

## Limits to parallelization

I mentioned briefly in Lecture 1 that programs often have a sequential part and a parallel part. We'll quantify this observation today and discuss its consequences.

**Amdahl's Law.** One classic model of parallel execution is Amdahl's Law. In 1967, Gene Amdahl argued that improvements in processor design for single processors would be more effective than designing multi-processor systems. Here's the argument. Let's say that you are trying to run a task which has a serial part, taking time  $S$ , and a parallelizable part, taking time  $P$ , then the total amount of time  $T_s$  on a single-processor system is:

$$T_s = S + P.$$

Now, moving to a parallel system with  $N$  processors, the parallel time  $T_p$  is instead:

$$T_p = S + \frac{P}{N}.$$

**As  $N$  increases,  $T_p$  is dominated by  $S$ , limiting the potential speedup.**

We can restate this law in terms of speedup, which is generally the original time  $T_s$  divided by the sped-up time  $T_p$ :

$$S_p = \frac{T_s}{T_p}.$$

If we let  $f$  be the parallelizable fraction of the computation, i.e. set  $f$  to  $P/T_s$ , and we let  $S_f$  be the speedup we can achieve on  $f$ , then we get<sup>1</sup>:

$$S_p(f, S_f) = \frac{1}{(1 - f) + \frac{f}{S_f}}.$$

**Plugging in numbers.** If  $f = 1$ , then we can indeed get good scaling, since  $S_p = S_f$  in that case; running on an  $N$ -processor machine will give you a speedup of  $N$ . Unfortunately, usually  $f < 1$ . Let's see what happens.

| $f$  | $S_p(f, 18)$ |
|------|--------------|
| 1    | 18           |
| 0.99 | $\sim 15$    |
| 0.95 | $\sim 10$    |
| 0.5  | $\sim 2$     |

<sup>1</sup><http://www.cs.wisc.edu/multifacet/amdahl/>

To get close to a  $2\times$  speedup for  $f = 0.5$ , you'd need to use around 18 cores<sup>2</sup>.

Amdahl's Law tells you how many cores you can hope to leverage in your code, if you can estimate  $f$ . I'll leave this as an exercise to the reader: fix some  $f$  and set a tolerance, i.e. you don't care about a speedup less than  $x$ . Use the equations to figure out how many cores give that speedup.

The graph looks like this:

To get reasonable parallelization with a tolerance of 1.1,  $f$  needs to be at least 0.8.

**Consequences of Amdahl's Law.** For over 30 years, most performance gains did indeed come from increasing single-processor performance. The main reason that we're here today is that, as we saw in the video in Lecture 2, single-processor performance gains have hit the wall.

By the way, note that we didn't talk about the cost of synchronization between threads here. That can drag the performance down even more.

## A more optimistic point of view

In 1988, John Gustafson pointed out<sup>3</sup> 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.

## Modern Processors

For most of this lecture, I showed the video by Cliff Click on modern hardware:

<http://www.infoq.com/presentations/click-crash-course-modern-hardware>

---

<sup>2</sup>v2 corrected typo:  $N = 18$  in Plugging in Numbers

<sup>3</sup><http://www.scl.ameslab.gov/Publications/Gus/AmdahlsLaw/Amdahls.html>