# CSC367 Notes

Parallel Programming

Last updated February 8, 2024

# Contents

# 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 $4096 \times 4096$ 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 $18\times$ 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.

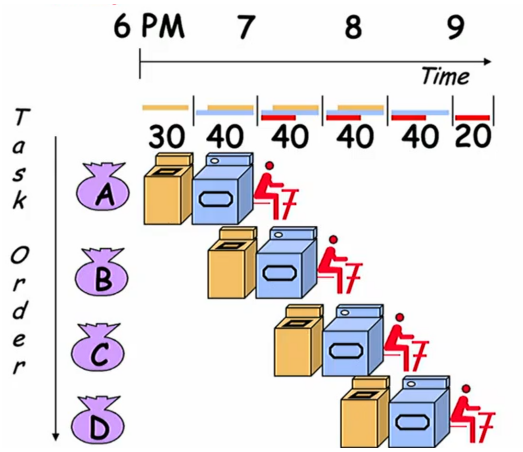
## 3.3 Cold and Warm Starts

**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 $t_f$s 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):
            C[i, j] = C[i, j] + A[i, k]*B[k, j]
```

Looking at this:

- $2n^3 = \mathcal{O}\left(n^3\right)$ 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?

```
for i in range(0, n):
    # read row i of A into fast memory (n -> n ** 2)
    for j in range(0, n):
        # read C[i, j] into fast memory (n -> n ** 3)
        # read B[:, j] into fast memory (n -> n ** 3)
        for k in range(0, n):
            C[i, j] = C[i, j] + A[i, k]*B[k, j]
        # 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 $M_{\text{fast}}$ 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 – it achieves way more throughput.

## 6 Parallel Architectures and Parallel Algorithm Design

How can processors communicate with each other?

- Use shared memory
  - Most laptops use this
- Use distributed memory

  - each processor gets its own memory
  - multiple SciNet nodes
  - anything where the processors are not sharing RAM
  - multiple processors to a socket – for example, a motherboard that has a socket to two CPUs
- SIMD and vector / CUDA
  - A graphics cards. It's not a general-purpose processor; it is designed for doing vector operations in parallel
- Hybrid: A mix of the above
  - Such as SciNet

## 6.1 Parallel Algorithm Design

- What parts of my program can be running concurrently? Not everything can possibly run concurrently; there are sometimes dependencies between different tasks
- How can we distribute those tasks efficiently? Those tasks will consume some input and produce some output. Now that we've decomposed those tasks, how are we going to deal with that?
- Once you have a lot of tasks running concurrently, you may have contention for resources. How do you make sure that you are not overloading a resource?
- At some point, you'll have to synchronize your programs. When will that synchronization happen. If I distribute my tasks, will that reduce synchronization, or will I pay overhead?

All of these depend on profiling. These are decisions I have to make, trade-offs I'll have to decide about. Profile and I'll know what's better.

## 6.2 Decomposition and Tasks

A task is a very abstract concept.

- **Decomposition**: dividing the computation in a program into **tasks** that can be executed in parallel
- **Task**: unit of computation that can be extracted from the main program and assigned to a process, and which can be ran concurrently
  - Tasks can range from individual instructions to entire programs. Which one is the best? It depends.

### 6.2.1 Example: Matrix-Vector Multiplication

Multiply a $4 \times 4$ dense matrix with a vector of size 4: $Ab = c$

Say that computing each output item is a task:

$$Ab = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}$$

Computing $c_0, c_1, c_2, c_3$ can be done independently, other than the fact that I need to put the results in the same output array.

## 6.3 Dependencies

On the notion of dependences, it's more about: what information does each task need before it can execute what it needs to execute? A way to neatly represent this is by using a task dependency graph.

- A task might need data produced by other tasks $\Rightarrow$ must wait until inout is ready

- Dependencies create an ordering of task execution ⇒ task dependency graph
- Directed acyclic graph: tasks as nodes, dependencies as edges
  - Start nodes: no incoming edges, finish notes: no outgoing edges

## 6.4 Granularity of Decomposition

Granularity: determined by how many tasks and what their sizes are.

