Scaling Theory and Machine Abstractions

Martha A. Kim

October 10, 2012

This document describes the Tufte handout LaTeX document style. It also provides examples and comments on the style's use. Only a brief overview is presented here; for a complete reference, see the sample book.

Scaling Theory

Metrics: Execution Time and Speedup

Parallelization is no different than any other program optimization. Thus, when evaluating the quality of a particular parallelization, we will use many of the same metrics that we use in the serial domain, namely: execution time and speedup. Execution time is simply a measure of the time required to complete a particular task. Speedup compares the execution time of a program before and after optimization. An optimization's speedup is simply the ratio of the runtime before optimization to the runtime afterwards:

$$Speedup = \frac{Runtime_{initial}}{Runtime_{optimized}} \tag{1}$$

For example, an optimization that offers a speedup of two will cause the program to run twice as fast.

Strong and Weak Scaling

A parallel program's scalability quantifies how execution time (or speedup) relates to the number of processors executing the program. While you already have some intuition about what it means for a program's performance to scale, we will formalize it here.

Let $time(p, x)$ be the time required by p processors to solve a problem of size x. We can then define the parallel speedup as before. On a problem of size x with p processors, the speedup is:

$$speedup(p, x) = \frac{time(1, x)}{time(p, x)} \tag{2}$$

Measuring such speedups, where an increasing number of processors are applied to fixed-size problem, is called *strong scaling*. By contrast, *weak scaling* measures speedups assuming a *fixed problem size per processor*:

$$speedup(p, x \times p) = \frac{time(1, x)}{time(p, x \times p)} \tag{3}$$

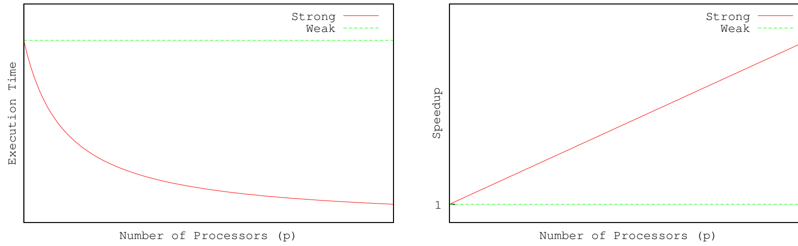


Figure 1: Ideal scaling patterns for strong and weak scaling.

Demonstrating strong scaling requires solving a fixed problem size faster and faster, while demonstrating weak scaling requires solving an increasing problem size within a fixed time budget. As their names imply, demonstrating strong scaling is typically more challenging than demonstrating weak scaling.

Amdahl's Law

Amdahl's Law, originally articulated by Gene Amdahl in 1967 [1], provides an upper bound on the speedup to be expected from a particular optimization. Assuming an optimization accelerates some fraction of the program's total runtime (f) by a factor of s_f (for speedup). Amdahl's Law states that the maximum overall speedup is:

$$Speedup = \frac{1}{(1-f) + \frac{f}{s_f}} \tag{4}$$

[1]

The key consequence of Amdahl's Law is that the overall speedup of an optimization is *limited by the non-optimizable portion of an application's execution time*.

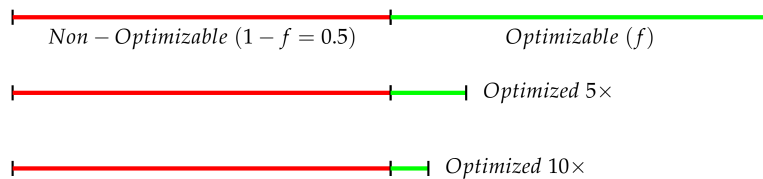


Figure 2: Amdahl's Law states that overall speedups are limited by the non-optimized proportion of an application's runtime. If half of an application's execution is not-optimized (shown in red), arbitrary optimization of the remainder (in green) results in speedups of at most 2.

