CSC367 Notes

Parallel Programming

Last updated May 21, 2024

Contents

1 Introduction

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. Capacitance is the ability of a system to store an electric charge.

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 $18\times$ 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.

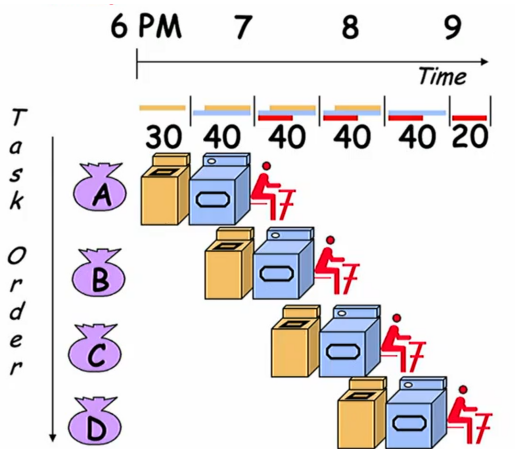
3.3 Cold and Warm Starts

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_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):
            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?

```python
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.

```python
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
```

```
6               # read block B[k, j] into fast memory
7               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
8           # 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 – 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 4×4 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 ⇒ 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 communication 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 (Threads, 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

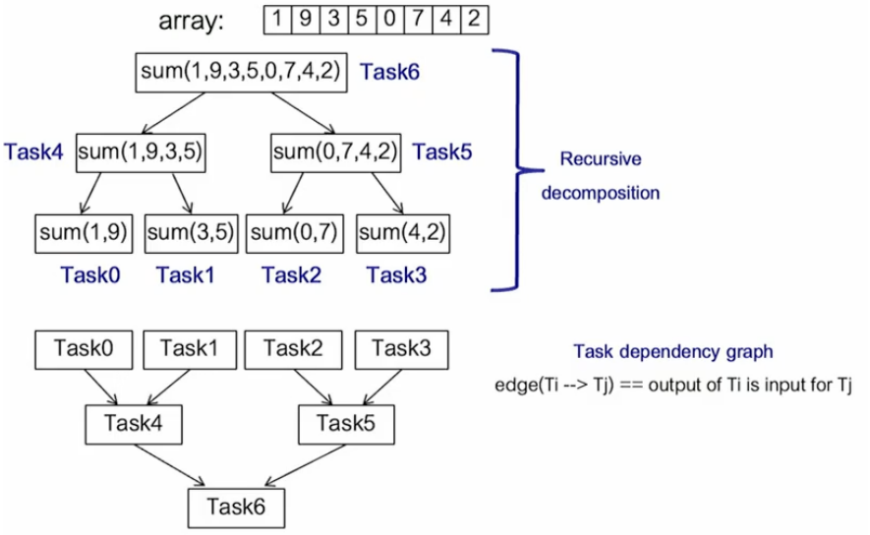
We have recursive decomposition and data decomposition.

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?

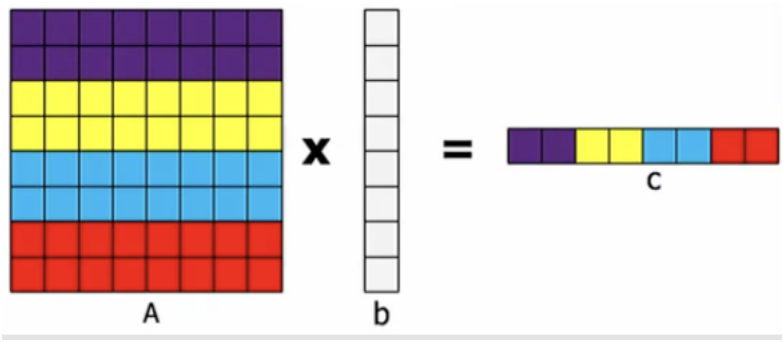
7.2 Data (Shard / Stride, Iterative)

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

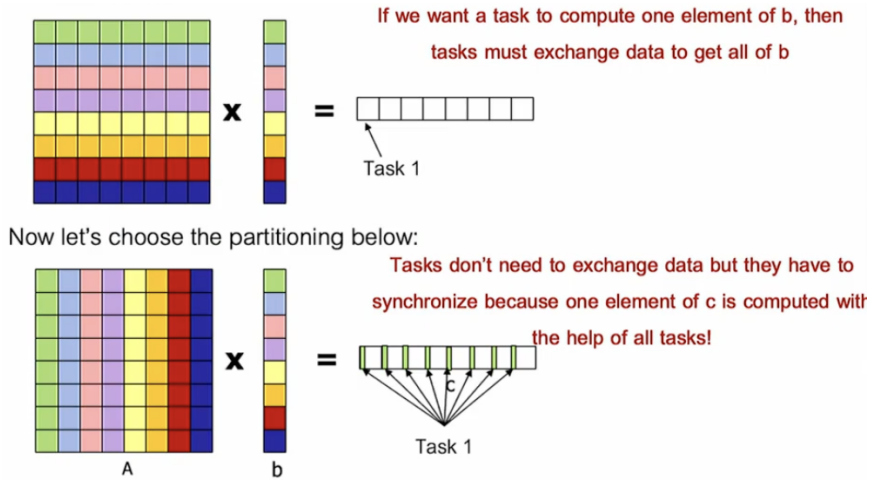
2. Induce task decomposition from the partitioned data

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



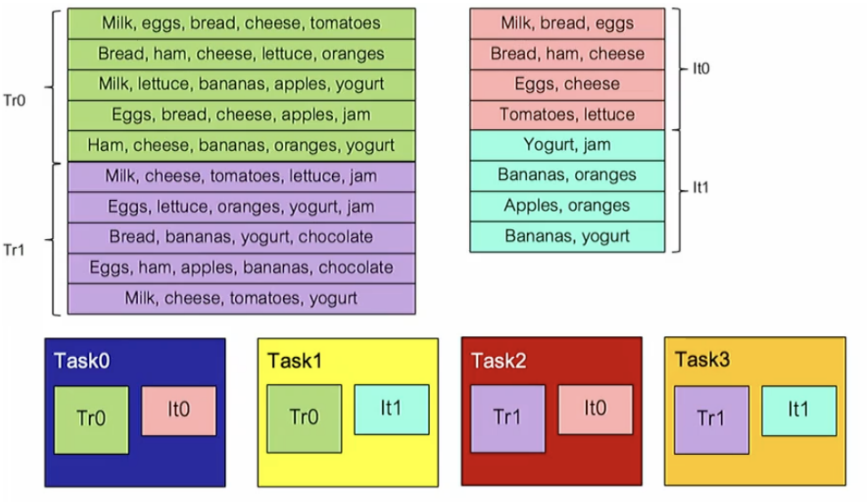
Just the data partitioning by itself will not determine how my task will be decomposed. I still have some choices to make. Should I **shard** or **stride**? It depends on the hardware. **You have to write both versions and profile them on a piece of hardware** (SciNet handles shading better, from lab 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.

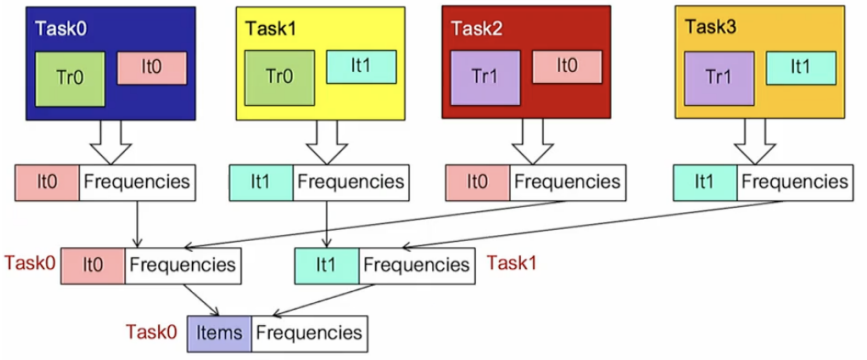


Communication overhead is quite high. Lots of modern hardware has ways to efficiently support concurrent operations. At least, addition is commutative.

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:

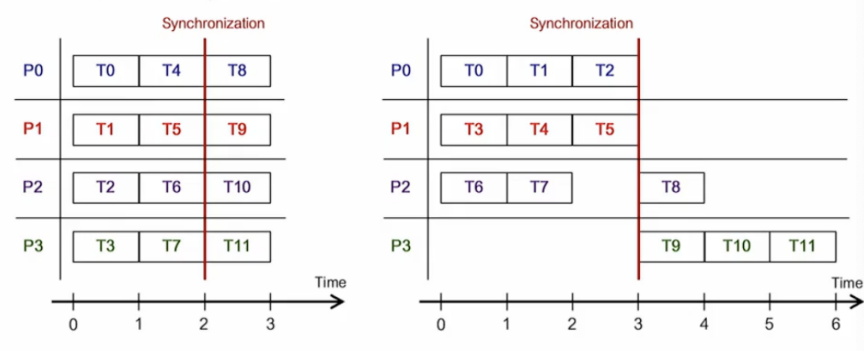
1. Load imbalance: high variance across tasks – this results in a lot of idling

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.

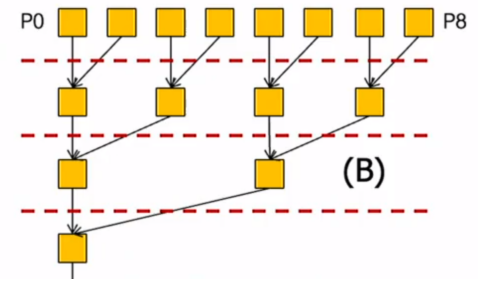
8.3 Interaction Overheads

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 is 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 result 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:

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

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 Super linear speedup

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

Super linear 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:

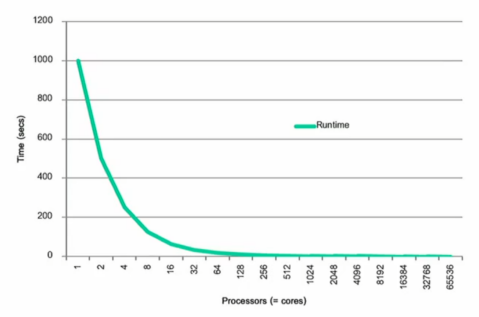
$$S = \frac{T_S}{T_p} = \frac{\text{Serial execution time}}{\text{Parallel execution time}}$$

$$E = \frac{\overbrace{\Theta\left(\frac{n}{\log(n)}\right)}}{\underbrace{n}_{\text{processor count}}}$$

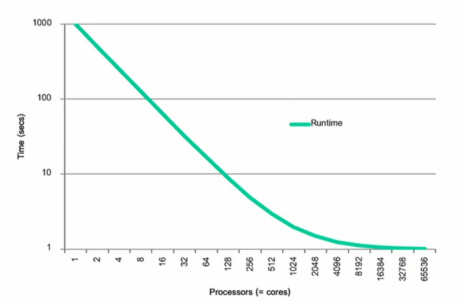
$$E = \Theta\left(\frac{1}{\log(n)}\right)$$

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

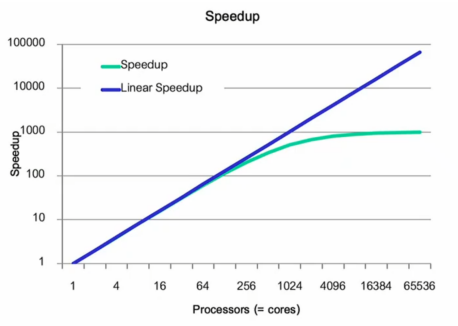
10.5 Reporting Running Time / Presenting Graphs



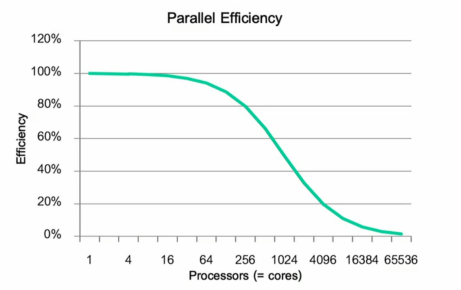
Time taken vs. processor count. Once you get past 32 processors, it becomes hard to read. What could I do? Read the log scale. This gives you a much better picture of what's going on.