- Coarse-grained: a small number of large tasks
  - Example: every task computes many outputs of $\vec{c}$. Divide $\vec{c}$ evenly, say $\frac{1}{2}$ of $\vec{c}$ will be computed by task 0, and the other half be computed by task 1.
  - Lots of computation performed before communicated is necessary; good match for message-passing environments (MPI, covered later)
  - Minimal synchronization needed. Infrequent communications: great for distributed systems who can't communicate all the time quickly
- Fine-grained: a large number of small tasks
  - Example: every task computes each output of $\vec{c}$
  - More suitable for **shared memory** environments (Pthreads, OpenMP). Communication has overhead. Frequent communication may be necessary, but communication in shared memory environments is faster.
  - Lots of synchronization needs to be done.

Note: we are decomposing into tasks; nothing is said about data.

### 6.4.1 Parallelism Granularity

How many processing is performed before communication is necessary between processes.

### 6.4.2 Degree of Concurrency

**Maximum degree of concurrency:** max. no of tasks that can be executed simultaneously at any given time (typically less than total no. of tasks, if tasks have dependencies)

**Average degree of concurrency:** Average number of tasks that can be executed concurrently during the program's execution

You want to have as many tasks running when possible.

$$\text{Avg degree of concurrency} = \frac{\text{Total amount of work}}{\text{Critical path length}}$$

Critical path: the longest path between any pair of start and finish nodes

Critical path length: Sum of node weights along the critical path

**Shorter critical path ⇒ Higher degree of concurrency**

You should only bother about stuff on the critical path. Of course, it can still take up resources of things not on the critical path. Optimize the thing on the critical path, and then that will no longer be your critical path. Something else will take more time, so optimize that.

## 6.5 Granularity and Concurrency

If granularity of decomposition is more fine-grained, then more concurrency is available. More concurrency ⇒ more potential tasks to run in parallel. If so, then can we reduce program execution time by increasing granularity of tasks? No. You'll hit a point where you do not have enough resources to efficiently handle all these tasks. You may reach a point where you cannot split up a task to anymore.

## 6.6 Task Interactions

Certain tasks may have to implicitly be synchronization from memory and that may not show on the synchronization graph. That often involves transferring some data.

If a value is stored for each task, then I need to communicate that value across all tasks. That is probably not the best strategy, and that will not show up on the task graph.

### 6.6.1 Read-only Interactions

Tasks only need to read data shared with other tasks.

### 6.6.2 Read-Write Interactions

Tasks can read or write data shared with other tasks. For example, in matrix multiply $Ab = \vec{c}$, tasks must write partial sums to $\vec{c}$.

An important detail: read-write interactions have some overhead. They synchronize through memory. If you are sharing information between multiple nodes in a compute cluster, you do not want to be constantly synchronization. Instead, use shared memory – it has a memory overhead way smaller. So, schedule these read-write interactions in shared memory.

## 6.7 Mapping Tasks to Processes

Mapping: assign tasks to processes.

This is how we'll be taking our bag of tasks earlier by doing decomposition. We have all these tasks, and now we want to assign them to processing power. We have a couple rules of thumb:

- Maximize the use of concurrency
- Minimize total completion time
- Minimize interaction among processes

- Often, task decomposition and mapping can result in conflicting goals.
- Degree of concurrency is affected by decomposition choice, but mapping affects how much of the concurrency can be efficiently utilized

Remember that if you throw all of these rules into a big object function, you won't have some perfect direction. You'll reach a point, where going in one direction reduces the benefit in one place, and going the other reduces the benefit in somewhere else. You will always have to assess trade-off. Do you want concurrency, or do you want synchronization overhead? It all depends on the architecture and the details of the program you are optimizing.

