CSC367 Notes

Parallel Programming

Last updated January 20, 2024

1 Why do it

Can't fit too many transistors in a small place or have too many operations per seconds. Dynamic power is proportional to V^2fC, and because increasing f also increases supply voltage, there's a cubic effect. Cramming too much power density generates a lot of heat. Hence, frequency cannot keep increasing.

We can increase capacitance C. Increasing cores increases capacitances, but only linearly. If you have multiple cores, you increase capacitance C, but it's linear, not cubic. Now you have multiple cores, you can run more instructions, so you can lower the clock speed to get the same instruction throughput. You can spread out this heat problem quite better.

Parallel systems are much more efficient in terms of power dissipation.

1.1 Is Moore's Law Dead?

Moore's law, in terms of chip density, is not totally dead yet, but we are hitting the limit due to quantum mechanics. The wires are getting so thin that they are getting smaller than the electron, so you get more power dissipation from leaking electrons from the chip.

Chip density continues to increase by double every 2 years, but the clock speed has pretty much hit a limit.

Parallel systems have become unavoidable if you want to keep growing the processing power. Number of processor cores may double instead. Power is leveling off.

1.2 How do we measure speed?

HPC (high performance computing), the unit goes down to **flop** (floating point operations, usually doubles). Flop/s (pronounced flops) mean floating point operations per second. Bytes represent size of data; a float is 8 bytes.

Lots of our machine are in the gigaflop range, but many of the fastest computers are in the petaflop or exaflop range. The fastest machine in the world, the frontier machine, has over 8.6M cores in it. You can see that no other machine on this list has more cores than that. Parallelism, if you use it properly, works great.

1.3 Clusters

We're going to use SciNet, a small version of frontier. It does not have 8.6M cores. This is what we'll use for the course.

SciNet itself services a huge number of researchers all across Canada, but they have a smaller segment of SciNet called 'teach' that is used just for courses.

There are 42 notes, each of them has 16 cores per node. There are other courses that use this cluster, and later in the course, we'll do distributed memory stuff that will use multiple nodes. Very quickly, it will take time for your code to queue, so you may not get immediate feedback.

1.3.1 Nodes

The login nodes are meant for you to schedule your jobs. Keep your work that you do with login nodes as light as possible. Do what you want with SSH, but do not do anything that would install anything on the login nodes.

1.4 What Does a Parallel Computer Look Like?

What does a shared memory parallel machine look like?

- Shared memory: processors can communicate to each other through memory
- There are multiple stages where you can hook in the memory to sync with each other

1.5 Concrete Examples of What Happens When I Want to Optimize Code for a Parallel Architecture

Let's start with matrix multiplication. Assume that every matrix is squared and every dimension n is a power of 2.

Next, let's consider what system we will compute the multiplication on.

- Two chips
  - 9 cores per chip
  - Has a floating-point unit; has "bulk" instructions (like fuse-multiply-add – two operations in one)

So, what's the peak? If I want to optimize for something, what is the theoretical top throughput I can put through the processor?

$$\text{Peak} = \underbrace{\left(2.9 \times 10^9\right)}_{\text{clock freq}} \times \underbrace{2}_{\text{chips}} \times \underbrace{9}_{\text{cores per chip}} \times \underbrace{16}_{\text{CPU instruction per cycle}}$$

We have our code, we have our architecture, and we have our target best performance we can get. Let's look at everything we have to do.

- Firstly, identify our application: we want to matrix multiply.
- We have an algorithm. Bring up some pseudocode for matrix-multiplication.
- We'll use some framework (programming language) to implement that algorithm.
- On the bottom level, it will be operating on the CPU/GPU/at least somewhere.

On Python, for a 4096×4096 matrix, on the Haswell system it takes about 6 hours. What's the calculation?

- $2n^3 = 2^{37}$ floating point operations $\longrightarrow$ 21042 seconds
- Python gets $\frac{2^{37}}{21042}$ = 6.25 MFLOPS (mega flop per second)
- Yet, the peak is 836 GFLOPS
- Python gets 0.0075% of peak

So, why is it so slow?

Let's copy the code into Java. On the Haswell architecture, it takes about 46 minutes (2738 seconds). Better than 6 hours, 8-9 times faster.