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.

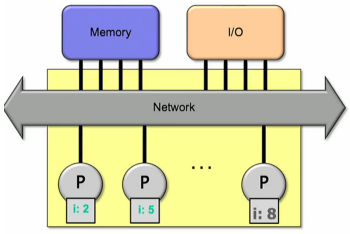
Or intuitively, the more processors you have, the more you need to coordinate different processes.

10.6 Rules of Thumb for doing Experiments

Keep these in mind!

1. Always work with the same datatypes. 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 will 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 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:

```
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).

```
int main() {
  pthread_t threads[NUM_THR];
  for(int i = 0; i < NUM_THR; i++)
    pthread_create(&threads[i], NULL, f, NULL);
  for(int i = 0; i < NUM_THR; i++)
    pthread_join(threads[i], NULL);
}
```

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.

Any resource that can be potentially written to needs to be locked. Also, make sure to unlock everything that gets locked, as that would result in a deadlock.

Be careful with fine-grained locking. It is very easy to mess up. How do I avoid this? Just understand the program semantics (meaning). As long as you understand the meaning of your code, work carefully and you can avoid these types of issues.

There are some solutions to fine-grained locking:

- Restructure code
- Just lock larger sections

Locks are not free.

11.8 Deadlock

Thread 1 waits for thread 2 waiting for thread 1 and so on. That creates a deadlock. **To break that, just always acquire locks in the same order.**

11.9 Overheads of Locking

When threads access the same locks, this results in lock contention. If lots of threads are contending for the same lock, it impacts performance as threads can't do work when that happens. Even with no contention, acquiring a lock has overheads. You better localize your computations to avoid this issue, and later in the future combine stuff. That's why reduction trees are helpful.

11.10 Thread Barriers

Threads that reach a barrier stop until all threads have reached it as well. If execution has stages, barrier ensures that data needed from a previous stage is ready. Use this if you have a dependence on the final result.

POSIX has a built-in barrier implementation.

12 OpenMP

OpenMP helps get around some pitfalls (overhead of thread creation, data race bugs, and deadlocks), but it doesn't eliminate all of them.

OpenMP stands for open specification for multi-processing. It is a collection of preprocessor directives. The output of the preprocessor is just more source code, which is the level that OpenMP operates at. When you're using OpenMP, the preprocessor is spitting out C code that has PThreads.

OpenMP is a portable, threaded, shared-memory programming specification with light syntax. It allows programmers to separate a program into serial regions and parallel regions, rather than P concurrently executing threads. It hides stack management and provides synchronization constructs.

It won't parallelize automatically, guarantee speedup, or provide freedom from data races.

Having a for loop inside an already parallel region won't make it parallelized.

12.1 OpenMP Conceptual Model

- Start with a single start thread (master thread)
- You have a parallel region (created by the `omp parallel`pragma)

- It forks, creating a group of worker threads. Everything in a parallel block is executed by every thread.
- Joins: at the end of the parallel block, all threads synchronize. Execution continues with only the initial (master) threads. You can opt out of this.

Threads can do the exact same work, share the same tasks, or perform distinct tasks in parallel.

12.2 Shared Memory

All threads share the same address space. OpenMP provides the ability to declare variables private or shared on a parallel block.

You must include the OpenMP headers in the files as it gives us some functions to work with. We must also link the OpenMP library (like the PThreads library).

You can debug with GDB and Val grind or using CLion's debugger.

12.3 Spawning Threads

The default number of threads to be spawned is stored in the environment variable. That will usually be the number of cores in your machine. Hyperthreading is when you have multiple threads in a core – it refers to two threads that don't actually exist. A lot of times, people will just disable hyperthreading.

There are some very commonly used OpenMP functions grouped together, which is included when you include the `omp` header file.

- `int omp_get_num_threads();`
  - This is important. Gets the number of threads when the closest enclosing parallel region (the inner-most block I am in).
  - Better call this in the parallel region; it should return the same thing amongst all threads so there is no point calling this beforehand.

- `void omp_set_num_threads(int n);`
- `int omp_get_max_threads();`
  - Max no. of threads that can be made.
- `int omp_get_thread_num();`
  - Thread ID in a group of threads. Each thread has a unique identifier. This helps us differentiate. This is the most important
- `int omp_get_num_procs();`
  - Processor count
- `int omp_in_parallel();`
  - Non-zero if called within a parallel region.

12.4 Private, Shared, and Variable Semantics

If I have shared variable, such as when I write `shared(v2, v3)`, what that means is that inside this parallel region, every single one of these threads will share a version of `v2` and `v3`, so that when one thread updates `v2`, it has to synchronize with all the other threads to ensure they have the same copy.

- `shared`: all threads share the same copy of variables
  - This is by default! If you don't say anything about the variables, all of them will be shared going into the parallel region.
- **`private`**: variables are local to each thread
  - With **`private`**, even if a variable has some value before the parallel region, if you set it to **`private`** it will be uninitialized when you hit the parallel region.
  - You could *just* use a local variable, but your compiler will release that variable when you leave that scope.

  - The induction variable of a for loop will always be private

- `firstprivate`: Like private, but value of each copy is initialized to the value before the parallel directive

What happens when I exit the parallel region?

- Shared keeps the value you set to it last
- For private, when you leave the parallel region, that variable will revert (this is NOT the same as your typical scoping)

12.4.1 What should be private, what should be shared?

Sharing results in synchronization overheads. Sometimes, you can't avoid sharing. Another complication is that by default, everything will be shared – that's a decision that was made and now my IDE just warns me if I don't specify a default myself. Shared is tricky if the data is not read-only, so there is a dramatic impact on correctness if you get this wrong.

You really have to look at your program and understand what is going on. There are actually some rules of thumb: SET THE DEFAULT:

- `default(shared)` – by default
  - Variables declared outside a parallel block become shared when the parallel region starts by default
- `default(none)` – you must exactly specify how every single variable used in parallel region should be handled, otherwise your program will not compile. Just **always do this**
  - Variables declared within a parallel block are implicitly private

12.5 Reductions

Instead of having to implement my own reduction tree using PThreads, I can just use the reduction command inside OMP, which implements the same thing for you. You

can leave your parallel region exactly as it was before. For example:

```
int s = 0;
#pragma omp parallel reduction(+:s) num_threads(8)
{
    s += ...
}
// can use s now, being the sum
```

Many operators are supported: `+`, `*`, `-`, `&`, `|`, `^`, `&&`, `||`

How does it work? For the specific example above: each thread gets a private copy of `s`, and OMP has a way to efficiently handle the shared synchronization overhead of updating at the end.

Reduction can do a lot of things for you.

12.6 The For Directive

For directive does the chunk calculation for you. In the parallel region, the `omp for` clause will divide up the work evenly.

Loops tend to be the most time-heavy part of your program. The loop often runs many instructions at a time. You can first look at your program to see where your time is being spent on the most. That will usually be a loop. Figure it out, start there, and figure out how to make the iterations of the loop independent. Once you've figured out how to make the loop body execute independently, you'll figure out how to make it run in parallel.

CONSTRAINTS

- Induction variable must be incremented by an integer amount
- Induction variable must be integer
- Loop guard must be a $<$, $>$, $\leq$, or $\geq$ expression
- No `break` instructions

Your program will not compile if these constraints are broken.

12.6.1 Implicit Barriers

```
// spawn threads
#pragma omp parallel {
  #pragma omp for
  for(...) {

  }
  // IMPLICIT BARRIER
  #pragma omp for
  for(...){

  }

}
```

To remove the implicit barrier:

```
// spawn threads
#pragma omp parallel {
  #pragma omp for nowait
  for(...) {

  }
  // IMPLICIT BARRIER
  #pragma omp for
  for(...){

  }

}
```

12.7 Scheduling Work To Threads

The `schedule` clause gives ways to assign iterations to threads:

```
schedule(scheduling_class[,parameter])
```

Scheduling classes include:

- **static**
- dynamic
- guided
- runtime

```
#pragma omp for schedule(...)
```

Static scheduling: iterations of loops are divided to each thread ahead of time.

Dynamic scheduling: split iterations into chunks, assign new chunk to thread only when idle. Use this if you're unsure how long each loop iteration will take. Still vulnerable to load imbalance, which could cause idling for some threads

Guided scheduling: Start with big chunks but reduce size as computation progresses. `Schedule(guided[, chunksize])`. Parameter `chunksize` (iterations) specifies the minimum chunk size, default 1.

Runtime scheduling: Let an environment variable decide this, which has to be picked up during runtime. Environment variable `OMP_SCHEDULE` determines class and chunk size. If no schedule type, runtime will auto-schedule (choose the best one)

12.8 Nested Parallelism

A parallel region may have additional parallel regions inside of it. You can get quite complicated of what you want to do in OMP.

12.9 Sections

```
1 #pragma omp parallel {
2   #pragma omp sections [clauses] {
3     #pragma omp section {
4       // task 1
5     }
6     #pragma omp section {
7       // task 2
```

```
    }
  }
  // IMPLICIT BARRIER
}
```

If I need some functions that are independent of each other to run in parallel, I just do this. In each section, it will have its own thread. You can apply the normal clauses to sections (private, reduction, `nowait`)

12.10 OMP Tasks

`#pragma omp task/s` is another directive similar to the section directive. OMP tasks do not have an implicit barrier.

```
int fib(int n){
  int x,y;
  if(n<2) return n;
  #pragma omp task shared(x)
  x = fib(n-1);
  #pragma omp task shared(y)
  y = fib(n-2);
  // barrier
  #pragma omp taskwait
  return (x+y);
}

int main() {
  int n = 9999;
  // parallel regions apply inside called functions
  #pragma omp parallel
  {
    // prevents multiple threads from calling this, the
       thread that hits this first
    // note that the master clause only ensures the master
       thread touches it
    #pragma omp single
    fib(n);
    // single creates an implicit barrier. Master does not
  }
}
```

Because the parallel clause is so common, you can merge it with sections like with `for`.

12.11 Barriers

Wait for all threads spawned by the closest enclosing `parallel` directive

```
#pragma omp parallel
{
  // all threads stop here and wait until barrier clears
  #pragma omp barrier
}
```

12.12 Critical Sections

Critical region where threads must serialize to avoid race conditions. For instance:

`#pragma omp critical [(nameofcritical region)]`

Simple example:

```
int data = 100;
#pragma omp parallel sections
{
  #pragma omp section
  {
    #pragma omp critical (datacrit)
    {
      data += 42;
    }
  }
  #pragma omp section
  {
    #pragma omp critical (datacrit)
    {
      data += 5;
    }
  }
}
```

12.13 Atomic Construct

When the critical section is just an update to a single memory location. The instruction after the `atomic` directive is executed atomically.

