entirely new problems.

Interference — code that is fine on its own may fail if run concurrently.

Liveness — making sure that a program does anything at all.

Starvation — making sure that all parts of the program make progress.

Fairness — making sure that everyone makes reasonable progress.

Scalability — making sure that more workers means more progress.

Safety — making sure that the program always does the right thing.

Specification — just working out what is "the right thing" can be tricky.

Concurrent programming is hard, and although there is considerable research, and even progress, on how to do it well, it is often wise to avoid doing it (explicitly) yourself unless absolutely necessary.

Processes, Threads and Tasks

All operating systems and many programming languages provide for concurrent programming, usually through a notion of *process*, *thread* or *task*.

The idea is that a process/thread captures a single flow of control. A concurrent program or environment will have many of these at a time.

A *scheduler* manages which threads are executing at any time, and how control passes switches between them.

There are many design tradeoffs here, concerning memory separation, mutual protection, communication, scheduling, signalling, ...

Usually processes are "heavyweight" (e.g., separate memory space), threads "lightweight" (shared memory) and tasks lightest. But usage is not precise or consistent. Complete systems will include multiple layers of concurrency.

Critical Sections

A central issue with multiple explicit threads is to avoid interference through shared memory.

```
void moveBy(int dx, int dy) {
    System.out.println("Moving by "+dx+","+dy);
    x = x+dx;
    y = y+dy;
    System.out.println("Completed move");
}

void moveTo(int newx, int newy) {
    System.out.println("Moving to "+newx+","+newy);
    x = newx;
    y = newy;
    System.out.println("Completed move");
}
```

Critical Sections

A central issue with multiple explicit threads is to avoid interference through shared memory.

```
void moveBy(int dx, int dy) {
    .
    .
  x = x+dx;
  y = y+dy;
    .
    .
}
```

```
void moveTo(int newx, int newy) {
    .
    .
  x = newx;
  y = newy;
    .
    .
}
```

Critical Sections

A central issue with multiple explicit threads is to avoid interference through shared memory.

```
void moveBy(int dx, int dy) {
    .
    .
  x = x+dx;
  y = y+dy;
    .
    .
}
```

```
void moveTo(int newx, int newy) {
    .
    .
  x = newx;
  y = newy;
    .
    .
}
```

Because both methods access the fields x and y, it is vital that these two *critical sections* of code are not executing at the same time.

Interference can result in erroneous states — perhaps violating object invariants and causing crashes, or possibly going undetected until later. Debugging can be difficult because scheduling is often non-deterministic.

Locks and more

There are many ways to ensure critical sections do not interfere, and refinements to make sure that these constraints do not disable desired concurrency.

- Locks
- Mutexes
- Semaphores
- Condition variables
- Monitors *etc.*

Constructs such as these can ensure *mutual exclusion* or enforce *serialisation* orderings.

They are implemented on top of other underlying locking mechanisms, either in hardware (test-and-set, compare-and-swap,...) or software (spinlock, various busy-wait algorithms).

Outline

Concurrency in Java

Java supports concurrent programming as an integral part of the language: threading is always available, although details of its implementation and scheduling will differ between platforms.

The class Thread encapsulates a thread, which can run arbitrary code. Threads have unique identifiers, names, and integer priorities.

Parent code can spawn multiple child threads, and then wait for individual children to terminate or (co-operatively) interrupt them.

```
class WatcherThread
      extends Thread {
 public void run() {
  // watch things going on
   ...
 }
}
```

```
class WorkerJob
  implements Runnable {
 public void run() {
   // do some work
   ...
 }
}
```

```
WatcherThread watcher =
    new WatcherThread();
WorkerJob job =
    new WorkerJob();
watcher.start();
Thread work =
    new Thread(job).start();
...
work.join();
watcher.interrupt();
```

Synchronized methods

Java provides mutual exclusion for critical sections through the **synchronized** primitive.

```
synchronized void moveBy(int dx, int dy) {
    System.out.println("Moving by "+dx+","+dy);
    x = x+dx;
    y = y+dy;
    System.out.println("Completed move");
}

synchronized void moveTo(int newx, int newy) {
    System.out.println("Moving to "+newx+","+newy);
    x = newx;
    y = newy;
    System.out.println("Completed move");
}
```

Synchronized methods

Two move methods cannot now execute at the same time on the same object.

```
synchronized void moveBy(int dx, int dy) {
    System.out.println("Moving by "+dx+","+dy);
    x = x+dx;
    y = y+dy;
    System.out.println("Completed move");
}

synchronized void moveTo(int newx, int newy) {
    System.out.println("Moving to "+newx+","+newy);
    x = newx;
    y = newy;
    System.out.println("Completed move");
}
```

Synchronized methods

Each **synchronized** method must *acquire* a lock before starting and *release* it when finished.

```
synchronized void moveBy(int dx, int dy) {
    System.out.println("Moving by "+dx+","+dy);
    x = x+dx;
    y = y+dy;
    System.out.println("Completed move");
}

synchronized void moveTo(int newx, int newy) {
    System.out.println("Moving to "+newx+","+newy);
    x = newx;
    y = newy;
    System.out.println("Completed move");
}
```

Synchronized blocks

Every Java object has an implicit associated lock, used by its **synchronized** methods. This can also be used to arrange exclusive access to any block of code:

```
void moveBy(int dx, int dy) {
  System.out.println("Moving by "+dx+","+dy);
  synchronized(this) {
    x = x+dx;              // Only this section of the
    y = y+dy;              // code is critical
  }
  System.out.println("Completed move");
}
```

The locking object need not be **this**. Sometimes a lock may be better on a contained (or "owned") object, or may be needed on a containing (or "owning") object.

Condition variables

Java refines critical regions with basic *condition variables*.

- Synchronized code that finds things are not suitable for it to proceed may wait() on the condition variable associated with its lock. This blocks the code and releases the lock.
- Another thread can acquire the lock and do some work. Because this may change the situation for the other thread, it should notify() or notifyAll() other threads of this.
- Threads waiting on the condition variable will be made runnable again, and can check to see if they are now ready to proceed.

Having threads block saves on busy waiting, and ensures that they only wake when there is something to check.

A simple blocking method

```
class Pigeonhole {

  private Object contents = null;

  synchronized void put (Object o) {

    while (contents != null)    // Wait until the pigeonhole is empty
      try { wait(); }
      catch (InterruptedException ignore) { return; }

    contents = o;               // Fill the pigeonhole
    notifyAll();                // Tell anyone who might be interested
  }
  ...
}
```

Outline

Summary

- Concurrency is essential (efficiency, responsiveness, speed)
- But can be tricky (interference, deadlock, fairness, . . .)
- Concurrency can be in the language, in libraries, or in both
- Java provides shared-memory concurrency in the language:
  - locks and critical regions with **synchronized**
  - condition variables with wait and notify
  - explicit spawning of multiple threads
- Java provides further concurrency abstractions in a library: java.util.concurrent.

Homework

Before the next lecture:

- Review the concurrency primitives provided by Java
  - Look at the methods in the Thread class
  - Understand *intrinsic* locking
- Read the Java Concurrency tutorial
  - http://java.sun.com/docs/books/tutorial/essential/concurrency
- Write a Pigeon Fancier program that:
  - Has a fixed number of pigeon-holes which are emptied by some dedicated pigeon-handler threads releasing pigeons after random delays;
  - Has a single thread which stuffs new pigeons back into empty holes;
  - Make sure the pigeon stuffer doesn't wait too long for any hole to become empty.