

6.2 The difficulty of creating parallel processing programs

The difficulty with parallelism is not the hardware; it is that too few important application programs have been rewritten to complete tasks sooner on multiprocessors. It is difficult to write software that uses multiple processors to complete one task faster, and the problem gets worse as the number of processors increases.

Why has this been so? Why have parallel processing programs been so much harder to develop than sequential programs?

The first reason is that you *must* get better performance or better energy efficiency from a parallel processing program on a multiprocessor; otherwise, you would just use a sequential program on a uniprocessor, as sequential programming is simpler. In fact, uniprocessor design techniques, such as superscalar and out-of-order execution, take advantage of instruction-level parallelism (see COD Chapter 4 (The Processor)), normally without the involvement of the programmer. Such innovations reduced the demand for rewriting programs for multiprocessors, since programmers could do nothing and yet their sequential programs would run faster on new computers.

Why is it difficult to write parallel processing programs that are fast, especially as the number of processors increases? In COD Chapter 1 (Computer Abstractions and Technology), we used the analogy of eight reporters trying to write a single story in hopes of doing the work eight times faster. To succeed, the task must be broken into eight equal-sized pieces, because otherwise some reporters would be idle while waiting for the ones with larger pieces to finish. Another speed-up obstacle could be that the reporters would spend too much time communicating with each other instead of writing their pieces of the story. For both this analogy and parallel programming, the challenges include scheduling, partitioning the work into parallel pieces, balancing the load evenly between the workers, time to synchronize, and overhead for communication between the parties. The challenge is stiffer with the more reporters for a newspaper story and with the more processors for parallel programming.

Our discussion in COD Chapter 1 (Computer Abstractions and Technology) reveals another obstacle, namely Amdahl's Law. It reminds us that even small parts of a program must be parallelized if the program is to make good use of many cores.

Example 6.2.1: Speed-up challenge.

Suppose you want to achieve a speed-up of 90 times faster with 100 processors. What percentage of sequential?

Answer

Amdahl's Law (COD Chapter 1 (Computer Abstractions and Technology)) says

$$\text{Execution time after improvement} = \frac{\text{Execution time affected by improvement}}{\text{Amount of improvement}} +$$

We can reformulate Amdahl's Law in terms of speed-up versus the initial execution time:

$$\text{Speed-up} = \frac{\text{Execution time before}}{(\text{Execution time before} - \text{Execution time affected}) + \frac{\text{Execution time affected}}{\text{Amount of improvement}}}$$

This formula is usually rewritten assuming that the execution time before is 1 for some unit of time, and by improvement is considered the fraction of the original execution time:

$$\text{Speed-up} = \frac{1}{(1 - \text{Fraction time affected}) + \frac{\text{Fraction time affected}}{\text{Amount of improvement}}}$$

Substituting 90 for speed-up and 100 for the amount of improvement into the formula above:

$$90 = \frac{1}{(1 - \text{Fraction time affected}) + \frac{\text{Fraction time affected}}{100}}$$

Then simplifying the formula and solving for fraction time affected:

$$90 \times (1 - 0.99 \times \text{Fraction time affected}) = 1$$

$$90 - (90 \times 0.99 \times \text{Fraction time affected}) = 1$$

$$90 - 1 = 90 \times 0.99 \times \text{Fraction time affected}$$

$$\text{Fraction time affected} = 89/89.1 = 0.999$$

Thus, to achieve a speed-up of 90 from 100 processors, the sequential percentage can only be 0.1%.

However, there are applications with plenty of parallelism, as we shall see next.

Example 6.2.2: Speed-up challenge: Bigger problem.

Suppose you want to perform two sums: one is a sum of 10 scalar variables, and one is a matrix sum of 10 by 10 arrays, with dimensions 10 by 10. For now let's assume only the matrix sum is parallelizable; we'll see later about the scalar sum. What speed-up do you get with 10 versus 40 processors? Next, calculate the speed-ups assuming 20.

Answer

If we assume performance is a function of the time for an addition, t , then there are 10 additions that can be done in parallel by 10 processors and 100 additions that do. If the time for a single processor is $110t$, the execution time for

$$\text{Execution time after improvement} = \frac{\text{Execution time affected by improvement}}{\text{Amount of improvement}} +$$

$$\text{Execution time after improvement} = \frac{100t}{10} + 10t = 20t$$

so the speed-up with 10 processors is $110t/20t = 5.5$. The execution time for 40 processors is

$$\text{Execution time after improvement} = \frac{100t}{40} + 10t = 12.5t$$

so the speed-up with 40 processors is $110t/12.5t = 8.8$. Thus, for this problem size, we get about 55% with 10 processors, but only 22% with 40.

Look what happens when we increase the matrix. The sequential program now takes $10t + 400t = 410t$ processors is

$$\text{Execution time after improvement} = \frac{400t}{10} + 10t = 50t$$

so the speed-up with 10 processors is $410t/50t = 8.2$. The execution time for 40 processors is

$$\text{Execution time after improvement} = \frac{400t}{40} + 10t = 20t$$

so the speed-up with 40 processors is $410t/20t = 20.5$. Thus, for this larger problem size, we get 82% with 10 processors and 51% with 40.

These examples show that getting good speed-up on a multiprocessor while keeping the problem size fixed is harder than getting good speed-up by increasing the size of the problem. This insight allows us to introduce two terms that describe ways to scale up.

