Week 12 – Friday

COMP 3400

Last time

- What did we talk about last time?
- Readers-writers problem
- Search-insert-delete problem
- Dining philosophers

Questions?

Project 3

Assignment 7

Form teams!

Distributed Computing

Distributed computing

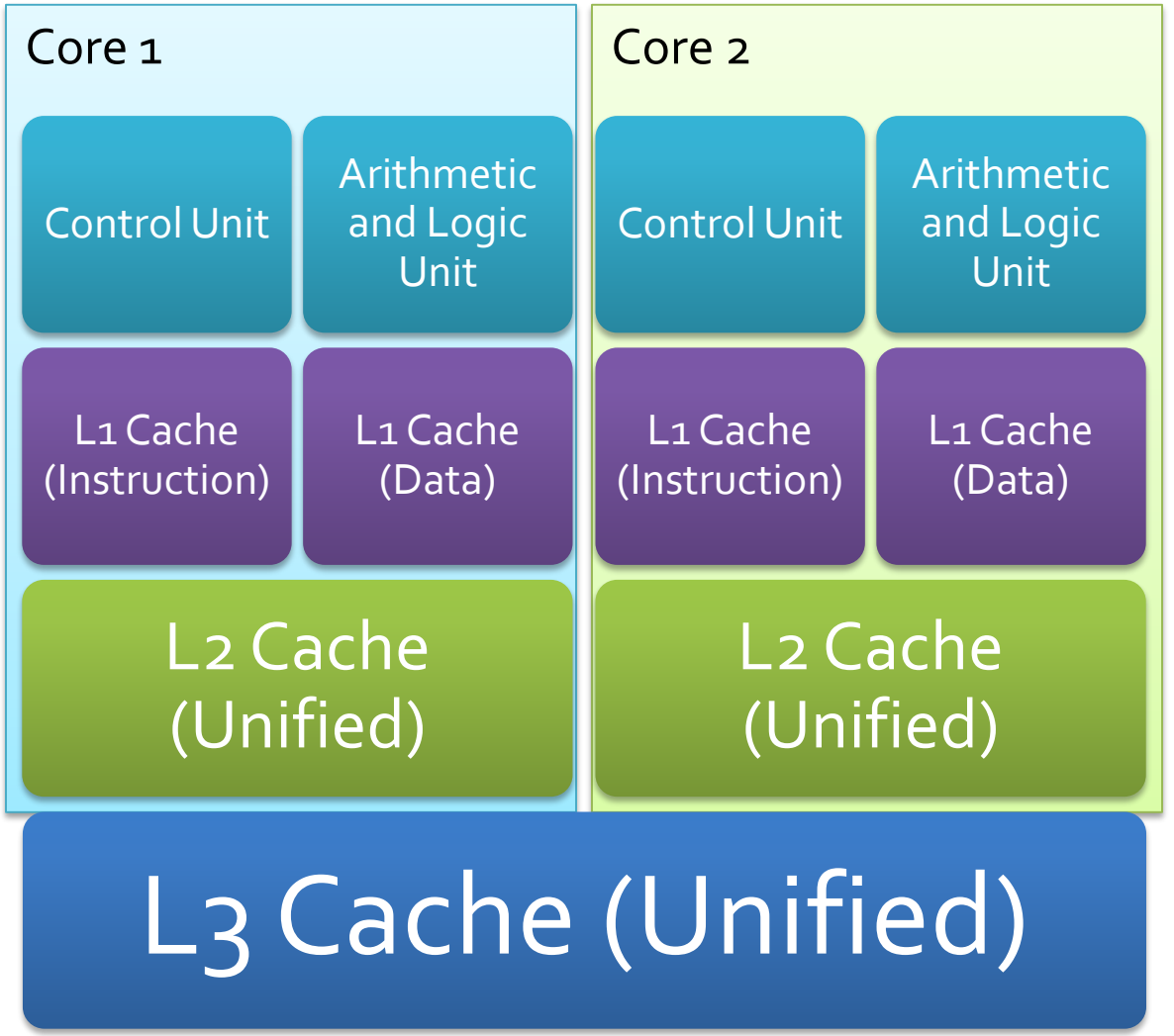
- The Internet has existed for a long time, but it's been transformed since 2000 by the power of distributed computing
- Commercial examples:
  - Google's search, app, and storage technologies
  - Facebook, which has more than 3.07 billion active users per month
  - Amazon Web Services, providing access to distributed computing that can scale based on demand
  - Blockchain technologies
- Crowd-sourced computing:
  - BOINC, a platform for home computers to work on big problems, like SETI@home
  - Folding@home, a platform for studying the computationally hard problem of protein folding
  - Great Internet Mersenne Prime Search, searching for large prime numbers, currently holding the record for the largest prime with 41,024,320 digits

Parallelism vs. concurrency

- Although a lot of computation involves both parallelism and concurrency, they're two different things
- **Concurrency** means that tasks can interact with each other
- **Parallelism** means that two tasks are running at the same time
- You can have concurrency without parallelism
  - Example: A multi-threaded program on a single-core system, which can still have race conditions
- You can have parallelism without concurrency
  - Example: Programs running on separate cores or processors that are computing part of a larger answer without coordination

Kinds of parallel systems

- Modern desktop and laptop CPUs are almost all multicore, meaning that they have separate cores capable of executing instructions independently
  - Dual core example on the right
- Symmetric multiprocessing (SMP) computers have many processors connected to the same memory
  - A model popular for older supercomputers
- Clusters use several machines connected on a network



Parallel Design Patterns

Parallel design patterns

- Concurrency is required for many systems to work and has many goals
- Parallel processing, however, is usually focused on getting work done faster
- To do this, parallel design patterns often work at two levels:
  - **Algorithmic strategy patterns** that break down a problem in a way that can be computed by multiple processors
  - **Implementation strategy patterns** for coding up parallel execution