```
1 #pragma omp atomic [read|write|update|capture]
2   op = expr;
```

Only simple operations, expression has to be sample. This can be translated to a critical directive.

Most hardware has support for this kind of operation. The reason why critical sections are more expensive is because the are more general and you can lock a more arbitrary section of your code.

12.14 Explicit Locking

Just like `pthreads`, OMP lets you create and destroy locks like `pthreads`. It's a 1-1 mapping.

12.15 Timing and Profiling

`omp_get_wtime();` wall clock time in second. The time measured per thread, and there's no guarantee that two distinct threads measure the same time.

12.16 MSI Protocol, False Sharing and Getting Around It

Each cache line is labeled with a state:

- M: modified.
- S: other caches may be sharing this block
- I: Cache block is invalid

Before a cache modifies a location, the hardware first invalidates all other copies.

So how do we get around false sharing?

```
#include <omp.h>
static long num_steps = 100000; double step;
#define   PAD     8    // assume 64 byte L1 cache line size
#define NUM_THREADS 2
void main ()
{  int i, nthreads;  double pi, sum[NUM_THREADS][PAD];
   step = 1.0/(double) num_steps;
   omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {        int i, id,nthrds;
             double x;
             id = omp_get_thread_num();
             nthrds = omp_get_num_threads();
             if (id == 0)   nthreads = nthrds;
             for (i=id, sum[id]=0.0;i< num_steps; i=i+nthrds) {
                          x = (i+0.5)*step;
                          sum[id][0]+= 4.0/(1.0+x*x);
             }
     }
   for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i][0]* step;
}
```

threads	1st SPMD	1st SPMD padded
1	1.86	1.86
2	1.03	1.01
3	1.08	0.69
4	0.97	0.53

False sharing only impacts writes.

Or you can instead just use critical sections and have each thread track their own variables. RT about same

```
#include <omp.h>
static long num_steps = 100000;  double step;
#define NUM_THREADS 2
void main ()
{ int nthreads; double  pi=0.0;
  step = 1.0/(double) num_steps;
  omp_set_num_threads(NUM_THREADS);
  #pragma omp parallel
   {        int i, id, nthrds;
            double x, sum;
            id = omp_get_thread_num();
            nthrds = omp_get_num_threads();
            if (id == 0)   nthreads = nthrds;
            for (i=id, sum=0.0;i< num_steps; i=i+nthrd
                        x = (i+0.5)*step;
                        sum += 4.0/(1.0+x*x);
            }
           #pragma omp critical
                    pi += sum * step;
  }
}
```

12.17 Performance and Scalability Considerations

- Starting a parallel OpenMP region isn't free
  - Overheads involved results in performance impact
- Try to parallelize the outermost loop
- Parallel regions must have sufficient speedup to overcome overhead
- Overheads depend on machine, OS, and compiler
- Amdahl's law – the less of a program that runs parallel the less the speedup

13 Distributed Memory Architectures

Early distributed memory machines had to pass tasks through each node until it reaches its proper destination.

To have a large no. of different transfers at once, our must have a large no. of distinct wires. Networks are like streets:

- Link: street
- Switch: intersection
- Distances (hops): no. of blocks traveled
- Routing algorithm: travel plan

Properties

- Latency
- Bandwidth

13.1 Design Characteristics of a network

Topology: how things are connected

- Crossbar
- Ring
- 2D
- 3D
- …

Routing algorithm:

- In a 2D torus: all east-west then all north-south, avoids deadlock

Switching strategy:

- Circuit switching full path reserved for entire message

- Packet switching: message broken into separately routed packets

Flow control (in case of congestion)

13.2 Performance Properties

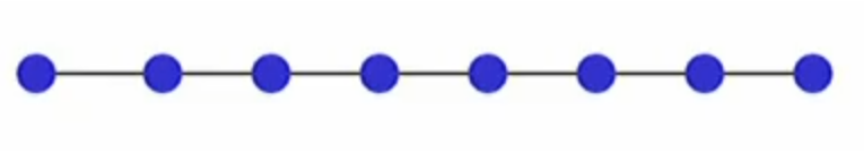
The maximum overall pairs of nodes of the shortest path between a given pair of nodes.

Latency is the delay between send and receive times. It tends to vary widely across architectures, and vendors often report hardware latencies. Application programmers care more about software latencies. It is key for programs with many small messages. It would be good to pack up messages into one, large messages but for some problems that isn't feasible.

Diameter: Max over all nodes, the shortest path between a given pair of nodes.

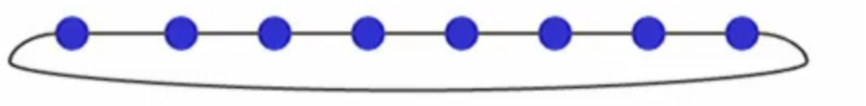
Bisection bandwidth: bandwidth across **smallest cut** that divides network into two equal halves. Bandwidth across "narrowest" part of the network. Bisection bandwidth is important for algorithms in which all processors need to communicate with all others.

13.3 Linear and Ring Topologies



Diameter: $n - 1$

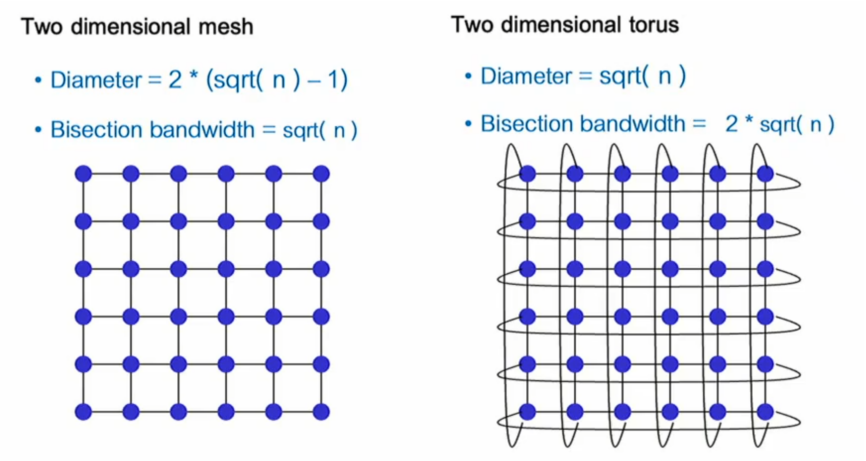
Bisection bandwidth: 1



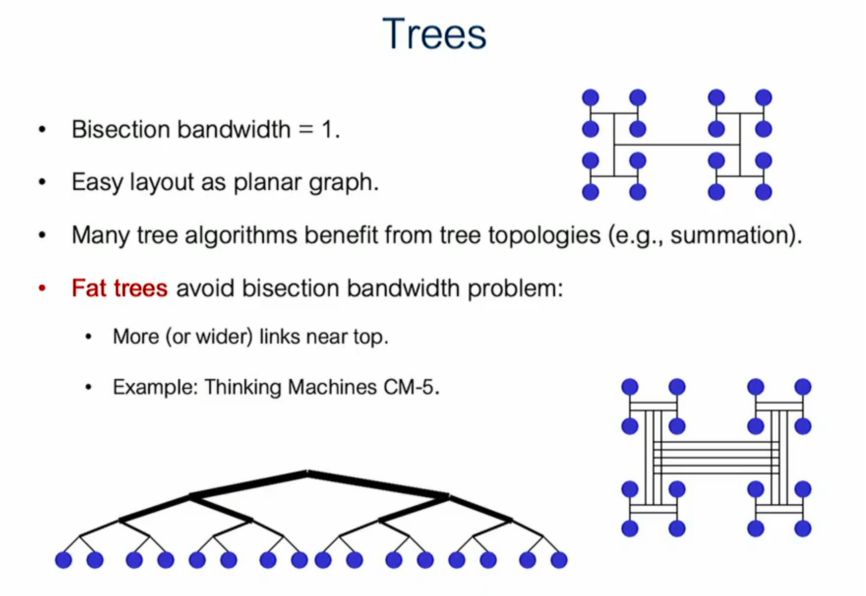
Diameter: $\frac{n}{2}$

Bisection bandwidth: 2

Natural for algorithms that work with 1D arrays



This generalizes to higher dimensions.



14 MPI

14.1 Message Passing Model

Single program multiple data. All processes execute same code, communicate via messages. It does support executing diff. programs on each host or process.

Benefit of message passing is that communication becomes very explicit. You are aware of its high bandwidth.

Views of message passing:

- Logical: each process is a separate entity within its own memory. It runs a separate instance; a process cannot access another's memory without explicit messages. Message passing implies a distributed address space
- Physical view: the underlying architecture could be either distributed or shared memory. Message passing can be simulated by copying or mapping data between different processes' memory space

You can use message passing on a multiple core/shared memory machine, but the message passing interface simulates a distributed environment on the shared memory machine.

14.2 Sending and Receiving Messages

Send and receive primitives:

```
1 Send(void *sendbuf, int nelems, int dest)
2 Receive(void *recvbuf, int nelems, int source)
```

Complexity lies in how the operations are carried out internally.

14.3 Blocking Operations

Only return from an operation when safe. Not always when message has been received, just guarantee semantics.

For example: blocking non-buffered send/receive

14.4 Blocking Non-buffered send/receive

Blocking means it will idle/stall on the send until it has confirmation that the transfer is complete.

Non-buffered means there is no intermediate store, i.e. the data will be transferred directly between nodes.

One scenario, it is possible that one program initiates a send while the other program waits for a receive. One program will request to send, then it has to wait for the receiver to reach the point in the program to say that it has received a message.

This whole time, the sender was idling.

Another possibility is that both programs reach their respective points of execution at the same time, so there is almost zero idling time, and the transfer can be initiated immediately.

Then, there's another case, where the sender is still doing some extra work, where it hasn't reached the send statement yet, but the receiver was way quicker and is waiting for the receive message. This results in some idle time. It is very likely, in a blocking case, where you will have this idling overhead.

It is also totally possible and likely that by mistake, you may introduce some deadlock. What causes a deadlock? The send is blocking. The send will not progress to ensure the receive has been gotten.