Strong scaling means measuring speed-up while keeping the problem size fixed. *Weak scaling* means that the problem size grows proportionally to the increase in the number of processors. Let's assume that the size of the problem, M , is the working set in main memory, and we have P processors. Then the memory per processor for strong scaling is approximately M/P , and for weak scaling, it is about M .

Strong scaling: Speed-up achieved on a multiprocessor without increasing the size of the problem.

Weak scaling: Speed-up achieved on a multiprocessor while increasing the size of the problem proportionally to the increase in the number of processors.

Note that the **memory hierarchy** can interfere with the conventional wisdom about weak scaling being easier than strong scaling. For example, if the weakly scaled dataset no longer fits in the last level cache of a multicore microprocessor, the resulting performance could be much worse than by using strong scaling.

Depending on the application, you can argue for either scaling approach. For example, the TPC-C debit-credit database benchmark requires that you scale up the number of customer accounts in proportion to the higher transactions per minute. The argument is that it's nonsensical to think that a given customer base is suddenly going to start using ATMs 100 times a day just because the bank gets a faster computer. Instead, if you're going to demonstrate a system that can perform 100 times the numbers of transactions per minute, you should run the experiment with 100 times as many customers. Bigger problems often need more data, which is an argument for weak scaling.

This final example shows the importance of load balancing.



Example 6.2.3: Speed-up challenge: Balancing load.

To achieve the speed-up of 20.5 on the previous larger problem with 40 processors, we assumed the load was perfectly balanced. That is, each of the 40 processors had 2.5% of the work to do. Instead, show the impact on speed-up if one processor's load is higher than all the rest. Calculate at twice the load (5%) and five times the load (12.5%) for that hardest working processor. How well utilized are the rest of the processors?

Answer

If one processor has 5% of the parallel load, then it must do $5\% \times 400$ or 20 additions, and the other 39 will share the remaining 380. Since they are operating simultaneously, we can just calculate the execution time as a maximum

$$\text{Execution time after improvement} = \text{Max}\left(\frac{380t}{39}, \frac{20t}{1}\right) + 10t = 30t$$

The speed-up drops from 20.5 to $410t/30t = 14$. The remaining 39 processors are utilized less than half the time: while waiting 20t for the hardest working processor to finish, they only compute for $380t/39 = 9.7t$.

If one processor has 12.5% of the load, it must perform 50 additions. The formula is:

$$\text{Execution time after improvement} = \text{Max}\left(\frac{350t}{39}, \frac{50t}{1}\right) + 10t = 60t$$

The speed-up drops even further to $410t/60t = 7$. The rest of the processors are utilized less than 20% of the time ($9t/50t$). This example demonstrates the importance of balancing load, for just a single processor with twice the load of the others cuts speed-up by a third, and five times the load on just one processor reduces speed-up by almost a factor of three.

Now that we better understand the goals and challenges of parallel processing, we give an overview of the rest of the chapter. The next section (COD Section 6.3 (SISD, MIMD, SIMD, SPMD, and vector)) describes a much older classification scheme than in COD Figure 6.1 (Hardware/software categorization and examples ...). In addition, it describes two styles of instruction set architectures that support running of sequential applications on parallel hardware, namely *SIMD* and *vector*. COD Section 6.4 (Hardware multithreading) then describes *multithreading*, a term often confused with multiprocessing, in part because it relies upon similar concurrency in programs. COD Section 6.5 (Multicore and other shared memory multiprocessors) describes the first the two alternatives of a fundamental parallel hardware characteristic, which is whether or not all the processors in the systems rely upon a single physical address space. As mentioned above, the two popular versions of these alternatives are called *shared memory multiprocessors* (SMPs) and *clusters*, and this section covers the former. COD Section 6.6 (Introduction to graphics processing units) describes a relatively new style of computer from the graphics hardware community, called a *graphics-processing unit* (GPU) that also assumes a single physical address. (COD Appendix B (Graphics and Computing GPUs) describes GPUs in even more detail.) COD Section 6.7 (Clusters, warehouse scale computers, and other message-passing multiprocessors) describes clusters, a popular example of a computer with multiple physical address spaces. COD Section 6.8 (Introduction to multiprocessor network topologies) shows typical topologies used to connect many processors together, either server nodes in a cluster or cores in a microprocessor. COD Section 6.9 (Communicating to the outside world: Cluster networking) describes the hardware and software for communicating between nodes in a cluster using Ethernet. It shows how to optimize its performance using custom software and hardware. We next discuss the difficulty of finding parallel benchmarks in COD Section 6.10 (Multiprocessor benchmarks and performance models). This section also includes a simple, yet insightful performance model that helps in the design of applications as well as architectures. We use this model as well as parallel benchmarks in COD Section 6.11 (Real stuff).

Benchmarking Intel Core i7 versus NVIDIA Tesla GPU) to compare a multicore computer to a GPU. COD Section 6.12 (Going faster: Multiple processors and matrix multiply) divulges the final and largest step in our journey of accelerating matrix multiply. For matrices that don't fit in the cache, parallel processing uses 16 cores to improve performance by a factor of 14. We close with fallacies and pitfalls and our conclusions for parallelism.

In the next section, we introduce acronyms that you probably have already seen to identify different types of parallel computers.

**PARTICIPATION
ACTIVITY**

6.2.1: Check yourself: Strong scaling.




1) Strong scaling is not bound by Amdahl's Law.



☐ True

☐ False

 [Provide feedback on this section](#)