CS61C: Great Ideas in Computer Architecture (aka Machine Structures)

Lecture 17: Process-Level Parallelism

Instructors: Rosalie Fang, Charles Hong, **Jero Wang**

Announcements

- Project 3A due tomorrow (7/25)
  - Project 3B due next Tuesday (8/1)
- Homework 5 due Wednesday (7/26)
  - There’s a “coding” question on this homework! Start early!
- Labs 7 and 8 due Thursday (7/27)
  - Will cover DLP and TLP
  - PLP will be covered in lab 9 next week
- Datapath Exam Prep section tomorrow (7/25)
  - In Cory 540AB from 6-8 PM

Agenda

- Data Races and Critical Sections
- Multiprocess Code

Data Races and Critical Sections

Data Races

- Note that when we ran Hello World parallel, we ended up with the threads running in random order
  - In fact, every time we run Hello World, we get a different order!
  - The x values stayed largely in-order, but didn't always strictly increase
- Recall: OS choose whichever threads it wants to run, and can change threads at any time
- One big downside to multithreading: program no longer deterministic, has random behavior
- Formally, a multithreaded program is only considered correct if ANY interlacing of threads yield the same result.

Data Race: Example

- If we run this code on 4 threads, what possible values could x have at the end?

```
int x = 0;
#pragma omp parallel {
   x = x + 1;
}
```

Data Race: Example

- To analyze this, we need to see the equivalent assembly code (let's use RISC-V)
  - Assume that no two instructions happen at the same time
- Only the loads and stores affect shared memory, so only need to consider different ways we can order loads and stores
- How many possible orders of loads and stores?
  - $8!/(2!)^4$ = 2520 different possible orders
  - Can use the fact that all the threads are identical to reduce this to 105 orders, but still too many to check manually.

```
                    sw x0 0(sp)
lw t0 0(sp)      lw t0 0(sp)      lw t0 0(sp)      lw t0 0(sp)
addi t0 t0 1     addi t0 t0 1     addi t0 t0 1     addi t0 t0 1
sw t0 0(sp)      sw t0 0(sp)      sw t0 0(sp)      sw t0 0(sp)
```

Data Race: Example

- Case 1: All the threads run one at a time
  - Purple thread reads x = 0
  - Purple thread stores x = 1
  - Brown thread reads x = 1
  - Brown thread stores x = 2
  - Red thread reads x = 2
  - Red thread stores x = 3
  - Blue thread reads x = 3
  - Blue thread stores x = 4
- Final value: 4

```
sw x0 0(sp)
lw t0 0(sp)
addi t0 t0 1
sw t0 0(sp)
lw t0 0(sp)
addi t0 t0 1
sw t0 0(sp)
lw t0 0(sp)
addi t0 t0 1
sw t0 0(sp)
lw t0 0(sp)
addi t0 t0 1
sw t0 0(sp)
```

Data Race: Example

- Case 2: The threads are perfectly interleaved
  - Purple thread reads x = 0
  - Red thread reads x = 0
  - Brown thread reads x = 0
  - Blue thread reads x = 0
  - Purple thread stores x = 1
  - Brown thread stores x = 1
  - Red thread stores x = 1
  - Blue thread stores x = 1
- Final value: 1

```
sw x0 0(sp)
lw t0 0(sp)
lw t0 0(sp)
lw t0 0(sp)
lw t0 0(sp)
addi t0 t0 1
addi t0 t0 1
addi t0 t0 1
addi t0 t0 1
sw t0 0(sp)
sw t0 0(sp)
sw t0 0(sp)
sw t0 0(sp)
```

Data Race: Example

- Case 3: Same as case 1, except purple's store happens last
  - Purple thread reads x = 0
  - Brown thread reads x = 0
  - Brown thread stores x = 1
  - Red thread reads x = 1
  - Red thread stores x = 2
  - Blue thread reads x = 2
  - Blue thread stores x = 3
  - Purple thread stores x = 1
- Final value: 1

```
sw x0 0(sp)
lw t0 0(sp)
addi t0 t0 1
lw t0 0(sp)
addi t0 t0 1
sw t0 0(sp)
lw t0 0(sp)
addi t0 t0 1
sw t0 0(sp)
lw t0 0(sp)
addi t0 t0 1
sw t0 0(sp)
sw t0 0(sp)
```

