Lecture 03—Gustafson's Law, Concurrency vs Parallelism

ECE 459: Programming for Performance

Patrick Lam

University of Waterloo

January 9, 2015

Multiprocessing in Three Acts

Act 1: **Attack of the Clones**
(symmetric multiprocessing systems)

Act 2: **The Empire Strikes Back**
(Amdahl’s Law)

Act 3: **A New Hope**
(Gustafson’s Law)

Amdahl's Law Generalization

The program may have many parts, each of which we can tune to a different degree.

Let's generalize Amdahl's Law.

$f_1, f_2, \ldots, f_n$: fraction of time in part n
$S_{f_1}, S_{f_n}, \ldots, S_{f_n}$: speedup for part n

$$\textbf{\textit{speedup}} = \frac{1}{\frac{f_1}{S_{f_1}} + \frac{f_2}{S_{f_2}} + \ldots + \frac{f_n}{S_{f_n}}}$$

Application (1)

Consider a program with 4 parts in the following scenario:

		Speedup	
Part	Fraction of Runtime	Option 1	Option 2
1	0.55	1	2
2	0.25	5	1
3	0.15	3	1
4	0.05	10	1

We can implement either Option 1 or Option 2.
Which option is better?

Application (2)

“Plug and chug” the numbers:

Option 1

$$\mathit{speedup} = \frac{1}{0.55 + \frac{0.25}{5} + \frac{0.15}{3} + \frac{0.05}{5}} = 1.53$$

Option 2

$$\mathit{speedup} = \frac{1}{\frac{0.55}{2} + 0.45} = 1.38$$

Empirically estimating parallel speedup P

Useful to know, don't have to commit to memory:

$$P_{\text{estimated}} = \frac{\frac{1}{speedup} - 1}{\frac{1}{N} - 1}$$

- Quick way to guess the fraction of parallel code
- Use $P_{\text{estimated}}$ to predict speedup for a different number of processors

Summary of Amdahl’s Law

Important to focus on the part of the program with most impact.

Amdahl’s Law:

- estimates perfect performance gains from parallelization (under assumptions); but,
- only applies to solving a **fixed problem size** in the shortest possible period of time

Gustafson's Law: Formulation

n:	problem size
$S(n)$:	fraction of serial runtime for a parallel execution
$P(n)$:	fraction of parallel runtime for a parallel execution

$$\begin{aligned} T_p &= S(n) + P(n) = 1 \\ T_s &= S(n) + N \cdot P(n) \end{aligned}$$

$$speedup = \frac{T_s}{T_p}$$

Gustafson's Law

$$speedup = S(n) + N \cdot P(n)$$

Assuming the fraction of runtime in serial part decreases as *n* increases, the speedup approaches *N*.

- Yes! Large problems can be efficiently parallelized. (Ask Google.)

Driving Metaphor

Amdahl’s Law
Suppose you’re travelling between 2 cities 90 km apart. If you travel for an hour at a constant speed less than 90 km/h, your average will never equal 90 km/h, even if you energize after that hour.

Gustafson’s Law
Suppose you’ve been travelling at a constant speed less than 90 km/h. Given enough distance, you can bring your average up to 90 km/h.

Part II

Parallelism versus Concurrency

Parallelism versus Concurrency

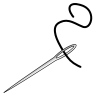
Parallelism
Two or more tasks are **parallel**
if they are running at the same time.
Main goal: run tasks as fast as possible.
Main concern: **dependencies**.

Concurrency
Two or more tasks are **concurrent**
if the ordering of the two tasks is not predetermined.
Main concern: **synchronization**.

Part III

Threads

Threads



- What are they?
- How do operating systems implement them?
- How can we leverage them?

Processes versus Threads

Process
An instance of a computer program that contains program code and its:

- Own address space / virtual memory;
- Own stack / registers;
- Own resources (file handles, etc.).

Thread
“Lightweight processes”.
In most cases, a thread is contained within a process.

- Same address space as parent process
    - Shares access to code and variables with parent.
- Own stack / registers
- Own thread-specific data

Software and Hardware Threads

Software Thread:
What you program with (e.g. with `pthread_create()` or `std::thread()`).

Corresponds to a stream of instructions executed by the processor.

On a single-core, single-processor machine, someone has to multiplex the CPU to execute multiple threads concurrently; only one thread runs at a time.

Hardware Thread:
Corresponds to virtual (or real) CPUs in a system. Also known as strands.

Operating system must multiplex software threads onto hardware threads, but can execute more than one software thread at once.

Thread Model—1:1 (Kernel-level Threading)

Simplest possible threading implementation.

The kernel schedules threads on different processors;

- NB: Kernel involvement required to take advantage of a multicore system.

Context switching involves system call overhead.

Used by Win32, POSIX threads for Windows and Linux.

Allows concurrency and parallelism.

Thread Model—N:1 (User-level Threading)

All application threads map to a single kernel thread.

Quick context switches, no need for system call.

Cannot use multiple processors, only for concurrency.

- Why would you use user threads?

Used by GNU Portable Threads.

Thread Model—M:N (Hybrid Threading)

Map *M* application threads to *N* kernel threads.

A compromise between the previous two models.

Allows quick context switches and the use of multiple processors.

Requires increased complexity:

- Both library and kernel must schedule.
- Schedulers may not coordinate well together.
- Increases likelihood of priority inversion (recall from Operating Systems).

Used by Windows 7 threads.