# 7 Decomposition Techniques

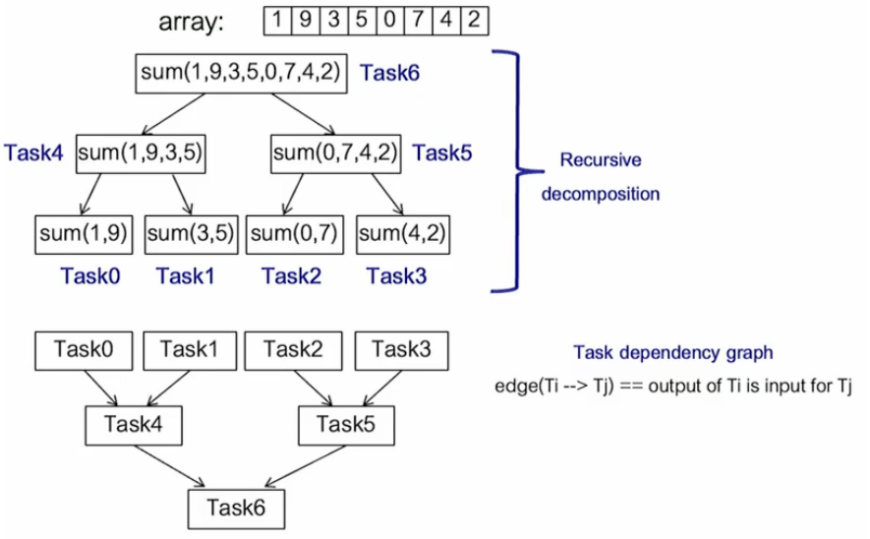
We have recursive decomposition and data decomposition.

Recursive decomposition is suited for divide and conquer algorithms, which primarily decomposes tasks. Data decomposition partitions the data to include task decomposition.

## 7.1 Recursive (Tree)

For example, for Mergesort: for every function call, I will spawn a task to handle each call. For example, I have a Mergesort routine which calls itself twice. What I'd do, is have two tasks to handle each of those calls. These calls would have tasks underneath them that would be spawning other tasks.

You don't have to use divide-and-conquer algorithms – say you want to add all elements to an array. Every time I have to add all elements, I'll just have to split.



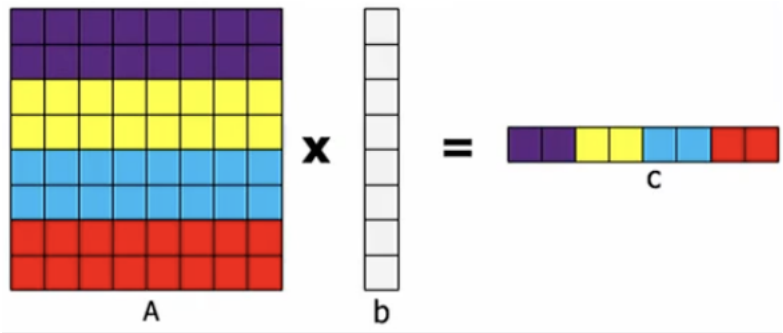

And there you go, that's a dependency graph for you. Is this a good idea?

## 7.2 Data (Shard / Stride, Iterative)

By first deciding how the data should be decomposed, that naturally put some constraints on how the tasks will be decomposed.

1. Partition the data
2. Induce task decomposition from the partitioned data

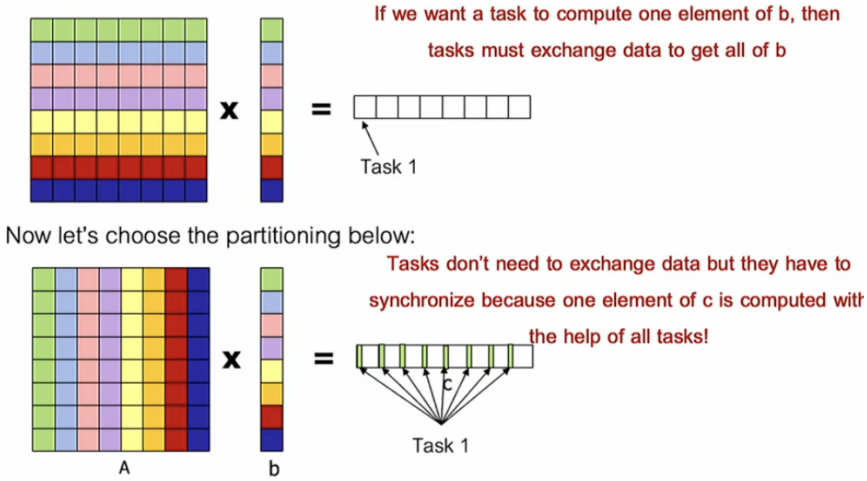
One way to do it – for example, for the matrix-vector multiplication, is to partition my **output** data:




Just the data partitioning by itself will not determine how my task will be decomposed. I still have some choices to make. Should I **shard** or **stride**? It depends on the hardware.

