Lecture 03 - Parallel Limitations and Concepts

ECE 459: Programming for Performance

Jon Eyolfson

University of Waterloo

January 9, 2012

Limitations

- Our main focus is parallelization

- Most programs have a sequential part and a parallel part

- Amdahl's Law answers "what are the limits to parallelization?"

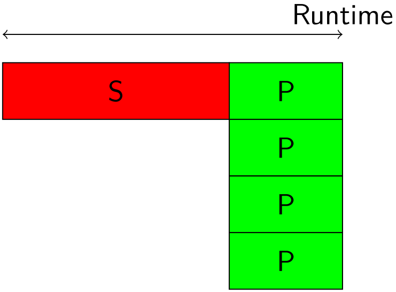
Formulation (1)

Let S be the fraction of serial runtime for a serial execution
Let P be the fraction of parallel runtime for a serial execution
Therefore, $S + P = 1$

If we have 4 processors, what do we want to happen to the following runtime?

Formulation (2)

We want to split up the parallel part over 4 processors



Formulation (2)

Let T_s be the time for the program to run in serial
Let N be the number of processors/parallel executions
Let T_p be the time for the program to run in parallel

- Under perfect conditions you will get N speedup for P

$$T_p = T_s \cdot (S + \frac{P}{N})$$

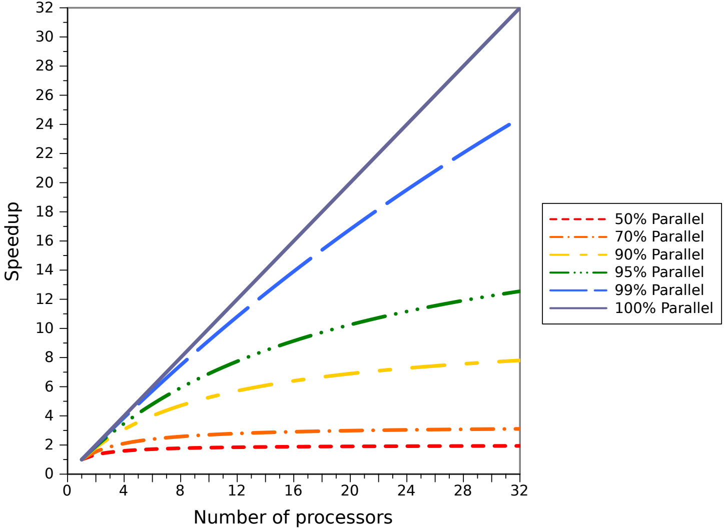
Formulation (3)

How much faster can we make the program?

$$speedup = \frac{T_s}{T_p}$$
$$speedup = \frac{T_s}{T_S \cdot (S + \frac{P}{N})}$$
$$speedup = \frac{1}{S + \frac{P}{N}}$$

We are assuming there is no overhead for parallelizing, or the costs are near zero

Scaling with Fraction of Parallel Code



Amdahl's Law

Replace S with $(1 - P)$

$$speedup = \frac{1}{(1-P)+\frac{P}{N}}$$

$$maximum\ speedup = \frac{1}{(1-P)}, \text{ since } \frac{P}{N} \rightarrow 0$$

As you might imagine, the asymptotes in the previous graph are bounded by the maximum speedup

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

Let $f_1, f_2, \ldots, f_n$ be the fraction of time in part n
Let $S_{f_1}, S_{f_n}, \ldots, S_{f_n}$ be the fraction of time in part n

$$speedup = \frac{1}{\frac{f_1}{S_{f_1}} + \frac{f_2}{S_{f_2}} + ... + \frac{f_n}{S_{f_n}}}$$

Application (1)

Consider a program with 4 distinct 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

Which option is better?

Application (2)

"Plug and chug" the numbers

Option 1

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

Option 2

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

Esimating P

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

$$P_{esimated} = \frac{\frac{1}{speedup} - 1}{\frac{1}{N} - 1}$$

- Quick way to guess the fraction of parallel code
- Use value of P to predict speedup for a different number of processors

Other Examples

We run a program in serial and find it spends 12.5% of it's execution on serial code and 87.5% on parallel code. How many processors do we need in order to get within 10% of the perfect parallel runtime?