Data Race: Example

- Some other ordering?
  - We can find orderings that give x=2,3
- Can we do any more/less?
- Can't go above 4
  - Only 4 "+1s" overall, so can't increase to 5 or more
- Can't go below 1
  - The smallest value that can be loaded by a thread is 0, so the smallest value that can be stored is 1. Therefore, the last store must be at least 1.
- Therefore, we can get any value between 1 and 4

```
sw x0 0(sp)
lw t0 0(sp)
lw t0 0(sp)
lw t0 0(sp)
addi t0 t0 1
addi t0 t0 1
addi t0 t0 1
sw t0 0(sp)
lw t0 0(sp)
addi t0 t0 1
sw t0 0(sp)
sw t0 0(sp)
sw t0 0(sp)
```

```
sw x0 0(sp)
lw t0 0(sp)
lw t0 0(sp)
addi t0 t0 1
addi t0 t0 1
sw t0 0(sp)
lw t0 0(sp)
addi t0 t0 1
sw t0 0(sp)
lw t0 0(sp)
addi t0 t0 1
sw t0 0(sp)
sw t0 0(sp)
```

Data Races: Retrospective

- In practice, most times you run this code, you get 4
  - Each thread is small enough that it's unlikely to get interrupted
- In summation, race condition bugs are:
  - Rare
  - Nondeterministic
  - Silent-failing (you get the wrong answer instead of crashing the program)
- Very difficult to debug. You have been warned…

Avoiding Data Races

- Formally, a multithreaded program is only considered correct if ANY interleaving of threads yield the same result.
- Ideally, try to have each thread works on independent data (no two threads write to the same value, or read a value that another thread wrote to)
- The hardest part of multithreading: maintaining correctness while also speeding up the code.
- How to handle cases where coordination is mandatory?

Atomic Operations

- Regardless of how we do it, this doesn't work if the instructions happen to be perfectly interleaved
- Why?
  - Every branch will have the same result, since no instruction reads a value AND writes a value to a memory location at the same time.
- Solution: Create an instruction that checks a value AND writes to a memory location at the same time.
- Known as "atomic" instructions because they do two things but are indivisible.
- RISC-V atomic extension example:
  - `amoswap.w rd rs2 (rs1)` : rd = 0(rs1), 0(rs1) = rs2 atomically
  - A bit hard to use directly, but using this, we can make synchronization primitives

Locks

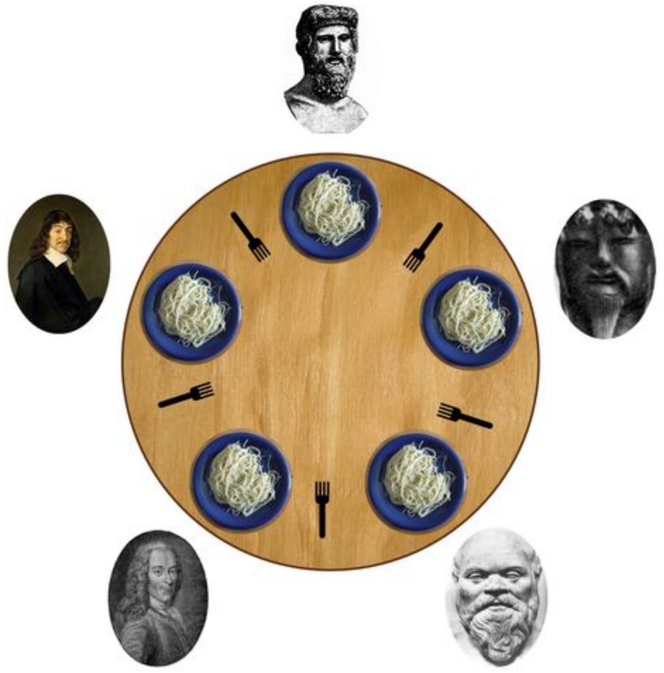
- A lock is an object which helps with synchronization
- Essentially, each thread can try to "acquire" a lock, but only one thread can have the lock at a given time.
    - Think bathroom stall lock; only one person can use the stall at a time, and we use atomics to make sure two people don't go into the same stall at the same time
- Formally, has two operations
    - acquire: Tries to acquire the lock. If successful, keep going. Otherwise, wait a bit and try again later.
    - release: Unlocks the lock and continues. Only works if we had the lock to start with.
    - Optional but common: try-acquire: Same as acquire, but if the lock is being used, return false and let the thread continue running
- Code surrounded by a lock is called a "critical section", because only one thread is allowed to run that section at a time.

Locks: Downsides

- Using a lock inherently means you need to pause one thread while waiting for another thread to run the critical segment
- Ends up making some parts of your code serial
  - Amdahl's Law strikes again!
- Is it possible for all threads to get stuck?

