

Lecture 2: January 28

*Lecturer: Horst Simon**Scribes: Mark Howison*

Disclaimer: *These notes have not been subjected to the usual scrutiny reserved for formal publications. They may be distributed outside this class only with the permission of the Instructor.*

2.1 Amdahl's Law

Let s be the fraction of the work done sequentially, so that $(1 - s)$ is the fraction that is parallelizable. Let P be the number of processors. Then

$$\text{Speedup}(P) = \frac{\text{Time}(1)}{\text{Time}(P)} \leq \frac{1}{s + (1 - s)/P} \leq \frac{1}{s}.$$

Amdahl's Law is more of a law than Moore's Law, in the sense that it is mathematically formulated. If you have sequential pieces that cannot be done in parallel, they will be the limiting factor. The speedup for a parallel program is bounded by one over the sequential part, so you have to make the sequential part as small as possible, otherwise you don't get good speedup.

Historically, there has been much debate over Amdahl's Law. Many people said that parallel systems would never work because of the limitations given by Amdahl's Law. Twenty years ago, I attended a conference where a speaker proved in a paper that the maximum speedup for any parallel system is 815, by making arguments about computations that must always be sequential. It wasn't unreasonable. In the same conference, another presenter showed that you could get a factor of 1000 speedup out of 1024 processors. This led to the new notion of scaled speedup, or of weak scaling and strong scaling. There is always confusion of these terms in the parallel processing literature.

In weak scaling, you make the problem bigger as you increase the number of processors. In strong scaling, you keep the problem fixed. To solve the most demanding problems, do you want to use 100,000 processors to solve a problem that could be solved on a laptop? No. Instead, you want to solve problems that couldn't be solved on the laptop in the first place. Amdahl's Law applies to strong scaling.

Parallel processing requires additional work or overhead. If you use thread programming, you have the overhead of starting threads. If you use communication, you have to have primitives that lock data. If you have anything in your computer that takes milliseconds, it is not worth implementing in parallel. It takes large chunks of parallel work to make these overheads worthwhile.

2.2 Locality

All computation, whether sequential or parallel, is highly dependent on locality. No matter where you look, you have a memory hierarchy: processors, cache, memory, tape storage, etc. The movement of data to the processor often takes more time than the process itself. Small memory is fast, while large mem is slow. Hierarchies are useful because they are fast on average.

There is a huge performance gap between processors and memory. Processor performance has been increasing according to Moore's Law, doubling every 18 months. At the same time, memory speed has been increasing

very slowly, around 7% per year. The processor–memory gap has been growing larger. In order to bridge the gap, memory hierarchies have become more complicated. Patterson’s Law says that every decade we have an extra level of cache in the architecture.

2.3 Load Inbalances

If you partition work into different size chunks, the largest piece determines how fast you go. Unequal task size is always a limit to performance. You always want things nice and equal, but in many applications you have fundamentally unstructured problems. How do you make the pieces the same size? With algorithms for load balancing. This is not as necessary for sequential computation.

2.4 Single Processor Machines

Why do you want to look at a single processors first? Performance is of course a theme throughout this lecture. We spend tens of millions of dollars on these big systems, yet only get 10% of peak performance on many of them. Many people ask, if you can only get 10% of a 50 million dollar system, have you wasted 45 million dollars? If you have an algorithm that bumps the performance from 10% to 20% of peak, you could save money. The interesting thing is that a lot of performance that is lost is also lost on a single processor because of the memory hierarchy. Performance is measured against peak values, but many computations in parallel programs are done by moving memory around, and so most of the performance is lost in the memory system.

For this lecture, we will only look at single-core processors. Parallel programmers need to be performance programmers, because what’s the use of 128 cores if you dont use them all well?

In the idealized uniprocessor model, we look at bytes or words in the address space and operations that are either reads, writes, or arithmetic. Reads and writes occur in fast memory called registers. Arithmetic is performed on the values in the register.