In the context of parallelization, serial work quickly dashes hopes of demonstrating strong scaling. [2]

[2] For an exploration of Amdahl's implications for multicore architectures, see related reading "Amdahl's Law in the Multicore Era" .

Gustafson's Law

Whereas Amdahl's Law assumes a fixed problem size, Gustafson's Law, stated by John L. Gustafson in 1988 [3], quantifies speedup if the problem size is allowed to increase. We derive Gustafson's Law here.

[3]

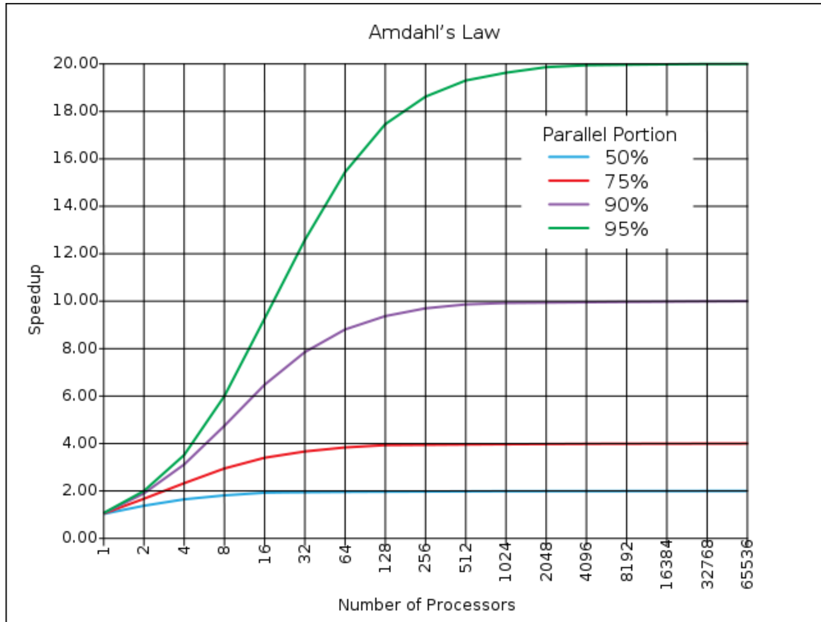
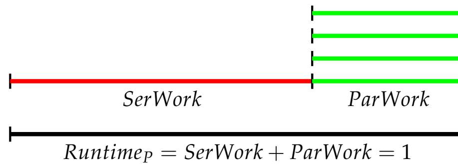
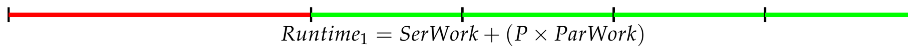


Figure 3: Amdahl's projected speedups for different proportions of serial work. Graph credit: Wikipedia

Let us first define $Runtime_P$ to be a program's runtime on a parallel computer with P processors with the available parallel work divided evenly amongst processors:



If instead we had only one processor, the runtime would be:



Speedup is, as before, the ratio of these two terms:

$$\begin{aligned} Speedup &= \frac{Runtime_1}{Runtime_P} && (5) \\ &= \frac{SerWork + P \times ParWork}{1} && (6) \\ &= SerWork + P \times (1 - SerWork) && (7) \end{aligned}$$

Note that using this formulation, speedup is *not* limited by the proportion of serial work ($SerWork$) in the program. What Gustafson's

Law observes is that for many problems, parallelism grows with the problem size, and the problem size grows with time.

Also note the parallels between Amdahl's and Gustafson's laws and strong and weak scaling. Whereas Amdahl's Law assumed a strict scaling setup (i.e., constant problem size in decreasing time), Gustafson's Law assumes a weak scaling setup (i.e., increasing problem size in fixed time).

Machine Models

In order to write high-performance parallel code, a programmer must understand something of the structure of the parallel computer that will execute it. For this reason, it is essential that some aspects of the machine be exposed to the programmer. However, it is equally essential to balance these specifics against the need for a simple abstraction and code portability.