So why not C? Well, it takes 1159 seconds (19 minutes). So, we have an 18× speed improvement over python.

Yet even the C implementation is a tiny fraction of the peak of the machine. We've utilized almost none of this processor.

In parallel systems, we have a tradeoff. We have a much manageable power dissipation, and theory we have a lot of power, but how do we write programs that can utilize all that power? And why is Python so slow and C so fast? How can we get C to approach the peak of the machine?

- Python is interpreted
- Java uses a virtual machine
- C is compiled directly to machine code. There's no overhead from having to interpret the code

Any optimizations we can do to utilize as much of the processor as possible? Yes, we can.

- We can exploit the fact that the cache exploits spatial locality

You can profile cache misses: `valgrind --tool=cachegrind ./mm`

This is an example of how small modifications to a program can have a big impact. Later, we'll get into more possible optimizations we can do.

1.6 Compiler Optimizations

Optimizing gives us speedups. Quite a lot. Nothing else to see here. However, don't trust your compiler too much.

1.7 Parallel Loops

Want to parallelize a loop? A natural thing is, well, we are doing this work to accumulate sums into rows of the result, C, so why not assign the work of each row to a thread? Each thread can be working away at accumulating into rows of C, and they don't need to worry about conflicting with each other. They have no dependence between different rows.

One way to do this is by using a `#pragma`. Here, we'll be using `#pragma omp parallel for` or `clik_for` or any other parallel programming model you like. It will spread out work among threads.

This gives us a very big jump, as we have the perfect ideal speedup from parallelism. We distribute the work across 18 different cores, meaning we are able to do it 18 times faster.

However, this is almost always not going to be the case for you. This is very unusual; people refer this algorithm as extremely parallel where you can subdivide the work into each core without any communication or dependencies.

Yet, we're at 5% peak. There's still a lot of unused potential. There are lots of more that can be done:

- Tiling
  - Makes data access patterns better
- Vectorization
  - Allows for GPU-like operations
- Matrix transposition
  - Change the data layout of matrices to make data access patterns more efficient
- Data alignment
  - Less important on modern x86 architectures, but better for smaller embedded systems

- Data alignment
- Preprocessing
- AVX instructions

Taking all of this and applying it to the parallel loop and we can get to 40% of peak. It's not going to be 100% of peak (that's really hard), but it's a very big improvement than what we started with.

1.8 How do I write fast code?

- Think
- Code
- Test/run
- Repeat

Consider:

- What hardware am I writing on?
- What is going on with my memory and arithmetic units?
- How should they be interacting for me to get the best performance?
- MOST IMPORTANT: Test what you do! It's easy to waste a lot of time if you forget this profiling step. Profiling is the only way to know if you're moving in the right direction.

2 Single Processor Machines

Before getting to parallelism, we need to start off with a single-processor version of your code.

Most of the time, applications are not going to use anywhere close to peak of the machine. The reason, much of the performance is lost on a single processor. The code running on one processor often runs at only 10-20% of the processor peak/

Most of that loss is in the memory system. Moving data takes much longer than arithmetic and logic. When we're going all the way to DRAM to fetch something, it's an eternity than LRU cache. The closer you can get to accessing memory physically located close to the processor, and high-speed low latency memory, the better you can do. This is the reason why caching exists – going to main memory is very slow.

2.1 Idealized Uniprocessor Control

- Processor
  - Control
  - Arithmetic (ALU, FPU)
- Memory
  - Main memory

The processor operates on variables: integers, floats, pointers, arrays, structures, etc.; the processor performs operations on these variables (arithmetic, logical, etc.), and the processor **controls** the order as specified by the program. ALUs can **only perform operations on values in registers.**

Each operation has roughly the same cost – for example, I can say that add takes the same time as multiply. If control, load, and store are not free. Branching can stall a lot of things. The x86 pipeline is long and very complicated, so if it expects to go somewhere in your program and you go somewhere else, it will have slowdowns.

2.2 What do Compilers Do?

The job of the compiler is not to optimize your code. It's just to do all of the annoying work of translating to machine code for you. It will take your C code, translate it into

some assembly code for that particular processor, and it will take care of loading stuff into the register file for you. It will do some optimization, but it's not perfect or some magic bullet that will make beautiful code for you automatically.