```
P0                          P1
---                         ---
send(&m1, 1, 1);            send(&m1, 1, 0);
recv(&m2, 1, 1);            recv(&m2, 1, 0);
```

What can you do? Swap the order? Not ideal. Lots of machines have hardware support that is useful for message passing. Most message passing platforms have additional hardware support for sending and receiving messages. They may support DMA (direct memory access) and asynchronous message transfer using network interface hardware.

- Network interfaces allow the transfer of messages from buffer memory to desired location without CPU intervention.
- Similarly, DMA allows copying of data from one memory location to another without CPU support

14.5 Blocking Buffered Send and Receive

Copy data to some buffer before sending. This lets you finish the send without getting confirmation from the destination node.

What is gotten rid of from the non-buffered implementation?

We got rid of the idling, but the extra copying has extra overhead. Also, we have potential problems with finite buffers, and deadlocks are still possible.

- 1. Potential problems with finite buffers
  - Example:

```
P0 (producer)                          P1 (consumer)
---                                    ---
for(i = 0; i < 1000000; i++){          for(i = 0; i < 1000000; i++){
   create_message(&m);                    recv(&m, 1, 0);
   send(&m, 1, 1);                        digest_message(&m);
}                                      }
```

- 2. Deadlocks still possible
  - Example:

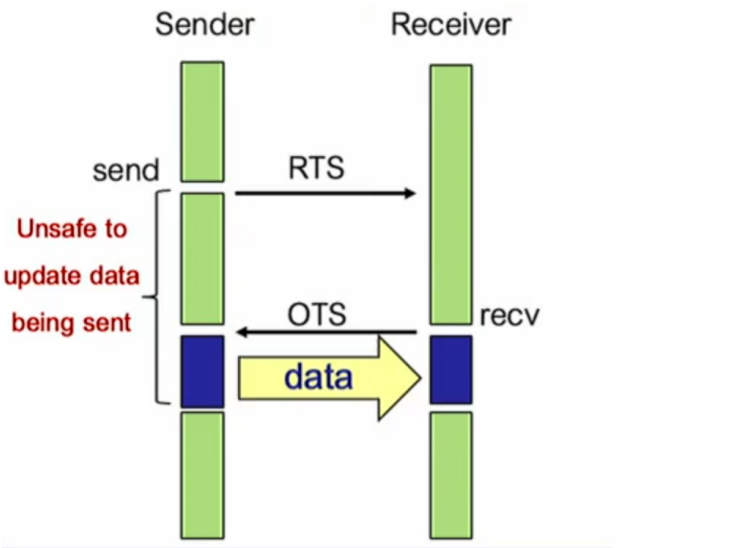
```
P0                                     P1
---                                    ---
recv(&m1, 1, 1);                       recv(&m1, 1, 0);
send(&m2, 1, 1);                       send(&m2, 1, 0);
```

- Solution is similar: break circular waits

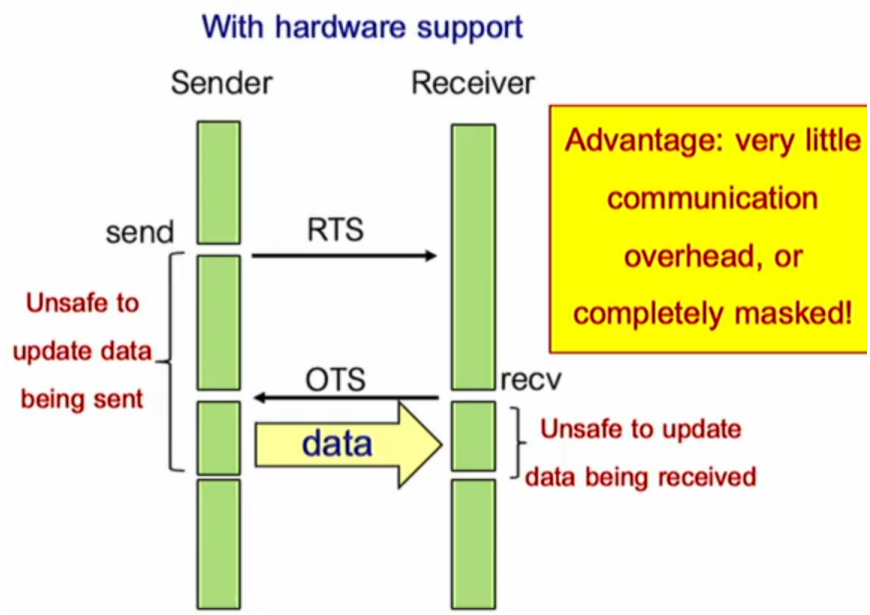
You can still have deadlock on receive. What you can do is have non-blocking operations. User is responsible to ensure that data is not changed until its safe. Send and keep the

program running. You have some extra parts of your interface to see if a message has been received or not.

14.6 Non-Blocking Non-buffered

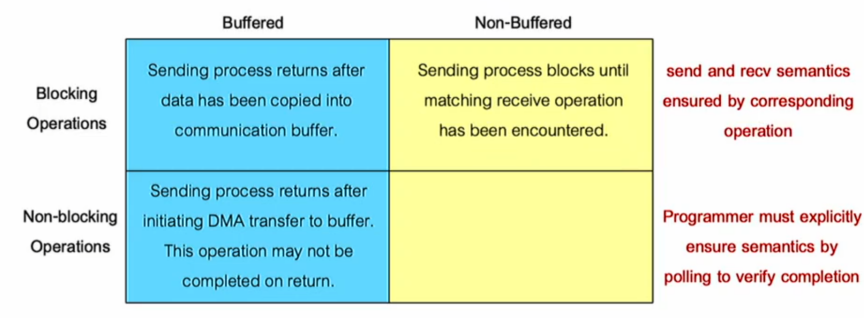


Don’t change your data until your data has been fully sent.



Advantage: very little communication overhead, or completely masked.

Summary of difference?



Be very careful when you initiate a send or receive. Carefully consider the implementation guarantees.

You have to understand what is implied by blocking. If you don't, you'll always run into deadlock.

When we transfer from blocking to non-blocking, blocking you have a lot more guarantees about the program (you don't have as much code to check that it is safe to transfer). When you transfer from a blocking call to a non-blocking call, you have more checks you have to do, and more code associated with that.

14.7 The Message Passing Interface

MPI is a standard for the message passing model. It is the standard library for message passing. There is a standard that is considered fully correct. This is one flavor of a message-passing model.

MPI_Init: initialize MPI environment

MPI_Finalize: terminate the MPI environment

MPI_Comm_size: get number of processes

MPI_Comm_rank: get the process ID of the caller

MPI_Send: send message

MPI_Recv: receive message

Commands:

- `MPI_init`: only called once at the start by one thread to initialize MPI environment
- `MPI_finalize` called at the end to do cleanup and terminate the MPI environment. On success, it returns MPI_SUCCESS, otherwise error code. Once calls, no more MPI allowed, even `MPI_init`, otherwise it is UB

14.8 MPI Communication Domains

Most of the time, you will just use `MPI_COMM_WORLD`: it takes all of the nodes available for you and puts them in the same communication world. Every node you have allocated can talk to any other node.

You can have much other complicated types of worlds. Say you want to have a totally different grouping of nodes, and you never want the nodes to communicate with each other, you want to have one higher node that communicates only between the two groups, you can have very complicated hierarchical layouts between everything – that is all done through communication domains.

14.9 Flavors of Communication in MPI

Collective operations: All processes in the communicator or group have to participate! E.g.

- Barrier
- Broadcast
- Reduction
- Prefix sum
- Scatter or gather
- All-to-all

Point to point operations: A processor explicitly communicates with one (or at least, not all) processors

14.9.1 Point-to-Point Communication

We have blocking send-and-receive and non-blocking send-and-receive

These are the send and receive function headers:

```
1 int MPI_Send(void *buf, int count, MPI_Datatype datatype,
     int dest, int tag, MPI_Comm comm);
2
3 int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
     int source, int tag, MPI_Comm comm, MPI_Status *status);
```

Argument 4 represents the rank (no. / ID) of the process we are sending/receiving to/from.

Each message has a tag associated to distinguish it from other messages. Each message will also have a status that you can get about the receive function. You'll need to know how big the received message is, which is not given by default.

In the case where you are dynamically allocating memory to store the result, well, you don't know how much memory you're going to allocate until you know the full size of

the message.

If you look at the receive prototype, there is nothing saying the full size of the message. What you have to do, is to figure out how much you're actually receiving before making a buffer.

14.9.2 Implementations of These Functions

`MPI_Recv` is **always** blocking. What happens, is:

- First, you have to wait for everything to come in and be finalized
- Everything after `MPI_Recv`, you can safely do whatever you want with the buffer

`MPI_Send` can be implemented with two options:

- Option 1: Call the send and return only when the corresponding receive has fully completed. The totally blocking operation.
- Option 2: Copy message into buffer and returns, without waiting. It will not wait for acknowledgement that the receive has been fully completed. In both, the buffer can be safely reused right after `MPI_Send`

14.9.3 Deadlock Avoidance

This is deadlock prone for option 1 of `MPI_Send`:

```
1 if(rank==1){
2   MPI_send(a);
3   MPI_send(b);
4 } else {
5   MPI_Recv(b);
6   MPI_Recv(a);
7 }
```

Match the order of `send` and `recv` operations! Otherwise, this could lead to deadlock. Or you could just use option 2 for `MPI_Send`.

You also want to avoid circular chain of send/receive operations.

14.9.4 Send-Receive at the same time

We can combine `MPI_Send` and `MPI_Recv` primitives into `MPI_Sendrecv`:

```
1 int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype
      sendtype, int dest, int sendtag, void *recvbuf, int
    recvcount, MPI_Datatype recvtype, int source, int
    recvtag, MPI_Comm comm, MPI_Status *status);
```

This lets a program send and receive at the same time with very little issue. The one restriction being, send and receive buffers must be disjoint. If not, use `MPI_Sendrecv_replace`.

14.10 Overlap Communication with Computation

MPI provides non-blocking alternatives.

```
1 int MPI_ISend(void *buf, int count, MPI_Datatype datatype,
    int dest, int tag, MPI_Comm comm, MPI_Request *request);
2
3 int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
    int src, int tag, MPI_Comm comm, MPI_Request *request);
```

- `MPI_Isend` starts a send operation, and returns before the data is copied out to the buffer
- `MPI_Irecv` starts a `recv` operation, and returns before the data is received into the buffer
- This allocates a request object and returns a pointer to it in the request argument. You as the programmer must verify whether the send and receive has completed.
- A non-blocking operation can be matched with a corresponding blocking operation.

How do we check if a send and receive have completed? We can use `MPI_Test` and `MPI_Wait`.

You'll see that in the `ISend` and `IRecv`, there is this `MPI_Request *request` argument at the end. The `request` is what you use to poll whether the operation is complete. If you want to see if a certain operation is done, take the associated request pointer, and feed it into

- `MPI_Test` returns non-zero if completed. If completed, request is deallocated, and set to `MPI_REQUEST_NULL`
- `MPI_Wait`: blocks until request completes, then deallocates request and sets it to `MPI_REQUEST_NULL`
  - This makes our send/receive look like a blocking send/receive.

This helps us guarantee that data has been sent or received. If we can't verify that data has been sent or received, this violates correctness

14.11 Barriers

Exactly the same intuition as a `pthreads` barrier or an `OMP` barrier. Ensures all the nodes in the communicator wait until they hit *this* point `MPI_Barrier(comm)` in their executions.

Be careful of deadlocking! Do **not** do conditional barriers:

```
1 if(cond){ MPI_Barrier(MPI_COMM_WORLD); }
```

Another thing is that barrier is separate from any other non-blocking operations. It will not keep track of things for you. If you call a barrier, it will not guarantee that your asynchronous send/receive have completed. These might still be running despite the barrier.

14.12 Broadcast

One-to-all: send `buf` of source to all other processes in the group, into their buffer.

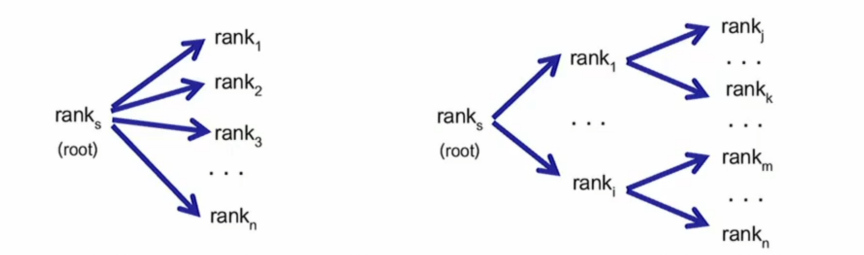
```
1 int MPI_Bcast(void *buf, int count, MPI_Datatype datatype,
    int source, MPI_Comm comm);
```

The last argument: `MPI_COMM_WORLD` to send to all processors

`MPI_Bcast` blocks until all processes make a matching `MPI_BCast` call. It is not required to block on all processes until the operation completes fully.

You could do the same thing with a send/receive by looping over all process numbers, but broadcast is optimized for this specific pattern. If you did all `recieves` for the root node, its buffer will be very contended.

What broadcast does, is that it creates a tree-like structure to reduce contention.



14.13 Reduction

You could also do reductions, like how `reduce` exists in OMP. You can do something similar in MPI. If you have data A, B, C, D in 4 different nodes, you can combine them together on the root node.

Reduce: combines the elements from the buffer of each process in the group and stores the result in `recvbuf` at the target receiver.

