Performance Models for Parallel Applications

Prof. Naga Kandasamy
ECE Department, Drexel University

We discuss three simple models which are used to provide insights to programmers and computer architects on improving performance of parallel software and hardware.

Amdahl's Law

The important principle of computer system design is to make the common case fast; that is, in making a design tradeoff, favor the frequent case over the infrequent case. This principle also applies when determining how to spend resources—dollars and personnel—since the impact on making some occurrence faster is higher if that occurrence is frequent. So, we have to decide what the frequent case is and how much performance can be gained by making that case faster. *Amdahl's law* is used to quantify this principle.[1] The performance gain that can be obtained by improving some portion of a program can be calculated using this law, which states that the performance improvement to be gained from using some faster mode of execution is limited by the fraction of time the faster mode can be used. So, Amdahl's law defines the speedup that can be gained by using a particular feature:

$$\text{Speedup} = \frac{\text{Execution time without using the enhancement}}{\text{Execution time using the enhancement when possible}}$$

Say that the speedup (or enhancement) can be used for some fraction of the original program which we shall term $\text{Fraction}_{\text{enhanced}}$. The speedup achieved when this enhancement is applied is $\text{Speedup}_{\text{enhanced}}$. So, the improved execution time is given by

$$\text{Execution time}_{\text{new}} = \text{Execution time}_{\text{old}} \times \left(1 - \text{Fraction}_{\text{enhanced}} + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}\right)$$

The overall speedup is given by

$$\begin{aligned} \text{Speedup} &= \frac{\text{Execution time}_{\text{old}}}{\text{Execution time}_{\text{new}}} \qquad (1)\\ &= \frac{1}{1 - \text{Fraction}_{\text{enhanced}} + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}} \end{aligned}$$

Example. Assume that 25% of your program is floating-point operations and that the rest of the program is integer operations. A design team proposes to speed up the floating-point operations by

[1] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," *AFIPS spring joint computer conference*, 1967.

five times using a dedicated floating-point hardware unit. Another team proposes to speed up just the integer operations by two times. Given that you have the budget to spend on only one design, which design would you fund, and why?

Answer. Let us consider the first design proposal involving speeding up the floating-point operations and call it $\mathrm{design}_\mathrm{A}$. Since we can speed up 25% of the code using this approach by 5 times, we can apply Amdahl's law to obtain the overall speedup

$$\begin{aligned}\mathrm{Speedup}_\mathrm{A} &= \frac{1}{1 - .25 + \frac{.25}{5}} \\ &= 1.25\end{aligned}$$

Now, consider the second design proposal involving speeding up integer operations and call it $\mathrm{design}_\mathrm{B}$. Since we can speed up 75% of the code using this approach by 2 times, the overall speedup is given by

$$\begin{aligned}\mathrm{Speedup}_\mathrm{B} &= \frac{1}{1 - .75 + \frac{.75}{2}} \\ &= 1.6\end{aligned}$$

So, according to Amdahl's law, $\mathrm{design}_\mathrm{B}$ provides a bigger bang for the buck.

Amdahl's law can also be used to ascertain the viability of implementing large-scale parallelism. Unfortunately, under the assumptions made by Amdahl when formulating his law, it can be quite pessimistic regarding the viability of large-scale parallelism. If N is the number of processors, s is the amount of time spent by a serial processor on serial parts of a program, and p is the amount of time spent by a serial processor on parts of the program that can be done in parallel, then Amdahl's law says that speedup is given by

$$\begin{aligned}\mathrm{Speedup} &= \frac{s + p}{s + p/N} \\ &= \frac{1}{s + p/N}\end{aligned}$$

where we have set the total time to $s + p = 1$ for algebraic simplicity. According to the above equation, even when the fraction of serial work s in a given problem is small, the maximum speedup obtainable from even an infinite number of parallel processors is only $1/s$. For example, if we have $N = 1024$ processors and only 5% of the code is serial, the overall speedup achieved is quite small—only 19.6 times—relative to the number of available processors.