Your compiler should reduce the no. of registers used; however, solving that is NP-hard.

Your compiler can interchange loops for you, can improve register use, but GCC is not going to do that for you by default. Here are some other optimizations:

- Unrolling
  - Constant number of iterations, get rid of the loop. Makes the program take more space, however. Also, you might not be able to make the best use of your instruction cache, and it makes it harder for your compiler to do register allocation, and you might get register spillovers, so the compiler has to store some partial results in memory and then load them back into register. This kills performance.
- Fuse loops
- Eliminate dead code
- Strength reduction ($\times 2$ changes to bit shifts)

How do you know that your compiler has done your job? Compilers are a black box when we normally use it. We call the compiler; we get some machine code and we run the machine code. What can we do?

- Profile (like always, you **have** to profile)
- Look directly at the assembly code (a burden you must do)

Why can't we trust the compiler at optimizing? They give up on complicated code. A compiler applies optimizations with some general policies: sometimes, you can outsmart a compiler even on simple code because you know the specifics of it. Compiler optimizations also tend to take forever.

When they work, they work. But most of the time, they don't, and they'll leave your code as it. What is happening? This is why compiler explorer is very good, as it will

highlight parts of your code where optimization failed. Don't put all your faith in your compiler.

2.3 Caching: False Sharing

What happens if:

- Two processors grab the same cache line
- One processor writes to something on the cache line
- The other processor won't get that update

How do we fix this? Cache coherence. All caches must represent what is in memory at all times. To ensure that it happens, when the processor writes something in cache, it has to transfer that update to the other processor. The same applies if the other processor writes to another element.

The problem with this, is that these different processors don't care about what these values are. The **false sharing** thing is totally pointless, and there is no point to do this with a lot of overhead.

This can happen and is something to be aware of when you write your program.

The reason why it's called false sharing, is because on multicore machines, the only way to communicate between cores is to share memory. False sharing is some degenerate behavior that shouldn't be happening.

2.4 How do we get around false sharing?

Different cores must access different caches.

That is, if core 1 will use a and core 2 will use b, a and b should be in different caches.

You can do this by `malloc`ing them separately.

3 Memory Hierarchy

The hierarchy goes like this, from fast / close to slow / far:

- Processor
  - Control, ALU, registers, on-chip cache
  - Latency: 1ns, size: KB
- Second-level cache (SRAM)
  - Latency: 10ns, size: MB
  - This is why a cache miss is *so* bad – caches are orders of magnitude faster!
- Main memory (DRAM)
  - Latency: 100ns, size: GB
- Secondary storage (Disk)
  - Latency: 10ms, size: TB
- Tertiary storage (Tape/cloud)
  - Latency: 10s, size: PB

Processor

The chip has a cache, rationale is cost. Because fast memory is expensive, you will not want to spend a lot of money to have a lot of it. It will be small, so you can fit it to the chip. The closer the chip is to the processor; you have less latency (because the speed of light).

Main memory

The volatile chips you would have on your motherboard.

Disks / SSD

Magnetic disks are really cheap.

Tertiary storage

AWS has big warehouses full of tape. They have robots that get the tape, slot it into a machine, and that's how you get your data. You can put in a bunch of data but accessing them is going to be *really* slow.

Alternatively, for servers, you have latency from the internet.

3.1 Latency and Bandwidth

Latency: The time it takes to go from one point to the other. For example: 100ns

Bandwidth: The amount of data transferred in a fixed period of time. For example: 100MB/s

3.2 Approaches to Handling Memory Latency

Temporal locality: Reuse data that is already in cache. Eliminate memory operations by saving values in small, fast memory (cache or registers) and reusing them (bandwidth filtering).

Spatial locality: Operate on data that are stored close to each other in memory and can be brought into cache together. Take advantage of better bandwidth by getting a chunk of memory into cache (or registers) and using the whole chunk.

Cache prefetching: Requesting data from memory ahead of its use. The cache is low-level hardware that is designed to be extremely fast, so you don't have direct control over it. Make sure your code is cache friendly.