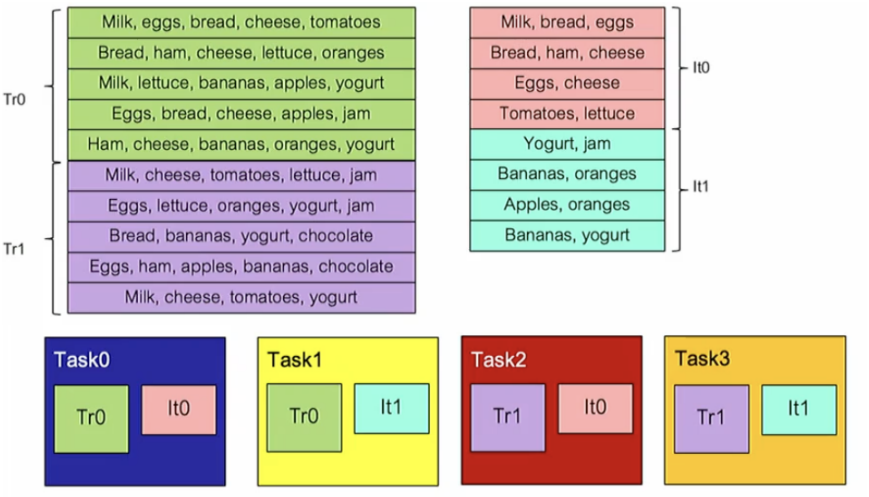
**You have to write both versions and profile them on a piece of hardware** (SciNet handles shading better, from lab 2).

Whether you partition input/output data, the experience should be very similar. The choice of how you split the data puts some constraints to how tasks can be assigned to the data.

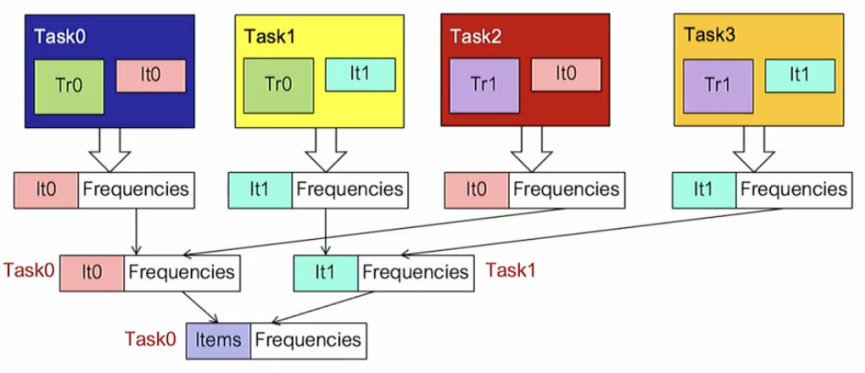



Communication overhead is quite high. Lots of modern hardware has ways to efficiently support concurrent operations. At least, addition is commutative.

We can also partition both input data and output data at the same time.




To calculate the milk-bread-eggs things, at one point, I need to know all of my transactions, but I can always split my work.




What is the trade-off for this technique? There's an extra step. What's the benefit? **I have eliminated resource contention.**

# 8 Mapping Techniques

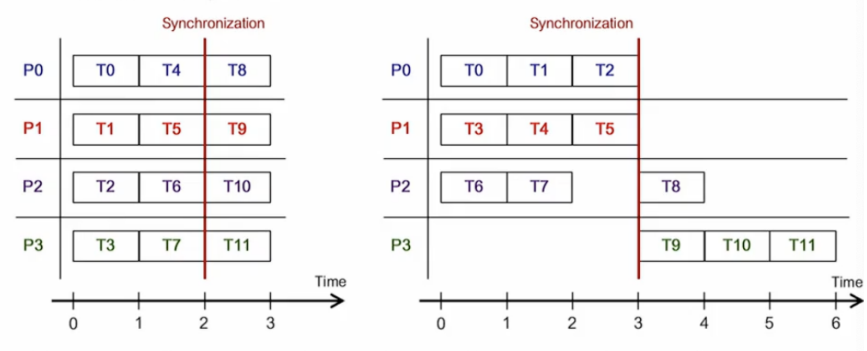
Why care about mapping the tasks? What if we randomly assign tasks to processes? Tasks are going to be given a certain amount of work, and we have a variable number of tasks. We want as much parallelism, but we want to reduce the overheads. For example:

1. Load imbalance: high variance across tasks – this results in a lot of idling
2. Inter-process communication: culprits of synchronization and data sharing

These two goals can be conflicting:

- Addressing Inter-process communication: Put interacting tasks on the same process. This can lead to load imbalance and in the extreme case, all tasks can be assigned to the same processor
- To optimize load balance, we break down tasks into fine-grained pieces to ensure good load balance. However, this can lead to a lot more interaction overheads

Warning: a balanced load may not necessarily mean no idling! Especially if there are dependencies. For example, we have tasks 0-11, and task 8-11 must wait until tasks 0-7 to finish.




## 8.1 Static Mapping

Decide beforehand, at compile time, which processor gets which task? This means I have no overhead for runtime. That is eliminated. If I know matrix-matrix multiplication, the size of my input data, then this is a good candidate for static mapping.

Yet, if task sizes are not known, this can potentially lead to severe load imbalances. One example – sparse matrices.

## 8.2 Dynamic Mapping

In the case where I need to do more sophisticated load balancing. If the task sizes are unknown, dynamic mappings are more effective than static times. This means that some runtime system must schedule the tasks. For large data, this may bring overhead.

### 8.2.1 Worker Pool

I have 8 tasks, and I have 4 threads. If thread no. 4 finishes early, it'll pick another task from the pool. This is the work pool model.

## 8.3 Interaction Overheads

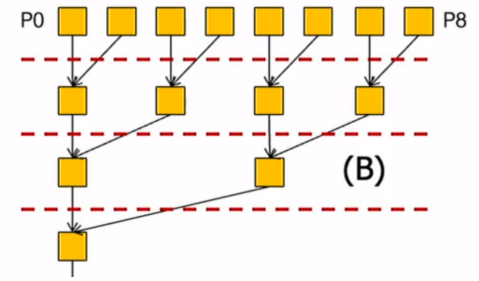
How do we contain interaction overheads?

1. Maximize data locality
    a. Don't have shared data all over the place. Group shared data in the same place. Minimize the volume of shared data and maximize cache use.
    b. Use local data to store intermediate results to decrease data exchange (I have done this).
2. Minimize the frequency of interactions
    a. Restructure the algorithm to access and use large chunks of shared data (amortize interaction cost by reducing frequency of interactions)
3. Accessing shared data concurrently can generate contention. For example, concurrent accesses to the same memory block, flooding a specific process with messages, and so on.
    a. Solution? Restructure the program to reorder accesses in a way that do not lead to contention
    b. Decentralize the shared data to eliminate the single point of contention. Want to calculate overall sums? Distribute the work such that it's shared across partial sum. And you can do this recursively or all at once (but recursively might work better due to its tree structure). No wonder dot products are faster.
4. Process may idle waiting for shared data, so instead, do useful computations while waiting
    a. Initiate an interaction earlier than needed, so it's ready when needed
    b. Grab more tasks in advance, before current task is completed.
    c. This may be supported in software (Compiler, OS) or hardware, but it is harder to implement with shared memory models. This applies more to distributed and GPU architectures.
5. Replicating data ⇒ no interaction overheads
    a. Beneficial if shared data is read-only

b. *Shared-address space paradigm: cache local copies of the data*

c. *Message-passing paradigm: replicate data to eliminate data transfer overheads*

This is the reduction tree. For example, a list $\sum$ might do this.




# 9 Parallel Algorithm Models

**Model** = **1 decomposition type + 1 mapping type + strategies to minimize interactions**

Commonly used parallel algorithm models:

- Data parallel model
- Work pool model
- Controller worker model
  - Names have changed!
  - Master ⟶ Main/primary/principal/**Controller**
  - Slave ⟶ **Worker**/secondary/agent

## 9.1 Data Parallel Model

- Decomposition: typically static and uniform data partitioning.
- Mapping: mostly static

Same operations on different data items, AKA data parallelism

How can we optimize it?