```
1  int MPI_Reduce(void *sendbuf, void *recvbuf, int count,
     MPI_Datatype datatype, MPI_Op op, int target, MPI_Comm
     comm);
```

The **target** node must have a valid receive buffer, but all the other nodes can have `NULL` as the `recvbuf`.

All processes must provide `send` and `recv` buffers of the same size and data type.

The built in operations include `MPI_SUM`, `MPI_MAX`, `MPI_MIN`, `MPI_PROD`, and so on.

14.14 All Reduce

It will do the same thing as `reduce`, but at the end, every node that calls it will receive the reduced information. All nodes must have the receive buffer in that case. Why would I call that?

- Calculating standard deviation: that will require you to calculate the mean first and give the mean to all nodes

14.15 Scatter

Take a contiguous array, take a chunk of the array, and send each chunk to a different node. For example:

$$\begin{bmatrix} A & B & C & D \end{bmatrix} \rightarrow \begin{bmatrix} A & - & - & - \\ B & - & - & - \\ C & - & - & - \\ D & - & - & - \end{bmatrix}$$

This is the function header:

```
1 int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype
     senddatatype, void *recvbuf, int recvcount, MPI_Datatype
      recvdatatype, int source, MPI_Comm comm);
```

This sends `recvcount` contiguous elements to each process. All of them receive the same amount.

- `Sendcount` is the number of elements sent to each individual process
- Send-related arguments are only applicable to the source, and are ignored
- `MPI_Scatterv` allows you to send a different number of items to each of the receiver

14.16 Gather

The inverse of scatter

$$\begin{bmatrix} A & B & C & D \end{bmatrix} \leftarrow \begin{bmatrix} A & - & - & - \\ B & - & - & - \\ C & - & - & - \\ D & - & - & - \end{bmatrix}$$

```
1 int MPI_Gather (void *sendbuf, int sendcount, MPI_Datatype
    senddatatype, void *recvbuf, int recvcount, MPI_Datatype
     recvdatatype, int target, MPI_Comm comm);
```

The calling convention for this is very similar.

There is a case where you may want to gather different amounts of data from different nodes. `MPI_Gatherv` might help you with that.

`MPI_Allgather` does what you think it does. Every node gets the same thing.

14.17 All-To-All

What does this do?

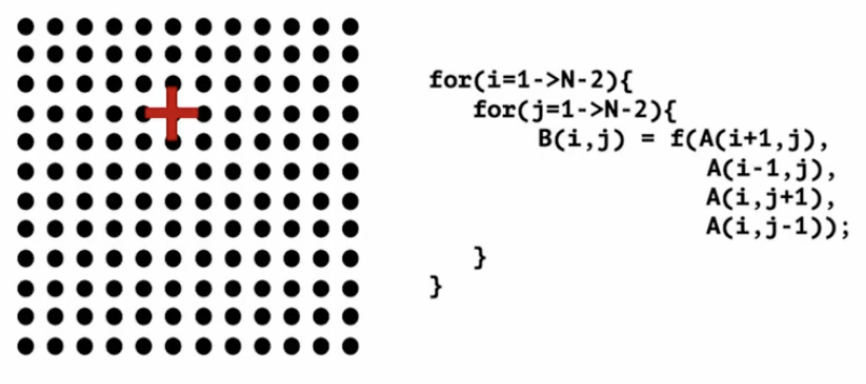
$$\begin{bmatrix} A_0 & A_1 & A_2 & A_3 \\ B_0 & B_1 & B_2 & B_3 \\ C_0 & C_1 & C_2 & C_3 \\ D_0 & D_1 & D_2 & D_3 \end{bmatrix} \rightarrow \begin{bmatrix} A_0 & B_0 & C_0 & D_0 \\ A_1 & B_1 & C_1 & D_1 \\ A_2 & B_2 & C_2 & D_2 \\ A_3 & B_3 & C_3 & D_3 \end{bmatrix}$$

Each process sends a different portion of `sendbuf` (`sendcount` contiguous items) to each other process, in order of rank, including itself. Notice how the matrix is **transposed** on the right side. If you need a solver that requires a transpose, just do all-to-all.

15 Strategies to Improve the Performance of simulations

15.1 Stencil Computation on a mess – Surface to Volume Ratio

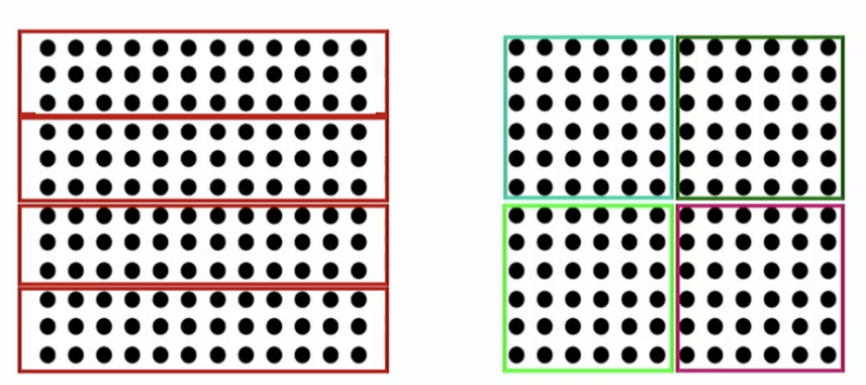
Say we have a mesh (rather table), and we need to step through the mesh one element at a time, requiring it to read neighbors (for example, doing a convolution).



```
for(i=1->N-2){
   for(j=1->N-2){
        B(i,j) = f(A(i+1,j),
                   A(i-1,j),
                   A(i,j+1),
                   A(i,j-1));
   }
}
```

When you go from this to a distributed setting, you need to figure out how you are going to divide the mesh across multiple nodes. No, you do not want to copy the entire mesh to all nodes, as the mesh could be extremely large.

Do you want to divide out the mesh by columns, by rows, or by a work pool (with square chunks)? The surface refers to all the edges and the volume refers to all the mesh points. A rule of thumb is you want to minimize the ratio of surface you have to volume you have, and it'll become clear why you want this.



The reason why you want to minimize surface, is because surface represents communication. Communication is expensive.

An edge crossing is where a communication edge is cut by the partitioning.

On the left-hand side, partitioning rows, how many edge crossings will I have? For a $n \times n$ grid with p partitions

- I have $p - 1$ horizontal lines
- n crossings per horiziontal line
- $n(p - 1)$ edge crossings

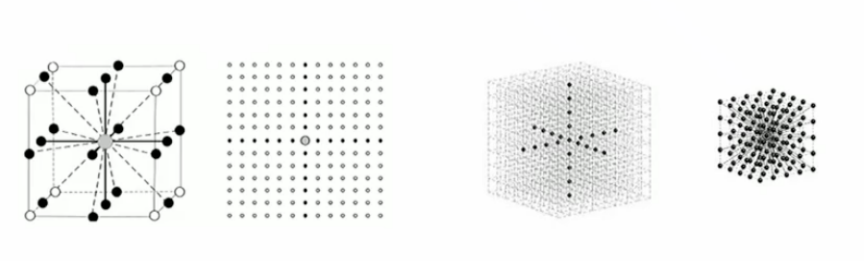
On the right-hand side, how many edge crossings are there?

$$2n\left(\sqrt{p} - 1\right)$$

There are 2 dividing lines.

Which has less communication? The RHS has less edge crossings, so less communication. A lot of the time, in distributed systems, because communication is more expensive, the RHS will have better performance. Yet, the LHS is harder to implement.

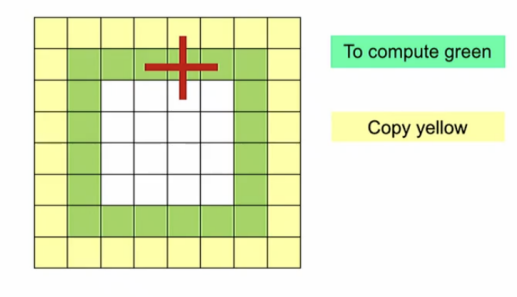
You can have n-dimensional stencils



15.2 Communication Patterns in Stencil Code

Each partition needs information from its neighbors.

What can you do? If you have a + stencil, you add an extra layer around the normal node partition, which is called the ghost zone.



For example, the yellow zone is the ghost zone. You do not have to communicate that again.

If you use an async send and receive to do this, you can hide the latency – once you are finished and start computing, you can initiate the send for the next version of the ghost zone while you're still doing the previous computation. By the time the computation is done, most of the transfer is already complete.

For instance, in the project, binning is a way to partition the particles. How might this be useful?

Every partition is going to have some extra elements surrounding the whole partition, so I don't have to keep sending and receiving individual elements. The computation can proceed as normal.

In some cases, it is common to have 2-point or 3-point or onward-sized stencils (like how larger convolution filters require a larger stencil). So, it's not a silver-bullet solution. Consider how much data you are transferring. MPI has some built-in routines for handling haloing and ghost zones.

15.2.1 How to do Ghost Zones

- In the previous algorithm ghosting required cells to be communicated between processors. This can happen before the computation initiates but you have to include it in your timing!
- Use non-blocking MPI operations for "halo" exchanges if possible to better overlap communication with computation! (hint: halos might help with your project's MPI section)
- In some algorithms ghosting can lead to extra/redundant computation compared to the implementation that does not have ghost zones: remember that computation is often cheaper that communication so practitioners often prefer to do more compute if it helps to reduce the overall communication.
- Size of ghost region (and redundant computation) depends on network/memory speed vs. computation (flops) speed.

15.3 Approximation of Interactions

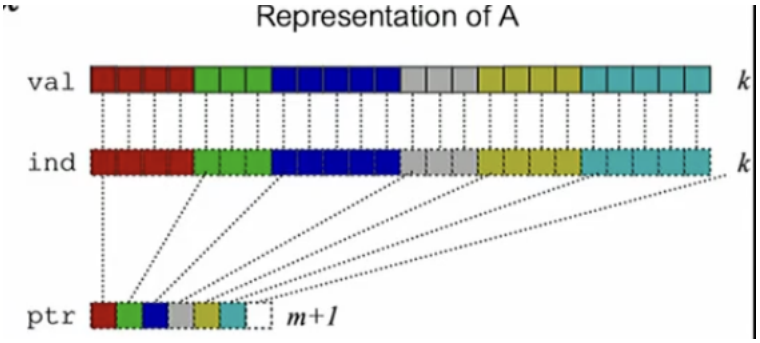
For the project, this might be more useful.

A huge sector of HPC is about simulation. You might be simulating particles; you may have some interaction with electrical charges. Electrical charge, in particular, even gravitational bodies, they react to each other using some sort of proportionality.

15.4 Compressing Data (Sparse Matrices)

Many scientific codes work with matrices that are sparse. A sparse matrix is a matrix that has very few non-zeroes and all of it other elements are zero. For example, if we want an adjacency matrix of a graph with very little edges.

How would we implement this? Well, here's one way



You go from a sparse representation to a dense representation.

For every non-zero in my `val` array, I will have a value in my `ind` array. The `ptr` array just tells me where the rows end and start.

If I want to do some computation:

Matrix-vector multiply kernel: $y = y + A\,x$

