Lecture 1 notes

1 Logistics

1. Basic information:
   • Name, level (undergrad, M.S., Ph.D.), department, registration status?
   • NYU Home account name (for parallel machine access)?
   • What programming languages do you know?
   • What numerical programming experience do you have?
   • Does your research involve any computational applications?
   • Have you done any parallel programming before?

2. Work: 3-5 homeworks (first will be next week) and a final project
   • Can work in groups on final project.
   • Projects can come from us, from your research, or from talking to other professors with parallel computing problems.

3. We will announce account information next week. We’ll use the ITS parallel machines.

2 Overview

The goal of this course is to learn to write fast programs. On the way, we will measure performance; tune programs to get the best performance on a single CPU; and write programs for parallel machines.

One common theme in the course will be measurement. When is a scientific code “slow?” How do you find the bottlenecks — the parts that take most of the time — so that you can improve performance with the least effort? What does it mean for a machine to be fast? The clock rate or peak arithmetic performance isn’t all there is to it! We want quantitative performance models to guide our software designs, and those models must be backed by measurement.
Another common theme will be *parallelism*, or doing multiple computations simultaneously. Scientific programmers have been writing parallel programs for specialized high-performance machines for a long time, but we will see parallelism everywhere — even in desktop machines. Computer architects are building machines with more and more parallel features: instruction-level parallelism within a single processor, vector processing hardware used for multimedia computing, highly-parallel graphics processing units (GPUs) in video cards, and multicore technology that puts multiple processors on a single chip.

*Communication speed* is an issue for high-performance programs, and this will be another theme in the class. It takes time to communicate between cores on a single chip, between processors on a single board, between boards in a single computer (e.g. between the motherboard and a video card), and between nodes in a parallel system. Even on desktop machines, the time that the processor needs to communicate with the main memory is often longer than the time that the processor needs to compute.

### 3 Architectural trends

In 1965, Gordon Moore wrote [5], “Complexity for minimum component costs has increased at a rate of roughly a factor of two per year.” Though Moore’s original prediction (later corrected to doubling every two years) referred to transistor counts, Moore’s name was attached to the observation that computer performance doubles every eighteen months. Other measures of computer complexity and performance have grown exponentially over the past few decades, too, like clock speeds, memory speed and capacity, and disk capacity.

While transistor counts double roughly every two years, other measures of computer power have different exponential growth rates. This has some unexpected consequences. One of the most significant consequences for us is this: processor speed has increased much more rapidly than memory speed, so that it is common for a modern computer to spend most of its time waiting for data to arrive from memory. Consequently, we have to be careful with memory use in order to achieve high performance.

---

1. The Mythbusters guys gave an entertaining metaphor for parallel computing applied to graphics at a recent NVidia conference [1]. It involves more paintball guns than you’ll probably ever see anywhere else.
Exponential growth in transistor counts has other unexpected consequences, too. Between increased transistor density and increased clock rate, the multi-GHz machines we have today generate an alarming amount of waste heat. Without cooling, these chips would be hotter than the surface of the sun (about 6000°C) [2]! This is a serious enough issue that my desktop machine — a “Prescott”-series Pentium 4 — not only turns off parts of the chip that are not being used, but it has thermal monitoring to slow down the system if the chip temperature gets too high! Largely due to power limitations, the rate of increase in clock rates and serial performance has slowed over the past few years, even though transistor counts continue to increase at the same rate [4, 3].

In addition to running into physical issues such as power dissipation, computer architects are having problems with turning more transistors into more performance. Modern processors are very complex, and part of that complexity comes from attempts to automatically schedule many instructions to run simultaneously at the hardware level. For the new generation of multicore machines which are appearing now, computer architects have put multiple independent processors (cores) on a single chip. These individual cores are not significantly faster than the processors in previous generations; it’s just that there are more of them. Consequently, understanding parallelism is now more critical than ever for achieving high performance, even on an ordinary desktop.

4 Understanding performance

In courses on algorithm design and analysis, we learn a simple model of the computer performance [2]. In this model, we count some basic operations, such as arithmetic. We assume these operations take most of the time; that they run serially; and that they all take about the same time to run. From these assumptions, we estimate how long computations will take. For example, we say it takes about $2n$ flops (floating point arithmetic operations, such as add or multiply). If we know our machine has a peak rate of some number of megaflop/s (millions of floating point operations per second), then presumably we can divide the number of flops in the computation by the speed of the machine in megaflop/s to find how long a computation will take. Unfor-
unately, the predicted time will usually be wrong — sometimes by orders of magnitude.