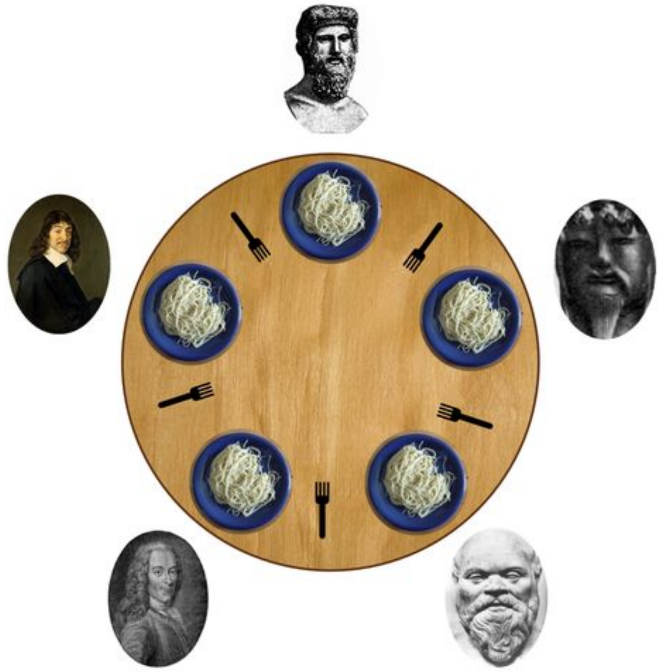
The Dining Philosophers Problem

- Five (pre-COVID) philosophers are sitting at a table eating spaghetti
- The philosophers will alternate between eating and thinking
- Between each philosopher is one fork
- To eat spaghetti, a philosopher must pick up two forks (the one immediately to their left, and the one immediately to their right). A philosopher will take a bite of spaghetti, put the forks back down, and resume thinking.
- Philosophers can't speak or coordinate
- Goal: prevent the philosophers from starving



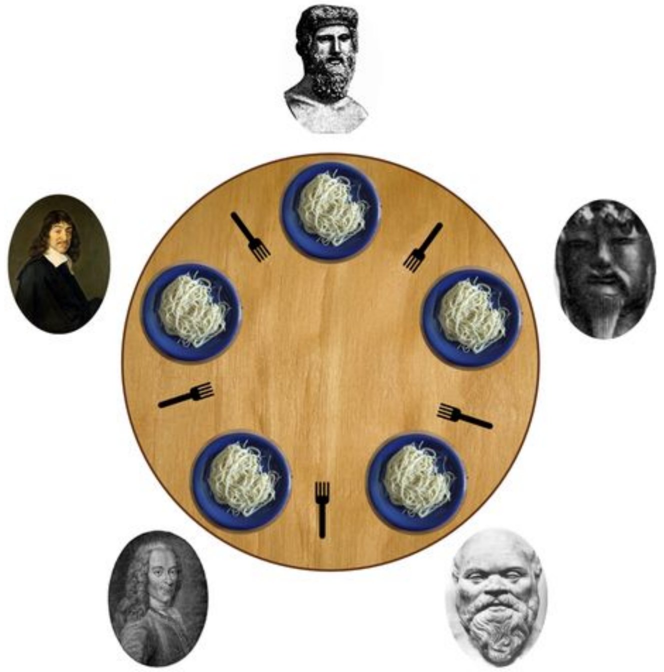
The Dining Philosophers Problem: Naive Solution

```
While True:
Acquire left fork; think until it's available
Acquire right fork; think until it's available
Take a bite of spaghetti
Put down left fork
Put down right fork
```



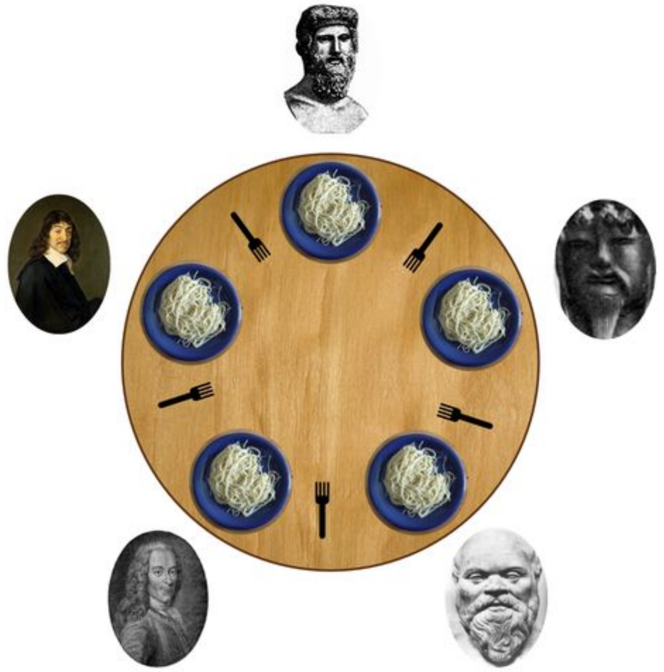
The Dining Philosophers Problem: Naive Solution Problem