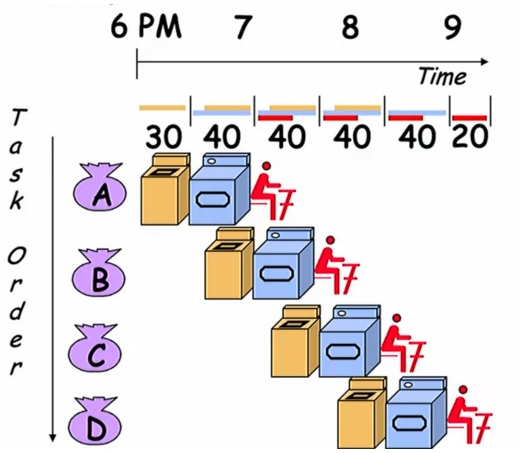
3.3 Cold and Warm Starts

Cold cache: When the data in the cache is stale: needs to read from memory. When profiling, you need to run your code several times in order to get a more representative idea of how long it takes the program to run. Median or average are great ways to do it.

Warm cache: The opposite

4 Pipelining and Parallelism

Dave Patterson's Laundry example:



Pipelining is when you do this instead of do everything sequentially. Pipelining helps with bandwidth but not latency.

4.1 SIMD (Single Instruction, Multiple Data)

Allows you to do multiple scalar operations at a time, making it vectorized. This boosts the throughput of your program.

Challenges: alignment in memory and gathering. This assumes that all your data is stored sequentially. If your data is spread out, this could result in latency.

5 Performance Models in Optimization

Dense matrix multiplication is used very frequently across many domains.

5.1 Using a Simple Model of Memory to Optimize

Very important!

We define the following terms:

- m: no. of memory elements (words) moved between fast and slow memory (slow memory operations)
- t_m: time per slow memory operation. **I have no control.**
- f: over the course of a whole program, how many of these arithmetic operations (FLOPS) occur over the entire operation
- t_f: time per arithmetic operation $\ll t_m$. **I have no control.**
  - Way less than t_m, because in order to execute an instruction, it's going to be some multiple of clock cycles. However, accessing from slow memory is going to make many orders of magnitude of clock cycles
- $q = \frac{f}{m}$: average **number of flops** per **slow memory access** (computational intensity / algorithmic intensity)
  - How many operations I can do per slow memory access
  - Minimum time possible = $f \cdot t$ when all data is in fast memory. It's never going to be the case; all of the time I must access slow memory
  - The actual time = $f \cdot t_f + m \cdot t_m$
  - $= f \cdot t_f \cdot \left(1 + \frac{t_m}{t_f} \cdot \frac{1}{q}\right)$

We want to make q as large as possible, as that minimizes the actual time in the equation above.

- Larger q means time closer to minimum $f \cdot t_f$
- $q \geq \frac{t_m}{t_f}$ needed to get at least half of peak speed
- $\frac{t_m}{t_f}$ is known as the machine balance. We don't have control over this.
  - How many t_fs it would take for one slow memory access

5.2 Matrix Vector Multiplication

Assume **x**, **y**, and one row of A fit in fast memory.

The code goes like:

```
for i in range(0, n):
    for j in range(0, n):
        y[i] = y[i] + A[i, j] * x[j]
```

Assume the cache line is **one word** (that holds a double), and that we can store as many cache lines as we want. So, what's m, f, q? (we note that $q = \frac{f}{m}$, known as $\frac{\text{total}}{\text{slow}}$).

$$m = \text{number of slow memory refs} = 3n + n^2$$
$$f = \text{number of arithmetic operations} = 2n^2$$
$$q = \frac{f}{m} \approx 2$$

Why? If we rewrite our code:

```
# read all of x into fast memory, taking n slow refs
# read all of y into fast memory, taking n slow refs
for i in range(0, n):
    # read row i of A into fast memory
    # over this procedure, this means n ** 2 slow refs
    for j in range(0, n):
        y[i] = y[i] + A[i, j] * x[j]
# write all of y back to slow memory
```

Matrix-vector multiplication is limited by slow memory speed. If we want to utilize half-peak, we need a q that is equal to the machine balance $\frac{t_m}{t_f}$. As processors become faster and faster, memory doesn't always keep up – machine balance $\frac{t_m}{t_f}$ they can be between 5 to 25.