Though a naive performance model is useful for understanding how computational costs scale as a function of problem size, it can mis-predict actual performance. In fact, a modern machine runs many instructions simultaneously, and even the time to run the same type of instruction — a memory read, for example — can vary by orders of magnitude depending on the state of the system. To arrive at a more realistic model for predicting performance, we need to understand a little more about how modern machines work. We will focus on two aspects of the machine architecture: instruction-level parallelism and the effects of the memory hierarchy.

5 Single-CPU parallelism

5.1 Pipeline parallelism

A single processor can execute several instructions at once using a pipeline. Think of doing laundry: if you have several loads to clean, you would usually simultaneously fold the first load, dry the second load, and wash the first load. The three loads are being handled in parallel, with each load at a different stage. Similarly, in a simple pipelined architecture there might be five stages to executing an instruction:

1. Fetch the instruction from memory;
2. Decode the instruction (e.g. decide it’s an addition);
3. Execute the instruction (e.g. actually add two numbers);
4. Read from memory, if needed;
5. Write back the results (e.g. store the sum in a register).

In a processor with this five-stage pipeline, we might have five instructions in flight simultaneously. In general, if we have a $p$ stage pipeline operating at maximum efficiency, then we expect to take $m + p - 1$ cycles to execute $m$ instructions. This is a pretty good improvement over the time it would take if we ran the instructions serially ($mp$ cycles). But there’s a catch: our five-stage pipeline can only handle five instructions simultaneously if
those instructions are sufficiently independent. If the first instruction writes a result to a register and the second instruction uses that same register as input, then the second instruction cannot start to execute (enter stage 3) until the first instruction has finished writing (exited stage 5). In this case, there will be a “bubble” in the pipeline while the second instruction waits for the first to complete.

Pipeline bubbles can occur because of data dependencies or because of branches. In either case, they reduce the level of parallelism available to the processor. This might not seem like such a big deal with a five stage pipeline, but the original Pentium machines have a twenty stage pipeline — and my desktop, a Pentium 4 Prescott has a pipeline with 31 stages. Despite branch prediction and other clever tricks for mitigating the effect of pipeline stalls, pipeline bubbles are a serious performance barrier for these machines.

5.2 Beyond pipelines

In fact, the architectural picture is even more complicated that we have described. Modern “superscalar” chips aggressively schedule instructions to use many functional units at once, so that a processor can simultaneously execute several instructions in the “same” pipeline stage, assuming those instructions use different functional units in the hardware. These instructions can even be executed out of order, though the hardware ensures that in the end the output of the instructions is written in serial order. The details are interesting, but they are well beyond the scope of this class. Fortunately, we don’t have to understand all the details of chip architecture in order to write fast code. One of the jobs of an optimizing compiler is to schedule instructions in a way that lets the chip make the best use of these parallel features; and typically, compilers do a much better job at this sort of scheduling than human beings. The role of the human, then, is to make any available instruction-level parallelism obvious to the compiler.

The chip in my desktop actually takes the superscalar idea one step farther with a feature called hyperthreading. A hyperthreaded CPU has two pipelines, but these pipelines share the same set of functional units that actually do the work of executing instructions (doing arithmetic, accessing memory, etc). The idea is that if one program doesn’t provide enough parallelism to keep the processor busy, perhaps two will do so. From the perspective of the operating system, my machine has two “logical” CPUs.
5.3 Short-vector extensions

A pipelined, superscalar architecture is good at simultaneously executing many different types of instructions (or at least different stages of instructions). A vector unit is a special piece of hardware that does the same operation to several different pieces of data. This type of parallelism is sometimes called SIMD: Single Instruction, Multiple Data. The old Cray machines were vector machines, but vector processing then waned in popularity for a time. Commodity machines started adding vector instructions about ten years ago because they are very useful for accelerating graphics and multimedia applications. The early Pentium chips had MMX (variously explained as “MultiMedia eXtensions” or “Matrix Math eXtensions,” though Intel never gave an official expansion of the acronym); later Pentium chips included SSE (Streaming SIMD Extensions), and then SSE2. A Pentium with SSE2 instruction is capable of doing simultaneous arithmetic operations with 16 pairs of byte operands, 4 pairs of float operands, or 2 pairs of double operands. Some chips from IBM or from Motorola (including the G4 and G5 chips used in Macs before Apple switched to Intel) have similar SIMD instructions, called the Altivec instructions.