- What happens?
- Imagine all philosophers grab the left fork at the same time
- All the philosophers wait for the right fork to be available, and get stuck thinking forever
- The five philosophers starve to death waiting for forks
- This is known as a deadlock; if this happens in your program, every thread gets stuck waiting for some other thread to finish work, so the program freezes.



The Dining Philosophers Problem: Resolutions

- An active topic in concurrency programming
- Several common solutions; goal is to get rid of the symmetry of the problem:
    - Only let four philosophers in to the dining room at any given time
    - Have a "manager" that assigns both forks at once
    - Choose one philosopher to pick up the right fork first before the left fork



Avoiding Data Races: Critical Sections

- Fortunately, OpenMP gives you some commands that let you use critical segments completely safely
- `#pragma omp barrier`
  - Forces all threads to wait until all threads have hit the barrier
- `#pragma omp critical`
  - Creates a critical segment in parallel code; only one thread can run a critical segment at a time.
- Using these guarantees you avoid deadlocks, so we'll recommend using these exclusively in this class.
- Locks get used significantly more in CS 162 (Operating Systems)

Parallel Hello World with Critical Segments

```
#include <stdio.h>
#include <omp.h>
int main () {
    int x = 0; //Shared variable
     #pragma omp parallel
     {
          int tid = omp_get_thread_num(); //Private variable
          #pragma omp critical
          {
               x++;
               printf("Hello World from thread %d, x = %d\n", tid, x);
          }
          #pragma omp barrier
          if(tid==0) {
               printf("Number of threads = %d\n", omp_get_num_threads());
          }
     }
     printf("Done with parallel segment\n");
}
```

Multiprocess Code

Multithreading vs Multiprocess Code

- Threads: Different instruction sequences on the same process
    - Threads on the same process share memory
    - "Easy" to communicate
    - Limited to a set of cores wired to the same memory block (1 node)
    - Analogy: a "hive mind"; you can do multiple things at the same time, but it's still largely the same entity
- Processes: Largely independent from each other
    - Different processes can't share memory
    - "Difficult" and time consuming to communicate
    - Can expand to as many cores as you have available, over as many nodes as you want
    - Analogy: A group of people; each person does their own thing independently

Multiprocess Framework Overview

- While a multithreaded program is fundamentally one program, individual processes are essentially distinct program instances entirely
- Different processes don't share memory, but generally can share the same file system
- In a multithreaded program, the entire thing crashes if any single thread crashes. In a multiprocess program, each process runs independently, so if one process crashes/terminates, the others still keep going.
    - Can create "zombie processes" that stay alive past the main process, and just eat resources until a system restart happens.
- Because you don't share memory, you can't use locks or concurrency primitives
    - Effectively restricts multiprocess programs to problems that can be split into entirely independent tasks

Multiprocess Mayhem

- In a multiprocess program, each process runs independently, so if one process crashes/terminates, the others still keep going.
- Because each process is considered independent, it bypasses many of the restrictions the OS uses to police programs
  - Use tons of memory? The OS will kill your program before it gets too big
  - Try to access someone else's memory? The OS will force a segfault
  - Run in an infinite loop? The OS will prioritize other programs to ensure everyone gets their share of computation time.
- Can be used to create what's known as a fork bomb: You write a program that creates two copies of itself to run.
  - Ex. In Windows, %0|%0 (in a bash script) is a fork bomb
- Causes exponentially many copies of the same program to run, eventually crowding out all the "real" processes and causing the system to crash.
- Please don't try this at home. This is malware. I don't think it will permanently damage anything, but…

Multiprocess Framework Overview

- Inter-process communication is done by sending messages between nodes
- Generally, messages take a lot of time to transmit/communicate:
  - Time to initialize a message packet >> time to send one byte of a message >> time to perform memory operations
- Main engineering hurdle of a multiprocess program is to find a way to split a large problem into smaller independent problems while minimizing the number of message packets and total amount of data passed back and forth

Open MPI

- Open MPI is the system that we'll be using to showcase multiprocess computation
  - Used in many supercomputers for distributed programs
  - Relatively simple
- Primarily built for Linux systems, since all the major supercomputers nowadays are x86 Intel systems with a Linux OS.
- Unfortunately, it does NOT work on Windows.
  - Please use the hive machines for these assignments!