Gustafson's Law

The above expression contains the implicit assumption that p is independent of N, which Gustafson argues is virtually never the case.[2] In his view, one does not take a fixed-sized problem and run it on various numbers of processors. In practice, the problem size scales with the number of processors. When given a more powerful processor, the problem generally expands to make use

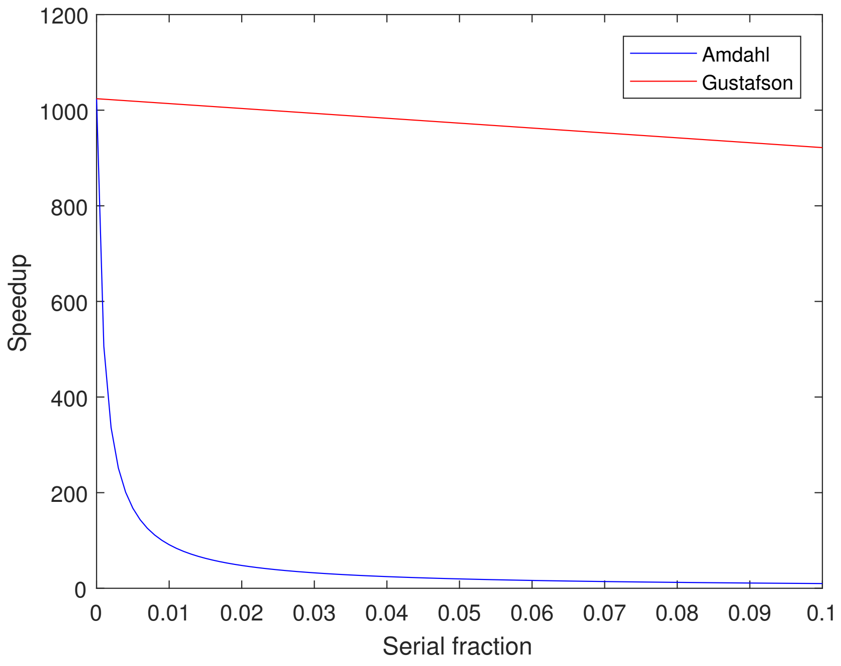
[2] J. L. Gustafson, "Reevaluating Amdahl's Law," *Communications of the ACM*, vol. 31, no. 5, pp. 532–3, May 1988.

of the increased facilities. Users have control over such things as grid resolution, number of time steps, difference operator complexity, and other parameters that are usually adjusted to allow the program to be run in some desired amount of time. Hence, it may be most realistic to assume run time, not problem size, is constant.

If we use s and p to represent serial and parallel time spent on the parallel system, then a serial processor would require time $s + p \times N$ to perform the task since we assume that the problem size is scaled with the number of processors. This reasoning gives an alternative to Amdahl's law: the so-called scaled speedup law. Since $s + p = 1$,

$$\begin{aligned} \text{Scaled\_speedup} &= (s + p \times N)/(s + p) \\ &= s + p \times N \\ &= N + (1 - N) \times s \end{aligned}$$

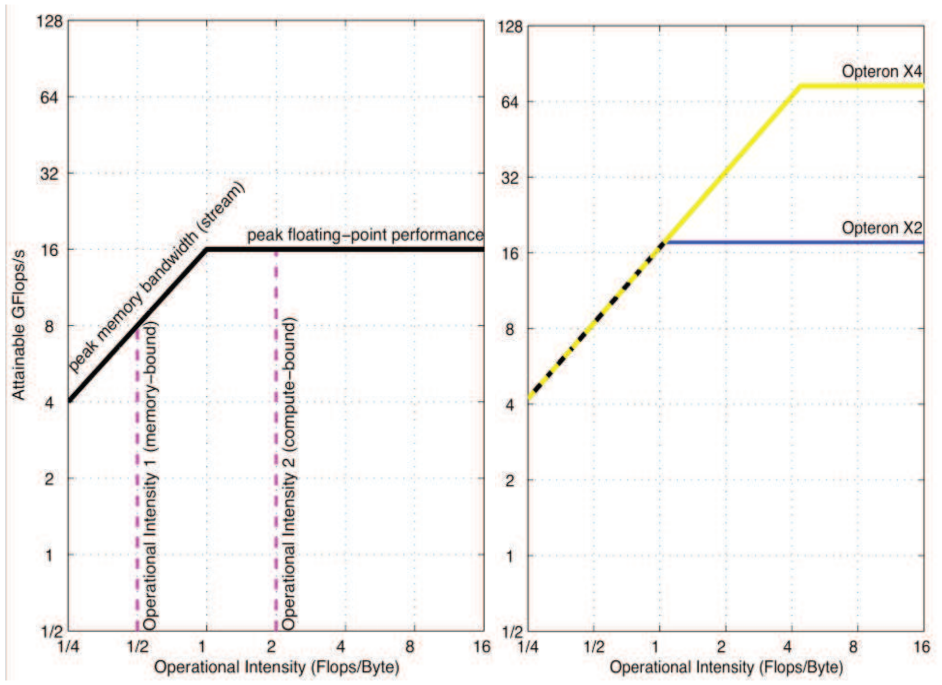
As shown in the figure below for $N = 1024$ processors, the scaled-speedup function is simply a line with a more moderate slope of $1 - N$. It is thus much easier to achieve efficient parallel performance than is implied by Amdahl's paradigm.