How much of the details of the machine are exposed to the programmer (and proportionally how much machine-specific optimization can occur) will depend on the purpose of the program. If an application is being written for a specific computer or supercomputer or gaming platform, for example, machine-specific optimization can yield critical performance benefits. However, if a code is intended to be portable, there is no use in over-optimizing it for one machine.

Machine abstractions

The RAM model The Random Access Machine (RAM) or von Neumann model is an abstraction of a sequential computer. This computer consists of an instruction execution unit and an unbounded memory containing both program data and program instructions. In the simplified world of the model, any location in memory can be read to or written from in unit time. (This is the reason for the "random access" name.) This model is so familiar that we hardly even think of its being a model. However, it allows programmers to reason about performance via a simple instruction count. In this model, the more instructions executed, the longer a program will take. In practice, of course, the execution time depends not only on intsruction count, but on the size of caches (dictating how frequently a program needs to go to memory), the size of physical memory (and how frequently page faults occur), what other programs are contending for these resources, and so on. However, to a first order, the RAM model's instruction count gives a pretty good approximation of the quality of an algorithm.

The PRAM model Given the value of the RAM model, it is only natural to try to develop an analogous model for use modeling a parallel computer. The PRAM (for parallel RAM) model is an attempt to do just that. The PRAM consists of an unspecified number of instruction execution units connected to a single, unbounded shared memory. This natural extension has been very useful for theoretically analyzing the limits of parallel algorithms, but it has not turned out as well on the practical front. For programmers, the PRAM model fails to accurately capture memory behavior. While the simplification of memory behavior was one of the key strengths of the RAM model, the PRAM model has gone too far in its memory simplification. PRAM fails to capture the fact that memory in a parallel architecture *does not* have uniform unit access time. It is significantly faster for an execution unit to read/write data that is nearby in memory than it is farther away, but the PRAM model represents *all* memory as being equidistant from the execution units. The more execution units present in the parallel system, the bigger this drawback becomes.

The CTA model The Candidate Type Architecture (CTA) corrects the drawbacks of PRAM by explicitly distinguishing the fast local memory accesses from the slower remote accesses. Like the PRAM, the CTA consists of an unspecified number of instruction execution units. However, unlike the PRAM, each of these units also has a piece of the system memory connected directly to it. Multiple of these execution/memory pairs are then connected via an interconnect. In the CTA, accesses to a data element in memory from the conjoined execution unit are fast (having unit time), while accesses from a remote execution unit are slower (requiring λ units of time).

As a result of this, Calvin Lin and Larry Snyder articulate the "locality rule" as follows: "Fast programs tend to maximize the number of local memory references and minimize the number of non-local memory references" [4].

[4]

While the CTA is slightly more complex than the PRAM, it still represents an aggressively simplified view of a machine's architecture. For example, note that the CTA says nothing about the topology of the interconnect. While in practice, the interconnect topology will determine the "nearness" of data to a given processor, the CTA's simplified, two-teir system of "here" and "not-here" (for local and remote data respectively) is sufficient to capture the lion's share of data locality. Another detail that the CTA does not specify is the memory referencing mechanism, be it shared memory, 1-sided communication or message passing.

Shared memory: A natural extension of sequential computers, shared memory represents a single, coherent memory image to multiple threads. This style of memory reference is thought to be easy to write, but difficult to debug for functionality and performance. These latter drawbacks come because shared memory makes it easy to introduce race conditions and to unknowingly create non-local references thereby hurting performance.

1-sided communication: One-sided communication is a relaxation of shared memory where all threads can access a shared address space, but there is no effort to keep the memory coherent. This change simplifies the hardware by removing the need for coherence support, but is harder on the programmer who now must contend with different accesses to the same variable yielding different results.

Message passing: Finally, message passing offers no illusion of a shared address space. Instead processors can access only their local data. Should a processor require non-local data, messages are used to explicitly send data from one processor's memory to another.