SPP (Synchro et Prog Parallèle)

Unit 1: Introduction & Motivation

François Taïani

What is parallel computing?

- Multiple executions of one or several programs
  - ➔ occurring “at the same time” (more on this)
  - ➔ interacting with each other in some ways
- Different models of interaction
  - ➔ shared memory
  - ➔ shared resources (printer, hardware bus)
  - ➔ through messages / events
- In this course
  - ➔ we focus on shared memory
- Link to distributed computing: strong
  - ➔ but not the same focus & problems

Why Parallel Computing?

- Parallel computing = much harder than sequential
  - ➔ So why bother?

- Take 5 minutes to think of
  - ➔ 3 examples of domain that use parallel computing
  - ➔ 3 reasons why parallel computing is useful

Why Parallel Computing?

- Examples
  - ➔ 3D graphics on GPUs (Graphical Processing Units)
  - ➔ web servers (100s of concurrent requests)
  - ➔ scientific simulations (High Performance Computing)

- Reasons
  - ➔ key reason 1 go faster (e.g. multi-core CPUs and GPUs)
  - ➔ to maximise resource utilisation e.g. CPU during I/O
  - ➔ to structure inherently concurrent applications: GUI, RT

Spotlight on GP-GPU

GPU: Highly parallel architecture

- several 100s of cores on one card
- now used for more than 3D: finance, e-science
- highly specialized hardware
  - ➔ all cores = same instruction flow or big penalty
  - ➔ different types of memory with different latencies
- known as SIMD archi (Single Inst. Multiple Data)
  - ➔ opposite is MIMD (typical multi core CPU)
- Specialised library for scientific computation
  - ➔ e.g. CUDA from NVidia

Spotlight on I/O

- Multi-core processors quite recent
  - ➔ for a long time only very high-end machines
- So why parallel computing? Because of I/O
- Example (pseudo code)
  - ➔ for i = 1 to 1000
    read file_input$_i$
    compute simulation based on file_input$_i$
    save simulation results to file_results$_i$
    endfor
  - ➔ Question: utilisation rate of CPU?
    (what kind of data do you need)

Spotlight on I/O (cont.)

- Example (pseudo code)
  - for i = 1 to 1000
    read file_input$_i$
    compute simulation based on file_input$_i$
    save simulation results to file_results$_i$
    endfor

- Question: utilisation rate of CPU? Assume
  - each input and result file = 200 kB
  - read / write bandwidth of hard-drive is 20 MB / s
  - simulations takes 80,000,000 assembly instructions
  - CPU frequency is 4 GHz

Spotlight on I/O (cont.)

- Some comments
  - ➔ big simplification: only disk bandwidth, no latency
  - ➔ in practice OS would ensure caching of reused files
- Main message: moving data much slower than CPU!
  - ➔ for same “complexity” of operations

```
AdminMacBook ftaiani [ftaiani] $ ls -hl bigfile
-rw-r--r--  1 ftaiani  staff    95M  5 Jan  2010 bigfile

AdminMacBook ftaiani [ftaiani] $ time cp bigfile bigfile2
cp bigfile bigfile2  0.00s user 0.24s system 4% cpu 6.008 total
```

Spotlight on I/O (cont.)

- Solution: parallelise program
  - ➔ for i = 1 to 1000 **do in parallel**
    read file_input$_i$
    compute simulation based on file_input$_i$
    save simulation results to file_results$_i$
    endfor
  - ➔ Question: What is the new CPU utilisation rate?