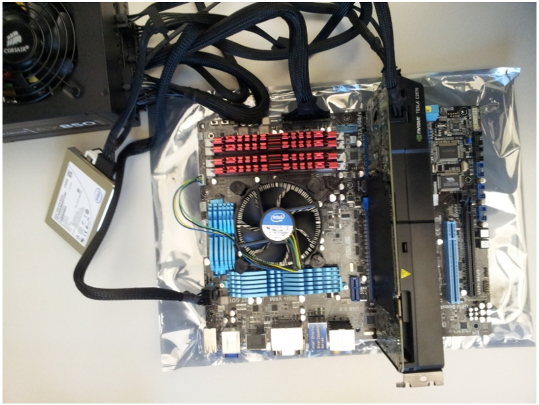
Live Coding Demo: Deducing the Thread Model

`time`, `top`, and `perf stat` all let you figure this out.

Part IV

Types of Multiprocessing

Example System—Physical View



- Only one physical CPU

Example System—System View

```
jon@ece459-1 ~ % egrep 'processor|model name' /proc/cpuinfo
processor : 0
model name: Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz
processor : 1
model name: Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz
processor : 2
model name: Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz
processor : 3
model name: Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz
processor : 4
model name: Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz
processor : 5
model name: Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz
processor : 6
model name: Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz
processor : 7
model name: Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz
```

- Many processors

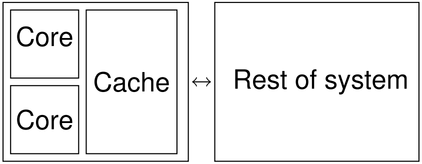
SMP (Symmetric Multiprocessing)

Identical processors or cores, which:

- are interconnected, usually using buses; and
- share main memory.

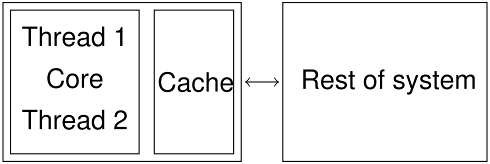
- SMP is most common type of multiprocessing system.

Example of an SMP System



- Each core can execute a different thread
- Shared memory quickly becomes the bottleneck

Executing 2 Threads on a Single Core



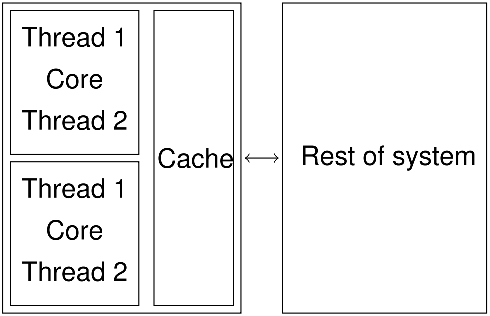
On a single core, must context switch between threads:

- every *N* cycles; or
- wait until cache miss, or another long event

Resources may be unused during execution.

Why not take advantage of this?

Executing M Threads on a N Cores



Here’s a Chip Multithreading example.

UltraSPARC T2 has 8 cores, each of which supports 8 threads. All of the cores share a common level 2 cache.

SMT (Simultaneous Multithreading)

Use idle CPU resources (may be calculating or waiting for memory) to execute another task.

Cannot improve performance if shared resources are the bottleneck.

Issue instructions for each thread per cycle.

To the OS, it looks a lot like SMP, but gives only up to 30% performance improvement.

Intel implementation: Hyper-Threading.

Example: Non-SMP system

PlayStation 3 contains a Cell processor:

- PowerPC main core (Power Processing Element, or “PPE”)
- 7 Synergistic Processing Elements (“SPE”s): small vector computers.

NUMA (Non-Uniform Memory Access)

In SMP, all CPUs have uniform (the same) access time for resources.

For NUMA, CPUs can access different resources faster (resources: not just memory).

Schedule tasks on CPUs which access resources faster.

Since memory is commonly the bottleneck, each CPU has its own memory bank.

Processor Affinity

Each task (process/thread) can be associated with a set of processors.

Useful to take advantage of existing caches (either from the last time the task ran or task uses the same data).

Hyper-Threading is an example of complete affinity for both threads on the same core.

Often better to use a different processor if current set busy.

Part V

Processes vs Threads

Background

Recall the difference between **processes** and **threads**:

- Threads are basically light-weight processes which piggy-back on processes' address space.

Traditionally (pre-Linux 2.6) you had to use `fork` (for processes) and `clone` (for threads).

History

`clone` is not POSIX compliant.
Developers mostly used `fork` in the past, which creates a new process.

- Drawbacks?
- Benefits?

Benefit: `fork` is Safer and More Secure Than Threads

1. Each process has its own virtual address space:
   - Memory pages are not copied, they are copy-on-write—
   - Therefore, uses less memory than you would expect.
2. Buffer overruns or other security holes do not expose other processes.
3. If a process crashes, the others can continue.

Example: In the Chrome browser, each tab is a separate process.

Drawback of Processes: Threads are Easier and Faster

- Interprocess communication (IPC) is more complicated and slower than interthread communication.
  - Need to use operating system utilities (pipes, semaphores, shared memory, etc) instead of thread library (or just memory reads and writes).
- Processes have much higher startup, shutdown and synchronization cost.
- And, **Pthreads**/**C++11 threads** fix issues with `clone` and provide a uniform interface for most systems **(on Assignment 1)**.

Appropriate Time to Use Processes

If your application is like this:

- Mostly independent tasks, with little or no communication.
- Task startup and shutdown costs are negligible compared to overall runtime.
- Want to be safer against bugs and security holes.

Then processes are the way to go.

For performance reasons, along with ease and consistency, we'll use **Pthreads**.