Modern GPUs (graphics processing units) take this sort of vector processing to a new level. We may talk about graphics processors more, later in the class.

6 Memory hierarchies and locality

Between faster clock rates and more parallelism, processor performance (as measured by benchmark codes) has doubled roughly every 18 months. In contrast, memory latency (the time between asking for a piece of memory and when the first bit arrives) has doubled roughly every ten years. By now, fetching data from main memory costs around a hundred nanoseconds, while instructions can be started at a rate of several per nanosecond. Computation is now cheap; it’s getting data that’s expensive!

Fortunately, in any short time period, most programs use only a small amount of data (sometimes called the working set). This is called temporal locality. Most programs also have spatial locality, that is, if the program reads or writes one memory word, it is likely to read or write nearby memory words around the same time.
Using locality, computer architects have a partial solution to the problem of feeding the execution stream in the face of slow memory. The solution is a memory hierarchy consisting of different “levels” of storage. Lower levels are fast but small, and higher levels are slow but large. Because of locality, most data a program uses can be kept in fast cache memory, with relatively few reads from the slow main memory.

6.1 Hierarchy levels

My desktop machine (a 3 GHz Pentium 4 “Prescott,” model 630, stepping R0) has the following levels of memory:

1. The register file, which operates at the same speed as instruction execution.

2. A 16 KB L1 data cache (“level 1” cache\(^3\), which has a latency of 3 cycles (1 ns).

3. A 2048 KB L2 data cache, which has a latency of 28 cycles (just under 10 ns).

4. 1 GB of main memory (DRAM, or dynamic random access memory), which has a latency of about 100 ns.

5. 16 GB of disk, which has a latency of about 10 ms.

In addition, the system provides virtual memory, which means that there’s a data structure that maps the addresses used by different programs to physical memory addresses. Part of this mapping is kept in a fast memory called a translation lookaside buffer (TLB). If a program uses address that isn’t listed in the TLB, the system consults the full page table, which is stored in main memory. So on a TLB miss, getting data from memory can cost two or three ordinary memory accesses (or more).

---

\(^3\)This machine also has a 16 KB cache for code, called the execution trace cache. The execution trace cache actually stores “micro-operations” rather than the machine instructions that are normally stored in memory. This is because the Pentium instruction set is sufficiently complicated that the machine has a sort of hardware interpreter to convert those instructions to a lower-level representation before running them. It’s like the hardware version of a Just-In-Time compiler for the Java Virtual Machine.
6.2 Cache lines and spatial locality

The L1 data cache on my machine is organized into 64-byte cache lines. Whenever there is an L1 cache miss – that is, whenever the processor needs a piece of data and cannot find it in the L1 cache – it reads 64 bytes of data from the L2 cache, even if the processor only requested four or eight bytes. That way, when the processor needs the next four or eight bytes, it will already be waiting in the L1 cache.

Because caches are organized this way, we can get the best performance by organizing programs to maximize spatial locality. For example, consider the following two versions of a loop to compute the centroid of \{(x_i, y_i)\}_{i=1}^n:

```c
void centroid1(double* xy, int n, double* result) {
    double x = 0, y = 0;
    int i;
    for (i = 0; i < n; ++i) x += xy[2*i];
    for (i = 0; i < n; ++i) y += xy[2*i+1];
    result[0] = x/n;
    result[1] = y/n;
}

void centroid2(double* xy, int n, double* result) {
    double x = 0, y = 0;
    int i;
    for (i = 0; i < n; ++i) {
        x += xy[2*i];
        y += xy[2*i+1];
    }
    result[0] = x/n;
    result[1] = y/n;
}
```

On a data set of five million points, I get the following timings (in nanoseconds) and effective processing rate (in MB/s):

<table>
<thead>
<tr>
<th></th>
<th>Time (ns)</th>
<th>Rate (MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>centroid1</td>
<td>4.4</td>
<td>8.7</td>
</tr>
<tr>
<td>centroid2</td>
<td>4.4</td>
<td>8.7</td>
</tr>
</tbody>
</table>