Open MPI

- Open MPI is the system that we'll be using to showcase multiprocess computation
- Compiled with mpicc, which is a wrapper for gcc that enables multiprocess code
  - #include <mpi.h>
- Runs with mpirun command
  - Syntax: mpirun -n <number of processes>
- When run, this command copies the program to all available nodes and loads the program repeatedly
  - Compare OpenMP, which loads the program once, and does a fork/join during computation

Aside: Naming of OpenMP vs Open MPI

- What do you think OpenMP stands for?
  - Open Multi-Processing… for a multithreading library
- What do you think Open MPI stands for?
  - Open Message Passing Interface
- As far as I can tell these two groups are entirely independent organizations that decided to name their systems really similarly to each other
  - Open is used to indicate that the system is open-source, I think?
- They're about as different as Java vs Javascript.
- The moral of the story: Engineers are bad at names

Open MPI: Setup

- `int MPI_Init(int* argc, char*** argv)`
  - Initializes the MPI framework, connects everything together, etc.
  - Should be done at the start of an MPI program
    - Technically a few things are allowed before MPI_Init, but best practice is to do this first.
  - Send in the addresses to argc and argv, though for Open MPI, it doesn't actually use them; this is for compatibility with other MPI systems
- `int MPI_Finalize()`
  - Finalizes the program, cleaning up the MPI framework
  - Should be the last thing done in an MPI program. Must be done by all processes before termination

Open MPI: Process Identification

- `int MPI_Comm_size(MPI_Comm comm, int *size)`
  `int MPI_Comm_rank(MPI_Comm comm, int *rank)`
  - Returns the number of MPI nodes in the group and process ID, respectively
  - The first argument can be used if you split up your processes into groups, but we won't go into this.
  - For this class, you can always use the constant `MPI_COMM_WORLD` to get the size of the entire program/process ID relative to all processes
- Note that all of these functions receive as input a pointer which will be used to store the return value. The actual return value is used to specify if the operation worked (0 if success, an error code if failure), so don't conflate the two!

Open MPI: Example

```
int main(int argc, char** argv) {
   if (argc != 2) {
      printf("Usage: %s <foldername>\n", argv[0]);
      return 1;
   }
   MPI_Init(&argc, &argv);
   int processID, clusterSize;
   MPI_Comm_size(MPI_COMM_WORLD, &clusterSize);
   MPI_Comm_rank(MPI_COMM_WORLD, &processID);
   ... //Actual Code
   MPI_Finalize();
}
```

Open MPI: Communication

- In any communication in real life, two things need to be true:
    - The sender should be ready to send a message
    - The receiver should be ready to receive a message
- Analogy: Playing catch; if I throw a ball and you're not ready to catch it, the ball will be lost
- The same is true in MPI
- `int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`
- `int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
- In order for a message to be sent, the receiver must call Recv, and the sender must call Send. Once both functions run, the message is sent.

Open MPI: Communication

- `int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`
- `int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
- buf: An array of data to be sent/a buffer to receive data
- datatype: constants used to specify the type of the input (ex. MPI_UINT64_T)
- count: How many elements of datatype to receive
- dest: For Send only, the process ID of the intended recipient of the message
- source: For Recv only, the process ID of the expected sender of the message
- tag: For when you want to further classify messages
    - A Send/Recv pair only "matches" if the tags are the same
- comm: The communication group; for this class, just set it to MPI_COMM_WORLD
- status: For Recv only, will be set to contain the source, tag, and error message of the communication

Open MPI: Communication

- Recv can specify a sender of MPI_ANY_SOURCE and tag of MPI_ANY_TAG to receive from any sender/tag
  - Often useful in manager-worker frameworks
  - Once this type of recv happens, the recipient can then check the status for more info. If the recipient doesn't need the status, you can set that argument to MPI_STATUS_IGNORE to save memory
- By default, Recv and Send are blocking; the process will wait until its partner is ready to communicate
  - This can lead to deadlock; ex. if two processes wait for each other to send a message
  - Can avoid this with IRecv and ISend, which are nonblocking versions of Recv and Send, respectively

Application to Matrix Multiplication

- Normally, a single matrix multiplication won't be easy to multi-process
  - Too much cross-communication, so trying to use MPI will likely just add a lot of file and message operations
- However, really useful if we have many matrix multiplications to do.
- Ex. Let's say we have ~100,000 independent matrix multiplications to do
  - You have 200k files "Task0a.mat, Task0b.mat, Task1a.mat, …" in a folder somewhere, and need to make "Task0ab.mat", "Task1ab.mat", etc.
- How would we parallelize this over 1000 processes?