Example:

Read address(B) into R1

Read address(C) into R2

$R3 = R1 + R2$

Write R3 to address(A)

The assumption is that each operation has the same cost. However, real processors have registers and caches, which store the values of recently used, nearby data, and different memory operations have very different costs.

2.5 Parallelism

Multiple “functional units” can run in parallel on one processor and can run operations in different orders, with different mixes having different costs.

Pipelining is a form of parallelism, analagous to an assembly line in a factory. An example (from Patterson) is that 4 people are doing 4 loads of laundry. Washing takes 30 minutes, drying 40 minutes, and folding 20 minutes. Without pipelining, the total time is 4×90 minute = 6 hours. With pipelining, as soon as the first is done washing, the next starts. The execution would instead take $30 + 4 \times 40 + 20$ minutes = 3.5

hours. The dryer is the limiting factor in this case. Pipelining is an important technology that improves bandwidth and is used in functional units to accelerate floating-point calculations.

Another example where parallelism is used in modern processors is *SIMD processing*. In a traditional processor, one operation produces one result. With SIMD, one operation can yield multiple results. For instance, SSE2 works on anything that fits into 16 bytes. In order to use SIMD, the data must be contiguous in memory, which could require extra effort for moving data around beforehand.

There are all tricks used on the processor and in memory architecture that help improve performance through parallelization. In theory, all of these details are hidden from the programmer and the compiler takes care of implementing them. In reality, the compilers are not as capable and the programmer has to perform optimizations, for instance by trying different compiler flags or writing assembly code.

2.6 Memory Hierarchies

The key picture to keep in mind is that memory exists at different levels. Fast, small memory is close to the processor, while large memory is farther away and slower. On-chip caches and registers access at processor speeds, while second level caches and main memory are much larger in size.

In order to use a hierarchy efficiently, you have to extract spatial and temporal locality from operations. Running everything randomly under this type of architecture would not work. Spatial locality refers to accessing nearby data. Temporal locality means reusing data often. A programming rule is that 20% of data is used very often, while 80% is touched only occasionally. There is a tricky benchmark called the GUPS developed at the NSA which does the opposite. It tries to read randomly across large memory, perform one or two operations, and write back to memory. In the last 10 or 20 years, there has been very little progress made on this benchmark, mostly coming from increases in overall memory speed. On the other extreme are matrix calculations that run extremely well on this type of memory hierarchy.

The processor-memory gap is the bane of computer architecture, and has to be gapped using memory hierarchies. Recent work on vector computers, like the Cray C1, have explored memory architectures without caches, but they are very expensive to build. Bandwidth has improved more than latency, which is getting worse.

2.7 Cache Basics

Cache is fast, expensive memory located close to the processor, mostly on-chip. It stores copies of data from main memory, and is often hidden from software. A *cache hit* accesses data already in cache and is cheap. A *cache miss* accesses data not in cache and is expensive, because it requires traversing the cache hierarchy. The *cache line length* is the number of bytes loaded together into the cache. Performing one load exploits spatial locality. *Associativity* refers to how memory is mapped and the resulting replacement policies. For example, with direct mapping, only one address (or line) can be in one location at a time. In a fully associative cache, any line can be in any location.

Multiple levels of cache have become a necessity because the processor-memory gap has grown so large. In the 80s, processors had one level of cache, in the 90s two levels, and now three levels. Caches have tradeoffs: a large, single-level cache seems beneficial, but you have more addresses to check, so it takes longer to replace data, and associativity requires more time. These tradeoffs are explored in computer architecture classes. For example, the Cray T3E eliminated one cache to speed up cache misses.

2.8 Membench and Memory Experiments

A *stride* of length s is a memory access pattern in which every s -th element is accessed. Consider a fixed stride length L and a loop over strides that grow longer from 0 to L . Repeating this inner loop many times and averaging over the outer loop gets rid of anomalies and leads to a measure of the memory access time based on this simple type of access pattern. For values of L that are smaller than the cache size, repeating many times and averaging guarantees that after a few times the whole array is sitting in the cache, which reveals cache hit time. For values of L larger than the cache size, a plateau of memory speed is reached. In practice, it is more difficult to measure cache sizes and access times on modern processors with complicated architecture.