- Choose a locality-preserving decomposition
- Overlap computation with interaction

This model scales really well with problem size by adding more processes.

## 9.2 Work Pool Model

Tasks are taken by processes from a common pool of work.

- Decomposition: depends as we don't know the input.
- Mapping: dynamic
  - Any task can be performed by any process

Strategies for reducing interactions?

- Adjust granularity (task size) – trade-off between load balance and overhead of accessing work pool
  - Better load balance with smaller task sizes, but higher overhead as more accesses are required
- Overlap computation and interaction: worker may extract the next task while computing

## 9.3 Controller-Worker Model

Commonly used in distributed parallel architectures. Very common on shared-memory systems. This model is way more common in distributed memory systems.

The **controller** acts as the runtime to push out tasks and input to nodes. It coordinates communicating and sending data. It will mostly not do work on the actual problem.

- Decomposition: depends (data, recursive)
- Mapping: often dynamic

How do we reduce interactions?

- Choose granularity carefully so that controller does not become a bottleneck
- Overlap computation on workers with controllers generating further tasks

# 10 Parallel Performance Model / Performance Metrics

Parallel execution over $N$ processes rarely results in $N$-fold performance gain. Why?

- Overhead of
  - Spawning threads
  - Inter-process communication
  - Idling
  - Excess computation, computation done just to get around communication that a non-parallel program wouldn't have to do

So, what are our performance metrics?

- Execution time
- Speedup
- Efficiency
  - Make good use of your resources, especially with compute clusters! Don't let nodes go idle.
- Example performance plots
  - That's how you represent it. It's really easy to misinterpret or fool experimental results.

How would you implement something?

- Implement a naïve example
- Do every optimization possible, and test the optimized version against the naïve model

## 10.1 Parallel Speedup

$$S = \text{Speedup} = \frac{T_{\text{serial}}}{T_{\text{parallel}}}$$

For example, summing all elements of an array, assuming we have an infinite no. of processors:

$$\begin{aligned} T_p &= \Theta\left(\log(n)\right) \\ T_{\text{serial}} &= \Theta(n) \\ S &= \Theta\left(\frac{n}{\log(n)}\right) \end{aligned}$$

### 10.1.1 Speedup Considerations

Speedup is calculated relative to the best serial implementation of the algorithm. First, improve your serial implementation, use that as the baseline, and optimize the improved serial implementation on $p$ processors. Bad serial results in bad parallel.

Maximum achievable speedup is **called linear speedup**. **This assumes no overhead.** For example:

- Each process takes $p$ times less than $T_{\text{serial}} \Rightarrow S = p$ = no. threads

Can we go over linear speedup? Totally, but…

### 10.1.2 Superlinear speedup

If $S > p$, we have **superlinear speedup.** But, if all processors spend less than $\frac{T_{\text{serial}}}{p}$, why not use 1 process to run the whole thing again?

Superlinear speedup only happens when the sequential algorithm is at some disadvantage to the parallel version, for instance:

- Data too large to fit in 1 processors cache
  - But when split up, it can fit into all of them

Example:

> If half of the data fits in one of the L1 caches and work can be divided between the processors, then the data only gets loaded once into the caches from memory.

## 10.2 Ways to do Speedup

If our old problem runs in two time blocks:

- $T_1$: time that cannot be enhanced
- $T_2$: time that can be enhanced
  - $T_2'$: time after enhancement, < $T_2$

Then:

- Old time: $T = T_1 + T_2$
- New time: $T' = T_1 + T_2'$

Then, the speedup overall would be

$$\frac{T}{T'}$$

## 10.3 Amdahl's Law / Maximum Speedup / Calculating Speedup

Puts a limit on how much speed you can get based on the fraction of the program that can be parallelized. **This is a great test question, familiarize yourselves!**

Amdahl's law states that, given:

- $T_1$ = fraction of work done sequentially (Amdahl fraction)
- $T_2 = 1 - T_1$, fraction parallelizable
- $p$ = processor count

Then:

$$\text{Speedup}\left(\text{Program}\right) = \frac{T}{T'} \leq \frac{1}{T_1 + \overbrace{\frac{(1 - T_1)}{p}}^{\text{This is } T_2}} \leq \frac{1}{T_1}$$

This means that even if the parallel part speeds up performance perfectly (moreover infinitely), our performance is still limited by the sequential part.