Multiprocessing ManyMatMul: Naive Approach

Have process 0 do tasks 0-99

Have process 1 do tasks 100-199

…

Have process 999 do tasks 99900-99999

- Any problems with this approach?
- While this will work, it might not load balance well
  - What if the tasks were sorted by size/the last 100 were 1000 times larger than all the other tasks?
- Need some way to dynamically assign work, without tons of communication

MPI Example: The Manager-Worker framework

- A very common framework for MPI programs; fairly simple to implement, while being versatile enough that you can adapt it to new purposes
- Assumes that the problem you're solving can be reduced to a set of independent tasks, that can be done in any order, independent of each other.
- Main idea: Have two roles:
  - Manager, whose job is to assign work and inform the user of progress
  - Worker, who receives work from the manager, and does the work
- Involves writing two versions of the code: one for process 0, and one for all other processes

Manager Pseudocode

```
Set up
While there's work to do:
    Wait until a worker says "I'm ready for more work" (recv from all)
    Find the next task to do
    Send to the worker what task to do
Repeat #Worker times:
    Wait until a worker says "I'm ready for more work" (recv from all)
    Send to the worker "All work done"
Finalize
```

Worker Pseudocode

```
Set up
While True:
    Send to the manager "I'm ready for more work"
    Receive message from manager
    If message is "Here's more work":
        Do the work
    Else if message is "All work done":
        break
Finalize
```

How to send messages?

- Generally want to minimize the size of each message.
- Easiest option: Assign each task/task type a number, and send that number as your message. This is your communication protocol.
  - Ex. -1 means "No more work", 0 means "Do task 0", 1 means "Do task 1", etc.
  - Up to you how you encode this, but make sure you document this somewhere for your sanity
- If some task requires input parameters or returns an output, might run several more rounds of recvs/sends.
- Alternatively, each task's input is from a file, and each task's output is sent to another file. This means you just need to send one number to assign a task, and the worker thread can directly read/write input files.

Multiprocessing ManyMatMul: Manager-Worker Approach

- We do end up "wasting" one process as a manager, but it's generally a good idea to not have the manager do other work
    - If the manager gets stuck with a hard task, ends up stalling all the other workers
- By having only one manager in charge of the big picture, no need to worry about concurrency issues
- Make sure that all processes receive a kill command; otherwise, we get zombie processes
- What if the tasks had some dependencies? (ex. Matmul 100 needs to be done after Matmul 99 and 98)
    - Set up a queue of work that can be done right now, and keep track of how much work needs to be done total
    - If a worker finishes when there's no work to do right now, tell the worker to wait and come back in a few milliseconds.

Multiprocess+Multithreading?

- You can theoretically run a multiprocess program as a multithreaded one without communications
- Generally, lack of communications causes the multiprocess program to be slower/less applicable
- At the same time, multithreaded code is limited to one node, while multiprocess code can be extended indefinitely.
- Can get some improvement by making one process per node, and each process uses #cores/node threads, but this will specialize your code more towards a particular architecture.

Performance Programming Overview

Optimization	Max Speedup	Pros	Cons
SIMD	~8x	Fairly applicable, minimal overhead	Limited by hardware, often hit hard by Amdahl's Law
Multithreading/OpenMP	#cores/node	More flexible than SIMD and MPI, generally	Concurrency issues, high overhead
Multiprocess/Open MPI	#cores	Can be extended arbitrarily large	Expensive communication, high overhead
Loop Unrolling	<2x	Reduces Branching	Minimal effect, significant penalty to maintainability
Cache Optimizations	~10x	Surprisingly good	Often requires algorithmic changes

Summary

- **Multithreading** is easy to write, but hard to get correct
- If you’re not careful, you will have **data races** or **deadlocks**
- **Multiprocess code** is generally written using a manager-worker framework
    - For 61C: Open MPI
- How else can we go faster?
    - Caches