5.3 Naïve Matrix Multiply

Assume all data fits in fast memory. Here's the algorithm for $C = C + AB$

```
for i in range(0, n):
    for j in range(0, n):
        for k in range(0, n):
```

```
4                 C[i, j] = C[i, j] + A[i, k]*B[k, j]
```

Looking at this:

- $2n^3 = \mathcal{O}(n^3)$ Flops.
- We only make $4n^2$ slow memory references as we read from each of the 3 matrices once and write to one of them
- $q = \frac{2n^3}{4n^2} = \frac{1}{2}n \in \mathcal{O}(n)$
- Large matrices can have q overtake the machine balance.

Realistically, you're not going to be able to fit 3 matrices in fast memory at a time.

5.4 Naïve Matrix Multiply Breaking The Assumption

Change our assumption: only three matrix rows can fit to memory. Now what?

```
1 for i in range(0, n):
2     # read row i of A into fast memory (n -> n ** 2)
3     for j in range(0, n):
4         # read C[i, j] into fast memory (n -> n ** 3)
5         # read B[:, j] into fast memory (n -> n ** 3)
6         for k in range(0, n):
7             C[i, j] = C[i, j] + A[i, k]*B[k, j]
8         # write C[i, j] into slow memory (1 -> n ** 2)
```

$$m = n + (n + n(1 + n + 1)) = n^3 + 3n^2$$

So:

$$q = \frac{f}{m} = \frac{2n^3}{n^3 + 3n^2} \approx 2 \text{ for large } n$$

No improvement over matrix vector multiply. The inner two loops are just matrix-vector multiply, of $A[i, :] \times B[:, j]$, similar for any order of 3 loops.

B gets read n times, A gets read once, C gets read and written 2 times in the inner most loop.

So, how can we reduce memory accesses?

5.5 Block / Tiled Matrix Multiply

Assumption: 3 tiles (individual blocks) can fit into memory

Consider A, B, C to be $N \times N$ matrices of $b \times b$ subblocks where $b = \frac{n}{N}$ is called the block size.

```
for i in range(0, N):
    for j in range(0, N)
        # read block C[i, j] into fast memory. Cache does this automatically.
        for k in range(0, N):
            # read block A[i, j] into fast memory
            # read block B[k, j] into fast memory
            C[i, j] = C[i, j] + A[i, j] * B[k, j]  # "recursive" matrix multiply, but this one assumes all three inner matrices fit in fast memory
        # write block C[i, j] back to small memory. Cache does this automatically.
```

So $m = (2N + 2)n^2$ (no. of slow memory accesses).

- $m = Nn^2$ (read each block of B N^3 times: $N^3 b^2 = N^3 \left(\frac{n}{N}\right)^2 = Nn^2$)
- $+Nn^2$ (do the same for A)
- $+2n^2$ (read and write each block of C once)
- $= (2N + 2)n^2$

So, a very common technique is to tile recursively for each cache level up to L1, then even into your register file. This is an inductive idea. The fastest matrix-matrix multiplications will tile everything for every level.

So:

- m is the no. of memory traffic between slow and fast memory
- Matrix has $n \times n$ elements, and $N \times N$ blocks, each of size $b \times b$
- f is the no. of floating point operations, $2n^3$ for this problem
- $q = \frac{f}{m}$ is our measure of algorithm efficiency in the memory system

The computational intensity q is:

$$q = \frac{f}{m} = \frac{2n^3}{(2N+2)n^2} \approx \frac{n}{N} = b \text{ for large } n$$

So we can improve performance by increasing the block size b. However, there is a limit of how much we can increase the block size due to hardware.

This analysis assumes that all 3 tiles of $A,\ B,\ C$ can at the same time fit into fast memory.

If M_{fast} is the size of fast memory, then the previous analysis shows that the blocked algorithm has computational intensity:

$$q \approx b \leq \left(\frac{M_{\text{fast}}}{3}\right)^{\frac{1}{2}}$$

5.6 Basic Linear Algebra Subroutines (BLAS)

Matrix-vector multiplication has a peak limit, as this is fundamentally limited by math. For matrix-matrix, this is different.