*If your speedup is limited by the serial parts of your program, you want to minimize the serial parts of your program.*

## 10.4 Efficiency

The fraction of time when **processes** are doing useful work, where

$$E = \frac{S}{p} = \frac{\text{Speed up}}{\text{Processor count}} \in [0,\ 1]$$

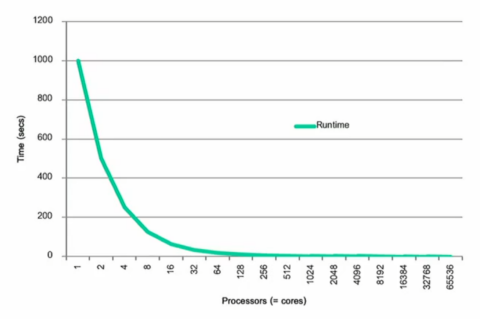
Example: summing an array of length $n$ over $n$ processes:

$$E = \frac{\overbrace{\Theta\left(\frac{n}{\log(n)}\right)}^{S = \frac{T_S}{T_p} = \frac{\text{Serial execution time}}{\text{Parallel execution time}}}}{\underbrace{n}_{\text{processor count}}}$$

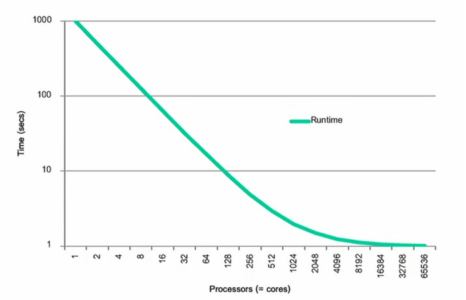
$$E = \Theta\left(\frac{1}{\log(n)}\right)$$

Shared memory should have good efficiency… breaking that might slow things down.

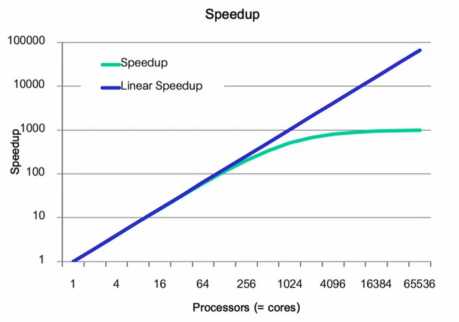
## 10.5 Reporting Running Time / Presenting Graphs



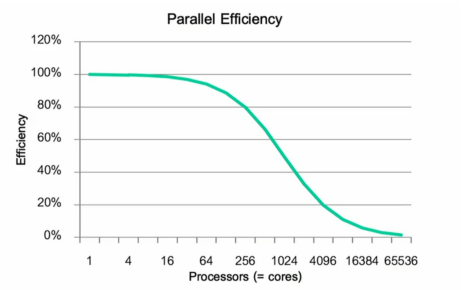

Time taken vs. processor count. Once you get past 32 processors, it becomes hard to read. What could I do? Read the log scale. This gives you a much better picture of what's going on.




Most of the time, you'll never go over 64 nodes. Using log-scale is one way to represent data better. This plot shows the total end-to-end runtime. You can see that there is some speedup, but here's an explicit plot that shows the actual speedup.



*Linear speedup* is the theoretical best case. The actual, green speedup is *S*, and some point you'll hit an asymptote. Why? **Amdahl's law puts an upper bound on how much speedup you can get**.




Why does efficiency go down as the no. of processors use go up? You can consider any point, the efficiency.

If you see the speedup graph, the rate of speedup goes down (first derivative) with more processors until you hit an asymptote.

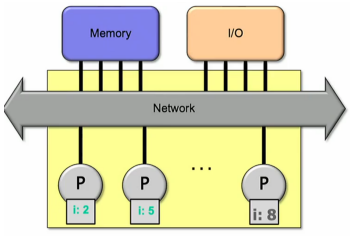
Or intuitively, the more processors you have, the more you need to coordinate different processes.

## 10.6 Rules of Thumb for doing Experiments

Keep these in mind!