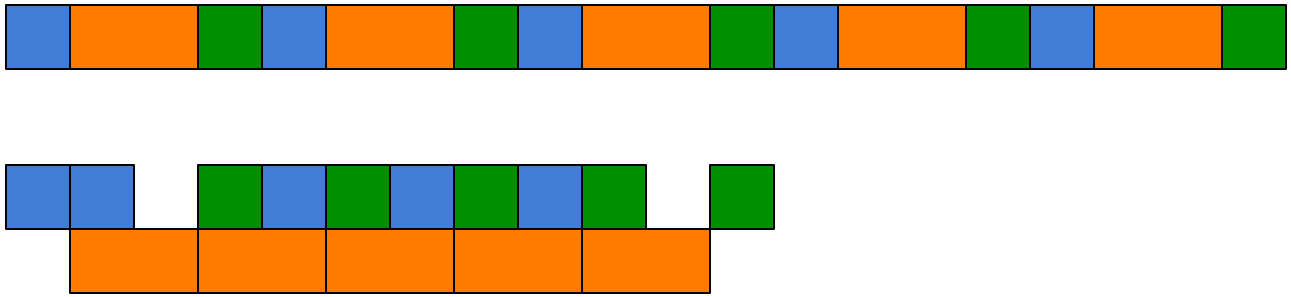
Spotlight on I/O (cont.)

- Data bottleneck found in many applications
  - ➔ web-servers: many requests, data on HD or DB
  - ➔ database servers: same issue
  - ➔ data-heavy application: reading from files
  - ➔ the reason why applications takes time to start!

- one CPU + several flows of executions = **concurrent programming**
  - ➔ illusion of “parallelism” provided by OS (see OS course)
  - ➔ same problems and same programming techniques

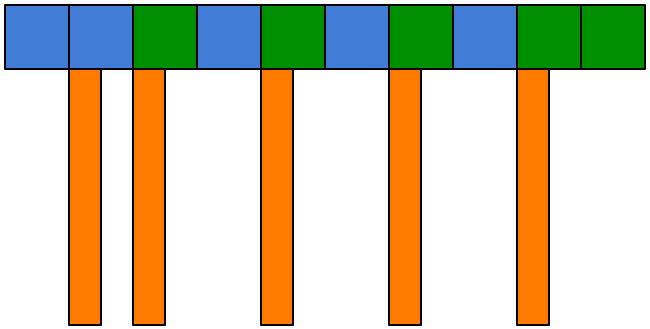
Limits of Parallelism

- Simulation example



…

- Can we go arbitrarily faster with more CPUs?
  - ➔ No, disk becomes bottleneck



Limits (cont) Amdahl's law

- Captures impact of **CPU bottlenecks**
- Idea: sequential execution of program (1 processor)
  - ➔ total execution time with 1 processor T_{seq}
  - ➔ proportion that cannot be parallelised: α
- Question:

  New "best possible" duration with n processors?

$$T_{par}(n) = \alpha\, T_{seq} + (1-\alpha)\, T_{seq} / n$$

$$T_{par}(n) = [\, \alpha\, (n-1) + 1\,] / n \times T_{seq}$$

- Best speedup: $n / [\, \alpha\, (n-1) + 1\,]$

Model: Processes

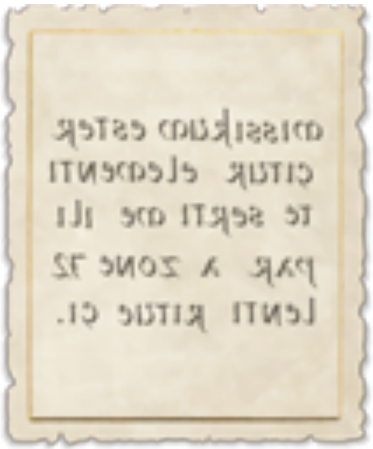
- To reason about parallelism: programming model
- Based on the notion of **processes**
  - ➔ = one **sequential** execution of one program
- Process ≠ program ≠ processor
  - ➔ program = code = recipe
  - ➔ process = worker, active entity making progress
  - ➔ one program can be executed several times

```
AdminMacBook ftaiani [ftaiani] $ ps -Uftaiani | grep gv
 2518 ttys000    0:00.07 gv
 2712 ttys002    0:00.05 gv
 2745 ttys003    0:00.00 grep gv
AdminMacBook ftaiani [ftaiani] $ kill 2518
AdminMacBook ftaiani [ftaiani] $ kill 2712
```

A Metaphor

The recipe =
the program (on HD)

The cauldron =
the memory



The potion =
the data / state

The cook = the process

Where is the processor?

