Generalizing Amdahl's Law. We made a simplification, which was that programs only have one parallel part and one serial part. Of course, this is not true. 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

Then,

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

Example. 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

(Note: these speedups don't have to be speedups from parallelization.)

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

"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.**

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

Empirically estimating parallel speedup P. Assuming that you know things that are actually really hard to know, here's a formula for estimating speedup. You don't have to commit it to memory:

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

It's just an estimation, but you can use it to guess the fraction of parallel code, given N and the speedup. You can then use $P_{\text{estimated}}$ to predict speedup for a different number of processors.

A more optimistic point of view

In 1988, John Gustafson pointed out[1] that Amdahl's Law only applies to fixed-size problems, but that the point of computers is to deal with bigger and bigger problems.

In particular, you might vary the input size, or the grid resolution, number of timesteps, etc. When running the software, then, you might need to hold the running time constant, not the problem size: you're willing to wait, say, 10 hours for your task to finish, but not 500 hours. So you can change the question to: how big a problem can you run in 10 hours?

According to Gustafson, scaling up the problem tends to increase the amount of work in the parallel part of the code, while leaving the serial part alone. As long as the algorithm is linear, it is possible to handle linearly larger problems with a linearly larger number of processors.

Of course, Gustafson's Law works when there is some "problem-size" knob you can crank up. As a practical example, observe Google, which deals with huge datasets.

Concurrency and Parallelism

Concurrency and parallelism both give up the total ordering between instructions in a sequential program, for different purposes.

Concurrency. We'll refer to the use of threads for structuring programs as concurrency. Here, we're not aiming for increased performance. Instead, we're trying to write the program in a natural way. Concurrency makes sense as a model for distributed systems, or systems where multiple components interact, with no ordering between these components, like graphical user interfaces.

Parallelism. We're studying parallelism in this class, where we try to do multiple things at the same time in an attempt to increase throughput. Concurrent programs may be easier to parallelize.

---

[1] http://www.scl.ameslab.gov/Publications/Gus/AmdahlsLaw/Amdahls.html

Processor Design Issues

Recall that we listened to Cliff Click describe characteristics of modern processors in Lecture 2. In this lecture we'll continue our quick review of computer architecture and how it relates to programming for performance. Here's another reference about chip multi-threading; we are going to study some of the techniques in the "Writing Scalable Low-Level Code" section.

<div align="center">

`http://queue.acm.org/detail.cfm?id=1095419`

</div>

Processes and Threads.   Let's review the difference between a process and a thread. A *process* is an instance of a computer program that contains program code and its own address space, stack, registers, and resources (file handles, etc). A *thread* usually belongs to a process. The most important point is that it shares an address space with its parent process, hence variables and code as well as resources. Threads have their own stack, registers, and thread-specific data.

Threads and CPUs.   In your operating systems class, you've seen implementations of threads ("lightweight processes"). We'll call these threads *software threads*, and we'll program with them throughout the class. Each software thread corresponds to a stream of instructions that the processor executes. On a old-school single-core, single-processor machine, the operating system multiplexes the CPU resources to execute multiple threads concurrently; however, only one thread runs at a time on the single CPU.

On the other hand, a modern chip contains a number of *hardware threads*, which correspond to the virtual CPUs. These are sometimes known as *strands*. The operating system still needs to multiplex the software threads onto the hardware threads, but now has more than one hardware thread to schedule work onto.

What's the term for swapping out the active thread on a CPU?

Implementing (or Simulating) Hardware Threads.   There are a number of ways to implement multiple software threads; for instance, the simplest possible implementation, **kernel-level threading** (or 1:1 model) dedicates one core to each thread. The kernel schedules threads on different processors. (Note that kernel involvement will always be required to take advantage of a multicore system). This model is used by Win32, as well as POSIX threads for Windows and Linux. The 1:1 model allows concurrency and parallelism.

Alternately, we could make one core execute multiple threads, in the **user-level threading**, or N:1, model. The single core would keep multiple contexts and could 1) switch every 100 cycles; 2) switch every cycle; 3) fetch one instruction from each thread each cycle; or 4) switch every time the current thread hits a long-latency event (cache miss, etc.) This model allows for quick context switches, but does not leverage multiple processors. (Why would you use these?) The N:1 model is used by GNU Portable Threads.

Finally, it's possible to both use multiple cores and put multiple threads onto one core, in a **hybrid threading**, or M:N, model. Here, we map M application threads to N kernel threads. This is a compromise between the previous two models, which both allows quick context switches and the use of multiple processors. However, it requires increased complexity; the library provides scheduling services, which may not coordinate well with kernel, and increases likelihood of priority inversion (which you've seen in Operating Systems). This method is used by modern Windows threads.

Live Coding Example. We can write a program to determine whether we are in a user-level threading model vs. the other two models. Just start a child thread and make both threads compute. If CPU utilization exceeds 100%, we are not user-level threading. The example I saw in class computes 2^{28} random numbers to spin the CPU.

(Is it possible to differentiate hybrid and kernel threading by writing a simple program?)