1. Always work with the same data-types. Do not replace `long` with `int` for any reason.
2. Always optimize the serial algorithm first before using it as the baseline for speedups.
3. When optimizing parallel code, if you notice you could've optimized the serial, **optimize the serial.**
4. Report running times! Don't neglect this. Running time and speedups are both important, and you can't get the full picture with one but not both.

# 11 P-Threads




This is the shared memory architecture. These cores work independently in parallel. If I have multiple of these nodes, I would have distributed memory.

A common programming model is using `Pthreads`. The OS provides this interface so I can use that. What the programmer needs control is, how do I create the parallelism? And in what order will the parallel instructions occur? Also, about data – making the line between private and shared data. And then we have to deal with synchronization.

## 11.1 Programming Model: Shared Memory

A program is a collection of threads of control, where threads can be created mid-execution. A program can spawn threads, and the threads must be scheduled by your operating system. As a programmer, you can't really control the scheduler.

What is the difference between a thread and a process?

| Process | Threads |
|---|---|
| Can't read data from other processes | Can read data from other threads |
| Expensive to create, as it has a lot of things to track | Cheaper to spawn and context switch to (has its own registers, and all of its memory is shared) |

Each thread has a set of **private variables,** for example, local stack variables.

| Local to a thread | Shared across all threads |
|---|---|
| Stack | Heap<br>Static variables (threads communicate implicitly by writing and reading to/from them) |

## 11.2 POSIX Threads

POSIX:

- Portable
- Operating
- System
- Interface

PThreads: what POSIX exposes, the POSIX threading interface

- System calls to create and sync threads
- Should be uniform across UNIX-like OS
- No explicit support for communication because shared memory is implicit

## 11.3 Creating PThreads

Signature:

```
1 int pthread_create(pthread_t *thread_id, const
    pthread_attr_t *attr, void *(*)(void *) f, void* arg)
```

Why are we using void pointers? Because C doesn't have generics. Blame them.

What are each of the arguments?

- `thread_id` is the thread ID or handle, used to halt or wait

- `thread_attribute` holds various attributes, like the minimum stack size or priority. For defaults, pass in `NULL`
- `f` is the function to run, which takes and returns a `void*`
- `fun_arg` is the argument you pass into `f` when it starts. It'll typically be a struct.

**You must link the `pthreads` library!** Compile using `gcc -lpthread`. Every OS implements PThreads differently. You must link it to your PThreads library.

## 11.4 Joining / Awaiting

Here's a typical pattern. `pthread_join` waits until the thread passed in stops executing, and the second argument just collects its return value (hence it's a pointer).

```
1 int main() {
2   pthread_t threads[NUM_THR];
3   for(int i = 0; i < NUM_THR; i++)
4     pthread_create(&threads[i], NULL, f, NULL);
5   for(int i = 0; i < NUM_THR; i++)
6     pthread_join(threads[i], NULL);
7 }
```

## 11.5 Synchronization

If you spawn a bunch of threads, they can run in *any* interleaving order. Arbitrary interleaving of thread executions can have unexpected consequences, so we need a way to restrict the possible interleaving of execution. The app is oblivious to scheduling, so we **cannot** know when we lose control of the CPU where it lets another thread or process run.

**Synchronization** gives us control over who can access what.

### 11.5.1 Race Conditions

This is what happens when 2 or more concurrent threads manipulate a shared resource without any synchronization. The outcome depends on the order in which accesses take place, which is unpredictable.

We need to ensure that only one thread can manipulate the shared resource at a time.

## 11.6 Mutual Exclusion / Critical Section

Given a set of $n$ threads $T_0, \ldots, T_{n-1}$ and a set of resources shared between threads, and a segment of code which accesses the shared resources, called the **critical section.**

We want to ensure:

- Only one thread at a time can access the critical section
- All other threads must wait at entry
- When thread leaves CS, another can enter

## 11.7 Mutex Locks

Typically associated to a resource, to ensure one access at a time, to that resource. It ensures mutual exclusion to a critical section. For Mutexes, a thread goes to sleep if the mutex is busy. Or you can spinlock, though most programs don't use it. Spinlocks can be faster as sleeping is expensive, so spin if the lock is short.

We can **acquire** a lock or **release** the lock.