```
for each row i
  for k=ptr[i] to ptr[i+1]-1 do
        y[i] = y[i] + val[k]*x[ind[k]]
```

What is compression good for? It can reduce the memory footprint by a lot. However, optimizing sparse matrix operations is very hard.

16 GPUs

GPUs are connected in the computer using PCIs. These are the very long “connectors” that you see in your computer.

Each link consists of one or more lanes. The more lanes you have, the better bandwidth you have. The longer the plug, the faster the bandwidth, as you have more lanes.

The GPU is connected to the CPU via the PCIe bus. The GPU has its own memory, usually much smaller than CPU memory. Most of the things transferred across the PCI

bus is data that you transfer to the CPU. That's why the better the PCI, the better the GPU. Most of the time, memory bandwidth is the bottleneck.

The transfer speed between GPU and its own memory is really high, 2 TB/s. The transfer between CPU and its own memory is pretty fast (90GB/s), the order being GB per second.

One of the reasons is because the CPU is all about optimizing latency. It optimizes serial performance. It is always focused on decreasing how long you have to wait before executing the next instruction. Even stuff like pipelining – just the way to decrease latency. It doesn't give much to throughput.

Only when you add more threads, you can get throughput increase. But most CPUs nowadays don't have that many threads.

Everything in the GPU is designed for a very specific purpose. While CPU is latency-optimized, GPU is throughput-optimized but hides latency pretty well.

This is why the GPU can be used as an accelerator on the side. The two optimizations complement each other.

You do want low latency on your CPU, but the CPU isn't designed for high throughput results. Everything that you need a high throughput result, you offload onto the GPU. You have things on the GPU not well suited for the CPU, and the other way around.

- CPU is optimized for low-latency access to cached data sets.
- It has a large control logic for effective execution of serial code.
- A few ALUs, large control logic

And

- GPU is throughput optimized, ideal for data-parallel computation
- More transistors dedicated to computation
- Lots of ALU/cores but small control logic – good for high throughput

The GPU has a small brain but lots of compute power. It is up to the programmer to make their program very parallel to use the computer power on the GPU.

If you don't have a nice way to split up your tasks, to the point they can't work in isolation, the GPU won't help you at all. That's why the GPU started off being designed for graphics, where it is embarrassingly parallel, but once you have complicated things like tree traversals, GPUs really don't benefit.

16.1 Streaming Multiprocessor (SM)

A collection of a bunch of these small cores on the GPU. When you change GPU models, what mainly changes are how many SMs you have.

What is inside an SM? Firstly, the register file. You will have thousands of registers. In a CPU, the register file is usually 32 at max + some special purpose registers, but each SM is going to have thousands. Problems like register spilling is more easily avoided on the GPU.

It has a simpler cache system. It is not like the CPU that has a big memory hierarchy.

Then you have a warp scheduler. That's it for an SM.

An SM is fundamentally, those components. Compared to a CPU, it is way simpler.

16.2 Computations With GPUs

The GPU itself is fundamentally a SIMD machine (single instruction multiple data). They have massive thread parallelism – 100s of processors. If you are doing a matrix operation, as if you are adding some value to every element of that matrix, you might as well have one thread for every element of the matrix. This takes advantage of massive thread-level parallelism, which is why GPU is great for data-parallel computations.

Say you want to add some array A and B to C. How would you do that on a CPU? Allocate memory, use for loop to add elements pairwise. On a GPU:

- Create N threads
- Partition data equally into ranges among the threads
- Each thread adds its own allocated portion of A, B, C

- Wait for all threads to finish

The maximum speedup is the no. of threads you have. You can't cram CPU threads indefinitely as they are expensive and harder to manage, that's why there is fewer no. of threads on a CPU.

And if you have threads on a CPU, you will have context switches and memory contention.

If you are going to do the exact thing on a GPU, what you must do is:

- Allocate memory on the GPU (allocate the arrays)
- Transfer the data onto the GPU from the CPU
- Launch a kernel that does the work (function executed by each processing thread)
  - This spawns a massive no. of threads
  - Each thread is instructed to handle a few elements
- Wait for all threads to finish
- Transfer results back to CPU memory

GPU hides memory latency better than CPU. GPU switches between threads to hide latency. Threads do **light-weight** jobs, and we have tons of registers, so switching between threads is free. The GPU is designed to have many threads working at the same time as possible. The processing units (SMs) are designed such that context switching and holding the state is extremely cheap.

GPUs have a lot more cores, but smaller clock speed (100s of MHz). Control unit is much simpler, so we have much less diverse computations. Smaller caches, but not quite the same goal as for CPUs.

You focus on specializing for one thing instead of having to be generous purpose, you can get a lot of gains.

16.3 Diversity of GPU Applications

- HPC

- Numerical analysis
- Physics simulations
- ML
- Databases
- Still good for data processing and rendering

16.4 Debugging on the GPU

It is harder to implement a debugger on a GPU. However, CUDA does have lots of insight tools – a profiler and an inspector. It can help a lot with profiling your work. If you want to do step-by-step debugging, you could try CUDA GDB. It is not as trivial to set up as a normal CPU debugger, but there is lots of info online to walk through it. This is your best bet if you are going to step through your GPU program.

In general, even if you have a perfect step through of your GPU program, it is not as easy to debug as CPU. Think about debugging a serial CPU program – it is really easy. When we started using `pthreads` and `omp`, that becomes a lot harder to debug, as that introduces a whole slew of complexity.

On the GPU, instead of having 10s of threads, you have thousands. Spend time to really look at your program and understand what it is doing before touching the debugger. You may not get as much information as you think if your step-by-step debug.

The typical workflow for a CUDA program:

- Allocate memory on the CPU (**host**)
- Allocate memory on the GPU (on the **device**)
- Transfer data to device memory
- Launch kernels
- Wait to finish
- Wait to finish
- Transfer data back to host memory, if needed

The basics:

- Host: CPU
- Device: GPU
- Kernel launched with `<<<BLOCKS, THREADS>>>`

And you have certain qualifiers

Qualifier	Executed on	Callable from	Conditions
`__device__`	Device	Device	
`__global__`	Device	Host (typically)	
`__host__`	Host	Host	

16.5 Grids, Blocks, Threads

Thread: basic unit of execution/parallelism. Can be organized in a 1D, 2D, 3D layout, but the way they are grouped together are in blocks. Threads are always collected together in blocks. Threads can communicate with each other, but threads outside the block cannot communicate with each other.

A **block** must always be a multiple of the warp size, as threads are scheduled in warps (batches of 32 threads). The reason why you can group threads in a block is because they have shared memory. If you have some reason for why it is useful for a group of threads to communicate with each other, then it's extremely nice to have that block idea. You take a block, those blocks get executed on a SM, but the SM itself will grab warps out of a block and run maximum warps at a time. Switching between multiple blocks, that is relatively cheap for the GPU to do.

Once you have multiple blocks, they are arranged in a grid. A grid can be 1D, 2D, or 3D, how you like to look at it.

- Threads in the same block can share data and sync with each other while doing their share of work

- They communicate by shared memory

However, threads in different blocks cannot cooperate and execute in any order.

A **grid** is a logical organization of a collection of blocks. The host launches different Kernels. These kernels run on different devices.

Each core executes a thread. A thread block is picked up by an SM. If every core is executing some thread, it is natural to have a thread block map to an SM, as an SM is built up of many different cores.

Let's try to take an $N \times N$ matrix and go through and increment all elements by some particular number. What you'd like to do, is to have every thread to handle incrementing one element of the matrix. You can assume the matrix we care about is square and let's assume $N = 1024$. We know the size of the matrix, and we want to decide how many threads should be in the thread block.

Pick block size to be a multiple of warp size. Then, we need $1024 \cdot 1024 \cdot \frac{1}{256} = 4096$ blocks.

If we organize threads in 2D blocks, we have 16×16 threads per block.

If we organize blocks in 2D grids, then we would have 64×64 blocks. This is just one of many choices we can make.

We can now declare them in CUDA C:

```
dim3 threadsPerBlock(16, 16, 1); //256 threads in total;
```

and

```
dim3 block(N / threadsPerBlock.x, N / threadsPerBlock.y,
1);
```

(This is 64×64 blocks) Which is 4096 blocks in total This is the execution configuration. This is how you tell the GPU everything you want laid out.

16.6 Identifying and Indexing a Thread

There are built-in notations which a thread can use to identify its index in the grids and blocks.

- `threadIdx` is thread index within its block
- `blockDim` is the size of a block, how many threads in each dimension
- `blockIdx` is the block index in the grid
- `gridDim` is the size of a grid, how many blocks in each dimension

Each thread can identify its assigned element, such as:

- `int i= (blockIdx.x * blockDim.x)+ threadIdx.x;`
- `int j = (blockIdx.y * blockDim.y)+ threadIdx.y;`

16.7 CUDA C Typical Program

- Allocate memory on CPU, the host
- Allocate memory on the GPU, the device
- Transfer data to device memory
- Launch kernels
- Wait to finish
- Transfer data back to host memory

16.8 Parallel Execution

Everything will be executed in units of a warp. The SM will take whatever thread blocks are assigned to it, and it will take a chunk or warped block threads at a time and execute them. Different blocks can execute concurrently, but within a warp, they can execute in a lock step – they are all at the same point of execution at the same time – until control flow interrupts. If you need to sync across other warps, use `__syncthreads()`.

16.9 Warp Scheduler

Every warp will always be executing the same instruction but applied to different data. The warp scheduler will pick a particular warp, it will start executing that. Some warps may have to stop, so the warp scheduler may have to context switch to another warp and get that warp to execute all its instructions and so on to other warps.

Thread divergence is when threads in a warp go through different execution paths, where some of them will have to execute one control path and others will have to execute another.

In some cases, all threads in the same warp execute the same control flow path. This is Coherent execution.

In the other case, you could have divergent execution, where threads in the same warp have different control flows. Threads must be in lock step, so they must converge and join again later. The GPU will execute all threads in one path, and the other path is blocked and must wait until the control flow reaches the other path. Then, that path will run.

After both of these cases are done, the control flow will join again, and they can once again execute at the same time.

This is a very important fundamental notion of compiler and GPU stuff.

16.10 Limitations

There are some fundamental hardware limitations of the GPU.

- There is a max. no of threads you can put in a block
- There is a max. block count per SM. Exceed that and your performance will suffer
- There is a max. block count per grid
- Those numbers depend on your GPU, so you have to look it up based on your machine
- If you get it wrong, your kernel will fail to launch

Even on the GPU, you will quickly run into limited benefit. While the GPU is a throughput machine and is optimized for launching a bunch of threads, but the best thing isn't always max. out the thread blocks or max out the kernel.

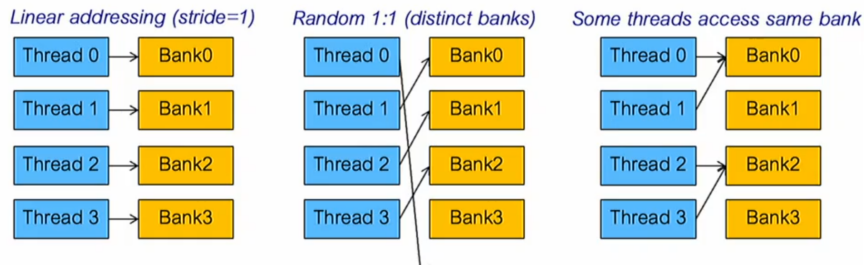
The best way is to profile.

17 General-Purpose Computing with Graphics Processing Units (GPUs)

17.1 Memory Types

- Global memory
  - Most data reside here
  - Host communication (this is where data gets transferred from or to CPU memory
  - Shared by all threads
  - Large size, a few GB, but slower than shared memory
  - L1 cache helps hide the latency for global and local memory accesses
  - Good bandwidth via memory coalescing
- Local memory
  - **Same speed as global memory**
  - Private per-thread global memory
  - Auto variables, register spill
  - Same speed as global memory but accesses are coalesced
- Shared memory
  - Lower latency than global memory
  - Acts as software programmable cache (a chunk of L1)

- Declare intention by using `__shared__` keyword
- Organized into 32 banks
  * Successive 32-bit words are assigned to successive banks
  * Any memory load/store of *N* addresses spanning *N* distinct memory banks can be serviced simultaneously ⇒ *N* times the bandwidth of a single bank
- You need to keep all the banks busy at all times. If you're not using all the banks at once, you won't get that speedup. There are patterns that could make speed worse than global memory access.



Each thread must use its own memory banks. If that is the case, then all banks are able to operate simultaneously. If you have a 1:1 mapping, then you are in good hands. Avoid bank conflicts! Threads accessing bytes within the same 32-bit word is okay as this will not result in conflicts.

If there are conflicts, accesses must be serialized. Now, you have a 2-way conflict, so you have one bank that is doing nothing. You want to avoid this as much as possible.

Shared memory is much faster than global memory (like L1 cache, a controllable part). Shared memory is shared by threads on a block, which lets threads cooperate. You can use `__syncthreads()` for block-level barriers.

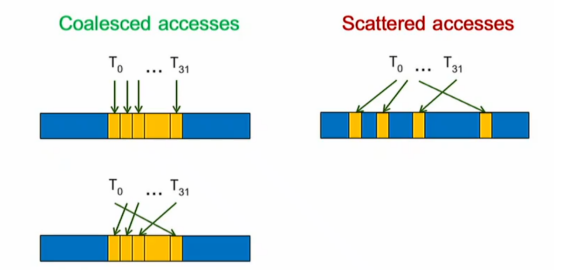
Its facilities global memory coalescing ion cases where it would not otherwise be possible. Avoid conflicts.

For GPUs, you need to have a lot of parallelism. You want to coalesce memory accesses – for all warps, you want them to access elements that are grouped together in memory. On the GPU, having divergent execution can kill performance, as all threads in a warp must be in the same step in the control-flow graph. Shared memory can help you make

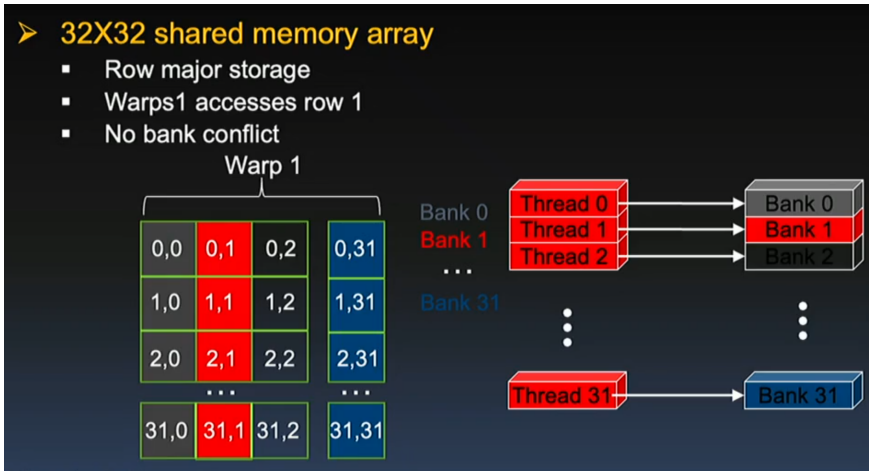
the most use of memory but avoid bank conflicts. Rework your data access patterns when needed.

17.2 Memory Coalescing

Wrap accesses should reference sequential memory location for best performance. If your memory is scattered, coalescing makes them contiguous.



17.3 Shared Memory Access



Every column is stored in a bank, and every warp is stored in a row. If every warp is accessing every row, then we have a perfect case. Every column element is stored in a distinct bank. We can here guarantee that there are no bank conflicts.

Consider the other case where instead of every warp accessing row-by-row, we assign warps to access by columns. Each warp would handle a column, and it would result in

accessing stuff from other banks. This results in bank conflicts, and we do not want them. **The takeaway – ensure that each thread only access its own bank.** If you really need to, broadcast. As long as you are accessing the same element, that's fine. If you are accessing distinct elements from a bank, that is a bank conflict.

If you are accessing the same element, there is no need to serialize the access, as if all these threads are loading element (0, 0), then there is no need to serialize that operation. If you have thread 0, 1, 2, and so on and they're all accessing different elements in memory, the hardware has to handle that differently.

Almost always, the bank size will be the same as the warp size.

17.4 Texture Memory

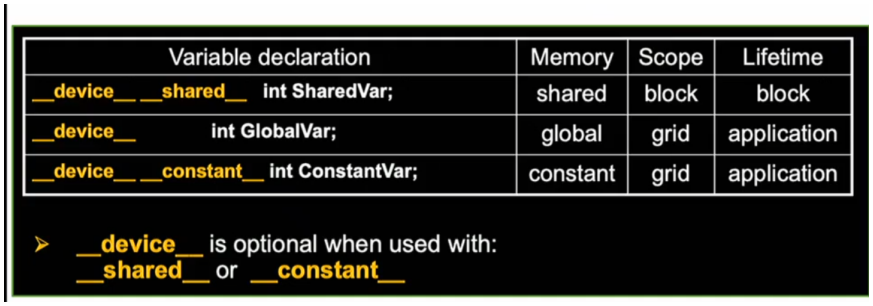
If you have the right use of textures, you'll get better performance. It is:

- A special type of read-only memory
- Although used for OpenGL or DirectX rendering pipelines, these can be used for general-purpose computing too
- Like constant memory, it is cached on chip, so there is more effective bandwidth by reducing requests to off-chip (global) memory
- Useful when memory access patterns exhibit good spatial locality (i.e. a thread reads from memory locations nearby where other threads also read).

When to use?

- If I never update data but I read it a bunch
- Do not use if the data is read rarely after it is written

17.5 CUDA Variables



Variable declaration	Memory	Scope	Lifetime
__device__ __shared__ int SharedVar;	shared	block	block
__device__ int GlobalVar;	global	grid	application
__device__ __constant__ int ConstantVar;	constant	grid	application

- __device__ is optional when used with: __shared__ or __constant__

17.6 Atomic Operations

Necessary to avoid race conditions on saved data. Normally, $x = x + a$ reads a value of x, adds a to the value, and writes it back. In a multi-threaded environment, race conditions can lead to wrong results, so we must have a way to perform this read-modify-write atomically. CUDA supports several atomic operations:

- `atomicAdd`, `atomicMin`, `...`

Atomics have great for many use cases, but do not rely on them blindly as it could harm performance. Think of the context about your program, do some profiling and experiment with different ways of laying out your programs. Sometimes, atomic is the fastest way, but it may not give you a lot of speedups if there is a lot of contention for a specific resource.

17.7 Memory Allocation

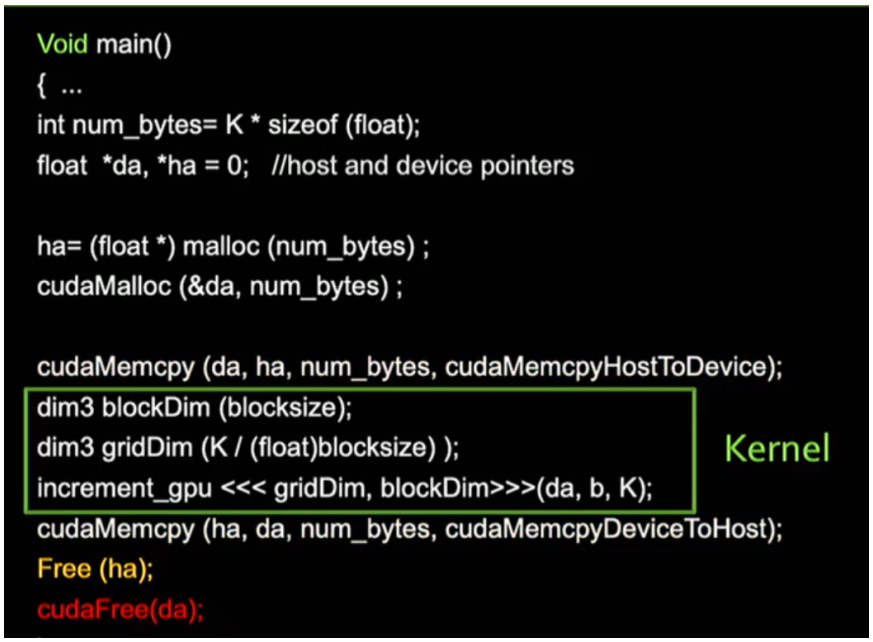
You have to first transfer everything you need initially from the host to the device, let the device do the computation, and then transfer the data you need back to the host. For the GPU, the memory API is similar to C. Pointers are used to allocated addresses on GPU and CPU, so do not dereference CPU pointers on the GPU and vice versa, as that would cause your app to crash.

Here is device memory managed from CPU:

- `cudaMalloc`
- `cudaMemcpy(void *dst, void *src, size_t nbytes, enum type_of_transfer)`
- `cudaMemset(void *pointer, int value, size_t count)`
- `cudaFree`

Here's an example of running a kernel. I have some area *da*. I'll have to figure out how to work with that. I need to figure out how much memory in need. Then, I allocate two pointers: `da` for device, `ha` for host. Then, I need to allocate memory on the CPU for `ha`, then I allocate the same amount of memory on the device using `cudaMalloc`.

Then, I have to first transfer from the CPU to the GPU. `Memcpy` is always the destination to the source. Then, I have a variable that tells if I am copying from the host to the device, or the other way around. Now, I'm free to launch my kernel and it will run, and once the kernel finishes running, CUDA will return control to the next statement.



This is the most common way to handle memory. There are newer types of memory such as unified memory, where in point of view the memory space is shared. You can also have pinned memory where the GPU will access CPU memory directly, but that has more specialized use cases. The way in the screenshot above is the most common

way you will do it.

18 Optimizing Parallel Reductions

Here are some common design patterns you'll see in CUDA code. If you want to figure out what the thread ID is inside a warp, that is usually called a lane. That is common among all SIMD designs. On your CPU, with your normal SIMD registers, you may have up to 16 lanes, maybe up to 32. Each element inside that SIMD register is referred to as a lane. That is the same for the GPU. That is what the Thread ID is inside the warp.