Roofline Model

Off-chip (or global) memory bandwidth is believed to be the constraining factor for performance for the foreseeable future. The *roofline* is a visual performance model that can be used to bound the performance of programs running on manycore processors. This model relates processor performance to off-chip memory traffic using a concept called *operational or arithmetic intensity*.

The above figures provide examples of how to use this model for programs written for AMD's



Opteron processor.[3] We can plot a horizontal line showing peak floating-point (FP) performance of the computer. Clearly, the actual FP performance of a kernel can be no higher than the horizontal line, since that is a hardware limit. Since X-axis is GFlops per byte and the Y-axis is GFlops per second, bytes per second—which equals (GFlops/second)/(GFlops/byte)—is just a line at a 45^o angle in this figure. Hence, we can plot a second line that gives the maximum FP performance that the memory system of that computer can support for a given arithmetic intensity. This formula drives the two performance limits in the graphs.

$$\begin{aligned}\text{Attainable GFlops/sec} = \min(&\text{Peak FP Performance},\\ &\text{Peak Memory Bandwidth} \times \text{Arithmetic Intensity})\end{aligned}$$

These two lines intersect at the point of peak computational performance and peak memory bandwidth. The horizontal and diagonal lines give this bound model its name. The Roofline sets an upper bound on performance of a kernel depending on its arithmetic intensity. If we think of arithmetic intensity as a column that hits the roof, either it hits the flat part of the roof, which means performance is *compute bound,* or it hits the slanted part of the roof, which means performance is ultimately *memory bound.* In the above figure, a kernel with operational intensity two is compute bound and a kernel with operational intensity one is memory bound, when executed on the Opteron X2 processor. The peak FP performance of the Opteron X2 is 16 GFlops, and the peak memory bandwidth that can be sustained is approximately 16 GBytes/sec.

Arithmetic intensity of a kernel is defined as the following ratio:

$$\frac{\text{Total FP operations}}{\text{Total data movement (in Bytes)}}.$$

[3]S. Williams, A. Waterman, and D. Patterson, "Roofline: An Insightful Visual Performance Model for Multicore Architectures," *Communications of the ACM,* Vol. 52, No. 4, pp. 65–76, April 2009

Example. Consider a program that performs the SAXPY computation.[4] SAXPY is a combination of scalar multiplication and vector addition: it takes as input two vectors of 32-bit floats `X` and `Y` with `N` elements each, and a scalar value `A`. It multiplies each element `X[i]` by `A` and adds the result to `Y[i]`. A simple C implementation looks like this.

```c
void saxpy(int n, float a, float *x, float *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
```

What is the arithmetic intensity of the above code snippet? Is the above code memory bound or compute bound as per the roofline model?

Answer. Examining the body of the loop, we see that $3 \times 4 = 12$ bytes of data — two loads and a store — are moved between memory and the CPU for two FP operations, which are add and multiply. The constant `a` is assumed to be in a register. So the arithmetic intensity is $2/12 = 0.167$, which if placed on the X-axis of the roofline graph would indicate a memory-bound operation, since $0.167\frac{\text{FLOP}}{\text{Bytes}} \times 16\frac{\text{GBytes}}{\text{sec}} = 2.67\frac{\text{GFLOP}}{\text{sec}}$. The arithmetic intensity of the double-precision version, DAXPY, is even worse at $2/24 = 0.083$.

We will revisit the notion of arithmetic intensity in kernels written for the Graphics Processing Unit (GPU) later in this class, as well as techniques to increase this intensity such as coalesced access of global memory and use of shared memory.

[4]SAXPY stands for "Single-Precision AX Plus Y" which is a function in the standard Basic Linear Algebra Subroutines (BLAS) library.