Example: Sun Ultra-2i 333mhz processor. This test can determine the access time and size of the L1 cache, L2 cache, and main memory. It is a simple benchmark that uses straightforward code. The resulting plot of average time per load at different values of L shows the characteristics of the memory architecture and the orders of magnitude difference in latency in different parts of the memory hierarchy.

The Stanza Triad is a smaller benchmark for prefetching. A *stanza* is a unit stride of a given length L . The test performs a scalar multiplication of one array plus another array for the entire stanza, then skips forward by k elements before repeating. This is a bandwidth benchmark that measures GB/s. If cache locality were the only phenomena, the plot would be flat (e.g., Pentium 3), but prefetching kick in for larger values of L and has a significant impact on memory performance.

The lesson is that even for simple programs (what is simpler than reading streams from memory?), performance is a complicated function of architecture features. To write fast programs, you need to consider architecture, even small features such as the L1 cache size. It would be good to have simple models for designing algorithms, but modeling of this behavior is very difficult and not strongly predictive. Some general rules for increasing performance are to use blocking and tiling, which are divide-and-conquer methods that cut data into pieces that fit nicely into caches. This is more of an art than a science, and requires an understanding of the architecture and adjustments to data access patterns.

2.9 Matrix Multiplication

This is one of the seven dwarfs, and is very typical in dense linear algebra. Matrix multiplication is relatively simple (it has no complicated data structures) and is rich in examples of optimization ideas that can be applied to other calculations. It also has a high potential for performance enhancement and is probably the most studied algorithm in scientific computing.

Example: Sun Ultra-1/170 with a peak performance of 330 megaflops. The simple approach is to hand code three nested loops, leading to poor performance (20 megaflops). The Sun optimized library demonstrates what a specialist can do with sophisticated code (what you learn in this class, basically in HW Assignment 1). The higher performance comes from exploiting the memory hierarchy. PHiPAC rewrites the matrix multiplication code with automatic tuning tools and yields performance on par with the Sun optimized library.

A matrix requires 2D storage, but memory is only 1D. There are two simple ways to map a 2D array into 1D memory: map by column (column major) or by row (row major). By default, FORTRAN uses column major while C uses row major. With column major storage, accessing a row cuts across cache lines. In the worst case, each cache line has only one row element.

Assume for simplicity a model with two levels of memory, fast and slow. Assume that all data is initially in slow memory. Matrix multiplication requires n^3 arithmetic operations (denoted f). The number of memory

elements moved (denoted m) is n^2 . The minimum time for computing is $f \cdot t_f$ (where t_f is the time per operations), when all the data is in fast memory, but you have to account for getting the data from slow to fast memory. The actual time is

$$f \cdot t_f + m \cdot t_m = f \cdot t_f \cdot (1 + t_m/t_f \cdot 1/q)$$

where t_m is the time per slow memory operation and q is a parameter for measuring computational intensity, the key to algorithm efficiency. Running a large enough matrix even on a poor computer can achieve close to peak performance. Larger values of q mean that the time is closer to the minimum, $f \cdot t_f$. Many operations in scientific computing have $q = 1$, exposing the speed of memory access, which is typically 200 times slower than floating-point operations. The t_m/t_f factor measures machine balance. If large, it indicates a poorly designed machine for scientific calculations, one that is limited by memory traffic.

Matrix-vector multiplication implements $y = y + Ax$ using a simple double loop. The memory operations are to read the vectors x and y , read a row of A , then write back y . Therefore, $m = 3n + n^2$ is the number of slow memory operations. The number of arithmetic operations is $f = 2n^2$. This means that $q = f/m \approx 2$ which is not good, since the computation is limited by slow memory speeds. Larger values of q guarantee performance closer to peak.

2.10 Summary

The details of the machine are important for optimizing performance. For general code, the compiler takes care of the details, but high-performance computing requires tinkering.