```
int lane = threadIdx.x % warpSize;
lane = threadIdx.x & (warpSize - 1); // the two lines
   return the same value
int warp_id = threadIdx.x / warpSize;  // warp ID inside a
   thread block
```

To dynamically allocate shared memory:

From the host, pass the shared memory size as an argument:

```
Reduce1<<<dimGrid, dimBlock, shMemSize>>>(d_idata, d_odata)
```

From the device function used the unsized `extern` array syntax:

```
Extern __shared__ int sdata[];
```

18.1 Parallel Reductions

We already know the reduction tree structure. This is a very common way to perform a parallel version of a reduction. If we have a large enough array, we need to employ multiple thread blocks. For most hardware and GPUs, you can have a maximum of 512 threads per block. Once you have anything larger than that, then you have to use multiple blocks, even with the largest block size.

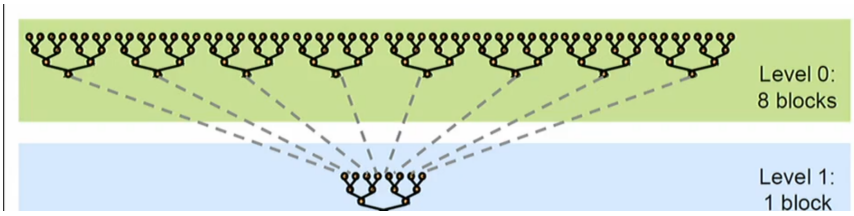
The idea is that if we have multiple thread blocks, and every thread block is in charge of reducing a portion of an array – if you want to use shared memory in the blocks them-

selves, from the previous lecture shared memory cannot be communicated among thread blocks, only within thread blocks.

So, how do we synchronize outside of thread blocks. If we have a partial result among all thread blocks, we try to synchronize. Once every thread block has finished computing, we will stop there, collect our results, and move onto the next level. Unfortunately, **not possible.** CUDA does not have global synchronization. So, how do we get around this?

We decompose into multiple kernels where each kernel launch serves as a global synchronization.

By splitting into multiple kernels, that is one way to enforce a global synchronization. By the time the kernel is done, all of the thread blocks have finished their partial results. Then, you can start another kernel.

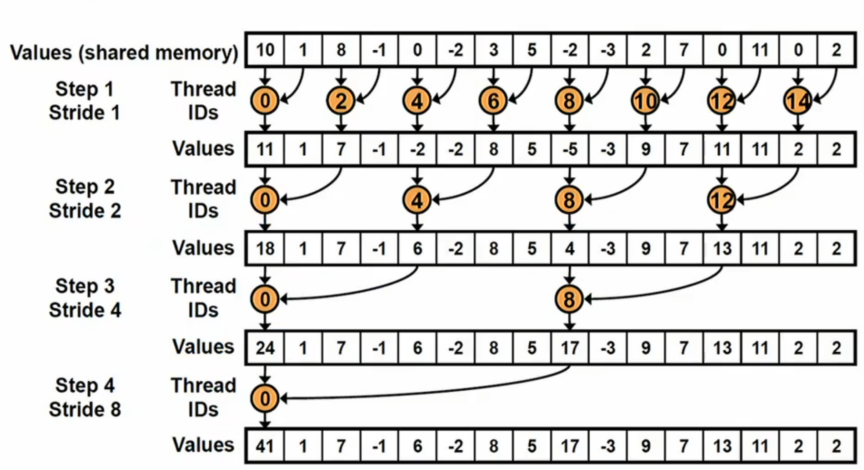


What are the problems we could face with this? Very similar to the tiled matrix multiplication example, we will show how a program might evolve over time.

18.1.1 Attempt 1: Interleave Addressing

Local data: Each thread loads one element from global memory to shared memory.

For reduce: A thread reduces two elements: thread 1 adds first 2 elements, thread 2 addresses the next two and so on. Half of the threads are deactivated after each step. When there is only one thread, stop the program. Then, write back to global memory. You start with $\frac{n}{2}$ partial results, then $\frac{n}{4}$, then all the way to 1 partial result and your reduction tree is done. It would look like this:



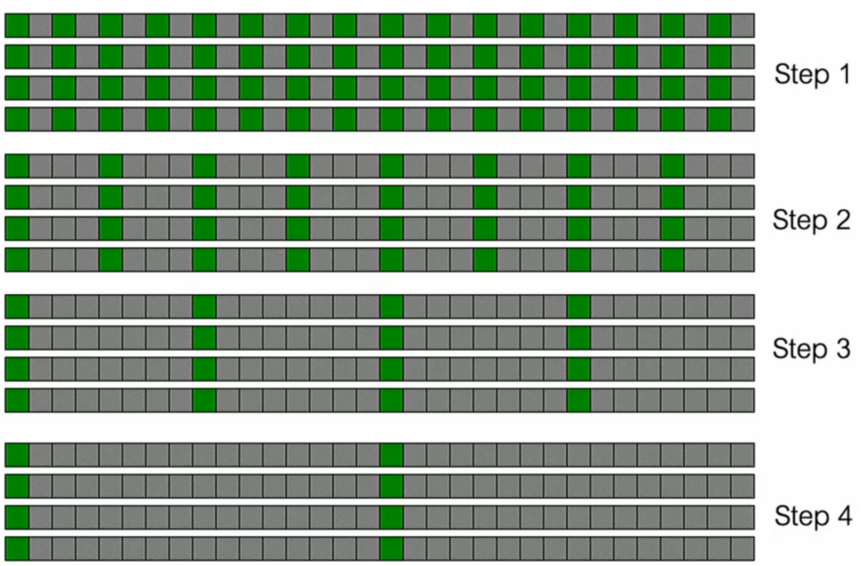
Implementation 1: Interleave Addressing

```
__global__ void reduce0(int *g_idata, int *g_odata) {
  extern __shared__ int sdata[];

  // each thread loads one element from global to shared mem
  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
  sdata[tid] = g_idata[i];
  __syncthreads();

  // do reduction in shared mem
  for(unsigned int s=1; s < blockDim.x; s *= 2) {
    if (tid % (2*s) == 0) {
      sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
  }

  // write result for this block to global mem
  if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

This is synchronization with a block, not all blocks in a GPU. Why are we exponentiating s? Because each time we loop, we use half as many threads. This is just a way to implement that.

This has an issue: this execution is divergent. What is happening is, in this example, is that more threads start taking different paths. If we time this kernel, the bandwidth that we attain is not good. This is no where near peak bandwidth. The time may seem fast, 8ms for a kernel for 4M elements, but really, this performance is not good. Note that the block size is 128 threads.

If a block is 128 threads, then we have 4 warps (each warp is 32 threads). What's happening is that one thread is doing a reduction, and another is disabled. This is extremely divergent. We start using less threads over time. What is a way we can do to make these warps more coherent?



Make sure that a warp either does nothing or everything (as much as possible).

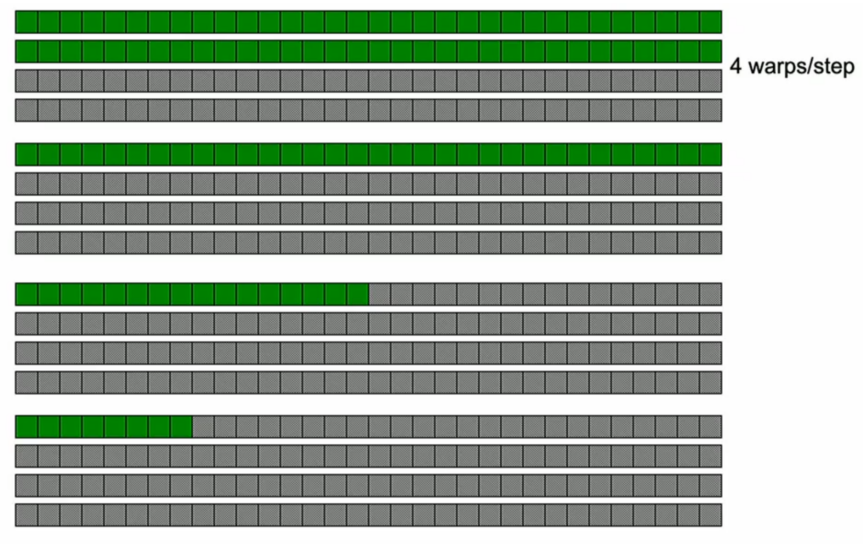
- Replace the divergent branching code:

```
for (unsigned int s=1; s < blockDim.x; s *= 2) {
   if (tid % (2*s) == 0) {
      sdata[tid] += sdata[tid + s];
   }
   __syncthreads();
}
```

- With strided index and non-divergent branch

```
for (unsigned int s=1; s < blockDim.x; s *= 2) {
   int index = 2 * s * tid;

   if (index < blockDim.x) {
      sdata[index] += sdata[index + s];
   }
   __syncthreads();
}
```

Which lets us end up like this:



19 CUDA Streams

19.1 Occupancy

Hiding latency: When one warp is stalled, execute a different warp. We need a metric related to how many active warps are on an SM. Because warps must lock step, if one thread is stalled, the entire warp cannot complete execution. To hide that latency, the scheduler will pick another loop. Context switching is very cheap on a GPU.

Occupancy: ratio of active no. of warps per SM to max no. of possible active warps

Higher occupancy does not always mean higher performance.

- At some point additional occupancy does not improve performance
  - Diminishing returns
- However, low occupancy is always bad due to poor memory latency hiding

Register availability is a limiting factor. You only have so many registers per thread, and you have to multiply that by your block count, and you can fill up your GPU's max register capacity. What can you do? Reduce the registers each of your threads need.

19.2 Pinned Memory

The idea of having direct communication with DMA between host and device. Most memory on your CPU is pageable memory, so it means that your OS can evict it.

Guess what happens when you try to access a region of evicted memory? This causes a page fault. The OS will hac to coordinate to bring that memory back into the ram. If you try to access a region of paged out memory, that's going to be problematic.

There are regions of your memory that are pinned. It means they can't be page out. Your OS is not allowed to copy this part, put it into disk… it is reserved to not be paged.

If you don't use pinned memory, if the memory you need is paged out, what has to happen is that the host has to copy your memory to pageable memory then your memory has to copy it into pinned memory.

Well, let's just use pinned memory right away.

19.3 CUDA Streams

When you compile a C program to an object file, you will see a list of instructions.

A stream is that notion applied to kernels on the GPU. If you want to have multiple kernels running concurrently, you need to know what instruction to execute next.

Operations in different streams can run concurrently.

All kernels and data transfers run in a steam.

- No stream specified; default stream used
- Default stream has slightly different semantics than other streams

Some important details about stream:

- Copies of memory from the host from the device and device to the host is blocking
- Copies are blocking, but kernel is async.

19.4 Non-Default Streams

You must create streams on the host side using some CUDA calls. If you want to use multiple, use stream create and stream destroy. To use a stream, use the stream as a context when launching a kernel.

19.5 Synchronization on a stream

You might need to sync the host code with operations in a stream.

`cudaStreamSynchronize(str)` blocks the host until all operations from `str` is complete.

19.6 Overlapping Kernels and Transfers

When you transfer data onto the GPU, it implies that there is a kernel that runs on the data. What you can do is interleave multiple data transfers. If you have multiple data streams, you have multiple data kernels, so you can transfer more data at a time.

For any one data transfer and any one kernel launch, you can't do any better than a synchronous transfer, you can still interleave data transfers and kernel transfers if you use streams.

- Within the same stream, order is enforced
- But if you have multiple streams, you can overlap operations across streams
- Both produce correct results

What is the difference? You must understand GPU scheduling