This is when I compiled using gcc with the optimization level `-O2`. At the highest level of optimization (`-O3`), the optimizer rearranged the calculation so that the sums were all...
Ver 1: 49967165 (4.99672 ns/double; 1601.05 MB/s)
Ver 2: 26626221 (2.66262 ns/double; 3004.56 MB/s)

That is, the second version runs about twice as fast as the first version — which makes sense, because the first version loads each cache line the data set twice. The observant reader will notice that the time to fetch a cache line seems to be about 40 ns, which is somewhat less than we might expect. I'm not sure whether to attribute that to overlapping requests to the memory subsystem or to the hardware prefetching in this model of Pentium.

6.3 Cache eviction and temporal locality

When we read more data than can fit in a cache, we have to **evict** old data in order to make room for new data. The choice of which entry should be evicted is the **replacement policy**. A common policy is LRU: replace the entry that was **least recently used**. If a program has temporal locality — i.e., if data is likely to be used in bursts — then newer entries are more likely to be used again soon, so evicting the oldest entry is natural. In practice, most implementations would use an approximation to LRU that requires only counters with a finite number of bits.

We can use a pure LRU replacement policy with an **associative cache** in which any piece of data can be stored in any location in the cache. However, fully associative caches cost a lot of hardware resources. An alternative that requires far less hardware is a **direct mapped cache**, in which any address is associated with a unique cache location. In a direct-mapped cache with $c$ entries, address $a$ goes in cache slot $b$ if $a \equiv b \mod c$. So if we had a four-slot direct-mapped cache with entries 0, 1, 2, and 3, and we read address 200 followed by address 204, both would map to slot position 0. We would have to evict the data from address 200 to resolve the conflict.

Direct-mapped caches use less hardware than fully-associative caches; but they also don’t let us take advantage of temporal locality. A **set-associative** cache costs less than a fully-associative cache, but still lets us use temporal locality. In a set-associative cache, data from each address can go into one of a small number of cache slots. To extend our example above, suppose we had a 2-way set-associative cache with eight entries (and so four sets). Then if we read address 200 followed by address 204, both would go into set 0 done at the same time that I created the data array, which meant both centroid routines appeared to run about instantaneously.
without conflict. If we then read address 208, we would evict the data from address 200, since that data was least recently used.

6.4 A memory microbenchmark

The membench program times regular read/write memory patterns. In pseudocode, the inner loop that we are timing looks like this:

```c
for (i = 0; i < len; i += stride)
    x[i]++;
```

where `x` is a 32-bit integer. For `stride = 4`, for example, the loop goes through a buffer of length `len` and increments every fourth four-byte integer.

I ran membench on my desktop, and in Figures 6.4 and 6.4, I show the average time to read and write an integer for different size buffers. The picture is complicated, but we can make a few observations from what we know about the memory subsystem:

- In Figure 6.4, there is a sharp “knee” at a stride of $2^6 = 64$ bytes for the small data sets. This corresponds to where we go from a stride smaller than the cache line size to a stride longer than a cache line.

- In Figure 6.4, there is a sharp knee for the large data sets, we see two “plateaus” at around 70 ns and 110 ns, respectively. I believe these plateaus correspond respectively to the time to access main memory and the time to read memory when there is a TLB miss.

- When the data set is smaller than the L2 cache size (2MB), the worst access time is about 10 ns. Once the data set reaches the size of the L2 cache and the program starts to miss the cache and go to main memory, average access times get much worse.

- When we look at short strides in Figure 6.4, we that for each stride there is a “knee,” a place where the average access cost goes up dramatically. This knee happens earlier for longer strides. This is because of the aliasing that occurs in set-associative caches.

- Looking at the bottom of Figure 6.4, we notice that at longer longer strides (longer than one 4K memory page), we can only access 256 bytes (64 integers), before we start to see TLB misses. This is because the TLB only has 64 entries!
Figure 1: Synthetic memory benchmark: access time vs. stride for different total sizes.
Figure 2: Synthetic memory benchmark: access time vs. bytes accessed for different strides.
References

[1] Adam & Jamie draw a MONA LISA in 80 milliseconds at NVIDIA’s show. http://www.youtube.com/watch?v=fKK933KK6Gg