Summary

- Important to focus on the part of the program which has the most impact
- Provides an estimation of perfect performance gains from parallelization
- Only applies to solving a fixed problem size in the shortest possible period of time

Formulation

Let n be a measure of the problem size
Let $S(n)$ be the fraction of serial runtime for a parallel execution
Let $P(n)$ be the fraction of parallel runtime for a parallel execution

$$T_p = S(n) + P(n) = 1$$
$$T_s = S(n) + N \cdot P(n)$$

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

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

- Shows that large problems can be efficiently parallelized

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 at your destination.

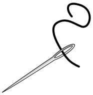
Gustafson's Law
Suppose you've been travelling at a constant speed less than 90 km/h. Given enough distance, you can bring your average up to 90 km/h.

Important Definitions

Parallelism
Two or more tasks running at the same time. Main goal is to run tasks as fast as possible. Main concern is dependencies.

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

Threads



- Important to understand what they are
- How they are implemented and used
- Ways we can take advantage of them

Comparison to Processes

Process

An instance of a computer program that contains program code and current activity.

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

Thread

In most cases, a thread is contained within a process.

- Same address space as parent process
  - Access to same code and variables
- Own stack / registers
- Own thread-specific data

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

- Simplest possible threading implementation
- Only the kernel can schedule threads on different processors
  - Required to take advantage of a multiprocessor system
- Context switching requires a system call
- Used by Win32, POSIX threads for Windows and Linux
- Allows concurrency and parallelism

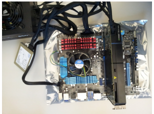
Thread Model - N:1 (User-level Threading)

- 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 them?
- Used by GNU Portable Threads

Thread Model - M:N (Hybrid Threading)

- Maps M application threads to N kernel threads
- Compromise between the previous two models
- Quick context switches and can use multiple processors
- Increased complexity, library provides scheduling
  - May not coordinate well with kernel
  - Increases likelihood of priority inversion (which we'll see later)
- Used by modern Windows threads

Example System - Physical View



- Only one physical chip

Example System - System View

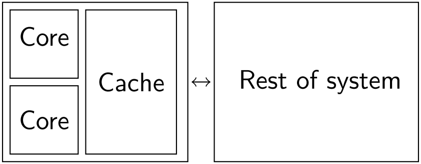
```
jon@ece459-1 ~ % egrep 'processor|model name' /proc/cpuinfo
processor : 0
model name: Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz
processor : 1
model name: Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz
processor : 2
model name: Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz
processor : 3
model name: Intel(R) Core(TM)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

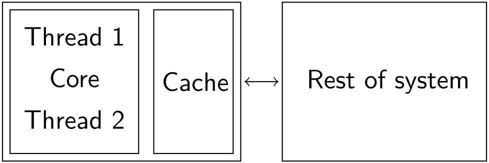
- Identical processors or cores
- Interconnected using buses or another type of communication
- Share main memory
- Most common type of multiprocessing system

Example of an SMP System



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

Executing of 2 Threads on a Single Core



- On a single core, must context switch between threads
  - Every number of cycles
  - Wait until cache miss, or another long event
- Resources may be unused during execution
- Why not take advantage of this?

SMT (Simultaneous Multithreading)

- Use any idle resources of a CPU (may be doing calculations/waiting for memory) for another task
- Cannot improve if shared resources are the bottleneck
- Issue instructions for each thread per cycle
- To the OS, it looks a lot like SMP, but only up to 30% performance improvement
- Intel implementation: Hyper-Threading

NUMA (Non-Uniform Memory Access)

- In SMP, all CPUs have the same access time for resources
- In this case, CPUs can access different resources faster (not just limited to memory)
- Schedule tasks to CPUs which can access resources faster
- Since memory is commonly the bottleneck, each CPU has it's 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
- May be better to use a different processor if current set is busy