Shared memory

- One cauldron (memory) for several cooks (processes)
  - ➔ usually one singe recipe (program)
  - ➔ but each cook might use a different bit of the recipe

Vocabulary

- In this module process = thread (of execution)
  - ➔ “process” more common in algorithms, theory
  - ➔ “thread” used in programming
- In operating systems: meaning can be different
  - ➔ process = thread(s) + address space + program
  - ➔ 1 thread = monothreaded
  - ➔ > 1 threads = multithreaded
  - ➔ but in some OS, each thread receives their own PID…

Multithreading in Java

Based on the class java.lang.Thread

Two ways to define what a Thread should do

- By extending the class Thread
  - ➔ class MyThread extends Thread
  - ➔ implement the method run()
  - ➔ create an instance of your class: t = new MyThread()
  - ➔ start your thread: t.start()

Multithreading in Java

- or implement the Runnable interface
  - ➔ class MyRunnable implements Runnable
  - ➔ implement method run()
  - ➔ create an instance of your class: r = new MyRunnable()
  - ➔ start a thread pointing to your instance:
    new Thread(r).start()
  - ➔ (or call new Thread(this).start() in your constructor)

Analysing a simple example

- Consider the following example (pseudo code)

```
shared var count int = 0

thread t1 is
var a int
begin
  for a=1 to 10,000,000
    count++
  endfor
end
```

```
thread t2 is
var b int
begin
 for b=1 to 10,000,000
    count++
  endfor
end
```

- Two questions:
  - ➔ How to implement it in Java?
  - ➔ What is the value of count after the program executes?

First a simplification

- Notice how both thread execute the same code
  - ➔ just replace b by a. Both are private variables.

```
shared var count int = 0

function f is
var a int
begin
  for a=1 to 10,000,000
    count++
  endfor
end
```

```
thread t1 is f()
thread t2 is f()
```

In Java

- Java has no functions, but classes so
  - ➔ define our own extension of Thread
  - ➔ define run() to be the same as function f
  - ➔ global variable count implemented as a static variable

```
public class MultiThreadedCounter extends Thread {

   static long count = 0;

   public void run() {
      for(int i=1; i<=10000000; i++) {
         MultiThreadedCounter.count++ ;
      } // EndFor
   } // EndMethod run

   public static void main(String[] args) {
      Thread t1 = new MultiThreadedCounter();
      Thread t2 = new MultiThreadedCounter();
      t1.start();
      t2.start();
   }

} // EndClass MultiThreadedCounter
```

What is the result of this?

- Quizz: What is the value of count after t1 & t2 end?

- Experiment: running the program

So what is happening?

- Best way to know: let's look at the byte code
  - ➔ javap -c MultiThreadedCounter : disassemble .class

```
public void run();
  Code:
   Stack=4, Locals=2, Args_size=1
   0: iconst_1
   1: istore_1
   2: iload_1
   3: ldc #2; //int 10000000
   5: if_icmpgt 22
   8: getstatic #3; //Field count:J
   11:  lconst_1
   12:  ladd
   13:  putstatic #3; //Field count:J
   16:  iinc  1, 1
   19:  goto  2
   22:  return
```

What we have

- A case of contention: 2 threads, 1 resource (count)
  - ➔ resource corrupted because access not controlled
- Reason: lack of atomicity (more on this later)
  - ➔ Even basic instructions are not necessarily atomic
  - ➔ ++ is not atomic in java: decomposed in steps
  - ➔ these steps might overlap with that of other threads!
- We need mechanisms to bring order to this
  - ➔ need for synchronisation
- That is what we'll study in the rest of the module

Summary

- Parallel & concurrent computing extremely important
  - ➔ to exploit multi-core architectures
  - ➔ to mask I/O slow latencies and bandwidth
  - ➔ to program inherently concurrent applications
- Difference between parallelism and concurrency
- Parallelism / concurrency can speed up programs
  - ➔ but limited by bottlenecks (e.g. disk, sequential sections)
- Key notion of *process* (= thread)
- java.lang.Thread
- Concurrent programming can easily go wrong!