Task parallelism and data parallelism

- There are two fundamental kinds of parallelism that are possible
- **Task parallelism**
  - Breaking up a problem into subtasks that can be run in parallel
  - Example: Alice cooks dinner, Bob cleans the house, and Catherine gets vengeance on their enemies
- **Data parallelism**
  - Doing the same tasks in parallel but on different data
  - Alice, Bob, and Catherine each chop up 1/3 of the total amount of carrots for a soup

Embarrassingly parallel

- The easiest kind of problems to parallelize are called **embarrassingly parallel**
  - Maybe there are many unrelated tasks that all need to get done
  - Maybe there's lots of data to process, and no coordination is necessary to process it
- The following code shows an embarrassingly parallel problem, since initializing the array could easily be divided up among many tasks

```
for (int i = 0; i < 100000000; ++i)
  array[i] = i * i;
```

Divide-and-conquer

- Algorithms themselves can suggest approaches for parallelism
- Divide-and-conquer algorithms divide problems into parts, find answers for the sub-problems, and then combine those answers into an overall solution
  - Quicksort partitions into two subarrays and then recursively sorts
  - Merge sort also divides and recursively sorts
- As discussed in COMP 4500, many important algorithms have a divide-and-conquer shape, and it's often possible to let each divided task be handled by a separate thread

Pipelines

- The idea of a pipeline is to divide a task into independent steps, each of which can be performed by dedicated hardware or software
- Example RISC pipeline:
  1. Instruction fetch
  2. Decode
  3. Execute
  4. Memory Access
  5. Writeback

Pipeline performance

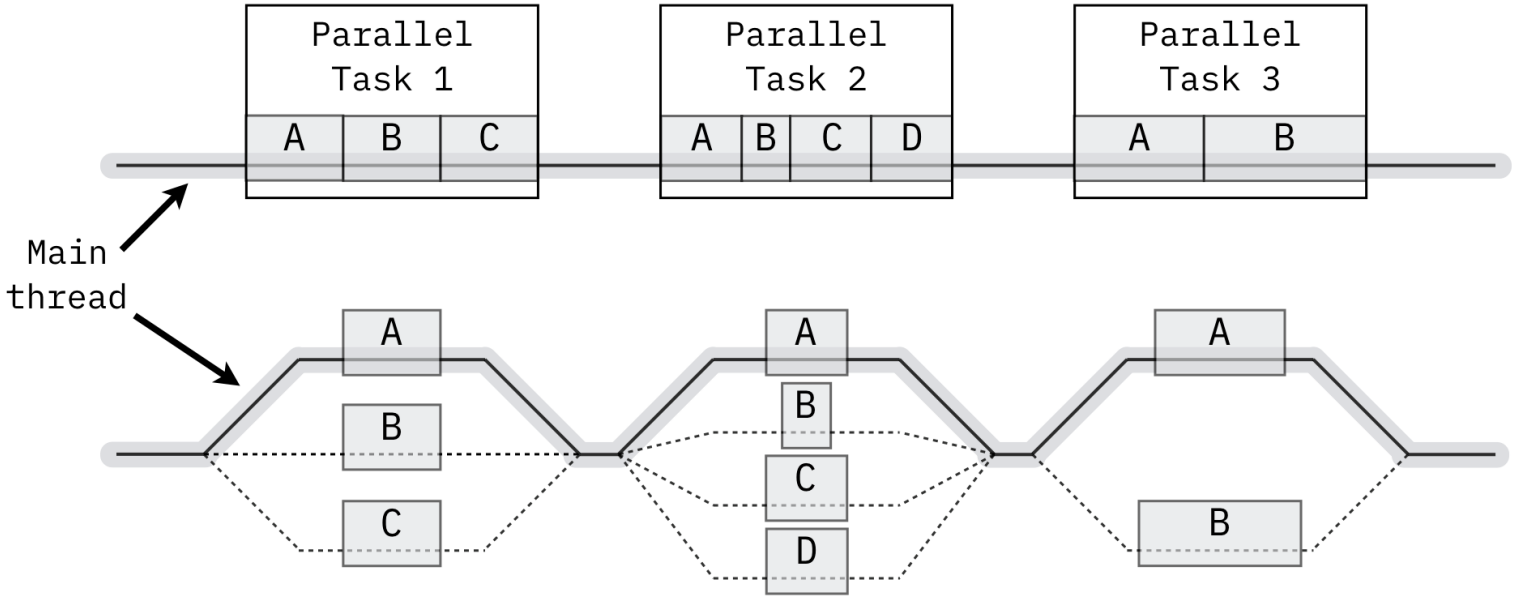
- Consider a TV show with the following tasks:
  1. Write
  2. Rewrite
  3. Film
  4. Edit
- Assume each task takes 1 week
- How much total time does it take to produce a 13 episode season with no pipelining?
- How much time does it take if we can pipeline stages with fully independent teams for each stage?
- Note that a pipeline's speed is limited by its slowest stage, the **bottleneck**
- Each stage of a pipeline can be executed by a separate thread
- If you have an ***n*** stage pipeline, what's the maximum speedup you can get?

Implementation strategy patterns

- After deciding on the algorithmic strategy pattern, it's necessary to turn it into code
- Several implementation strategy patterns are common:
  - Fork/join
  - Map/reduce
  - Manager/worker

Fork/join

- The **fork/join** pattern uses a main thread that spawns additional threads when there are parallel tasks to be done
- After those tasks complete, the main thread joins the spawned threads
- A fork/join pattern could be used for either task parallelization or data parallelization



Fork/join in code

- The following code shows the fork part of a fork/join pattern where a thread is created to do some of the work of initializing a large array

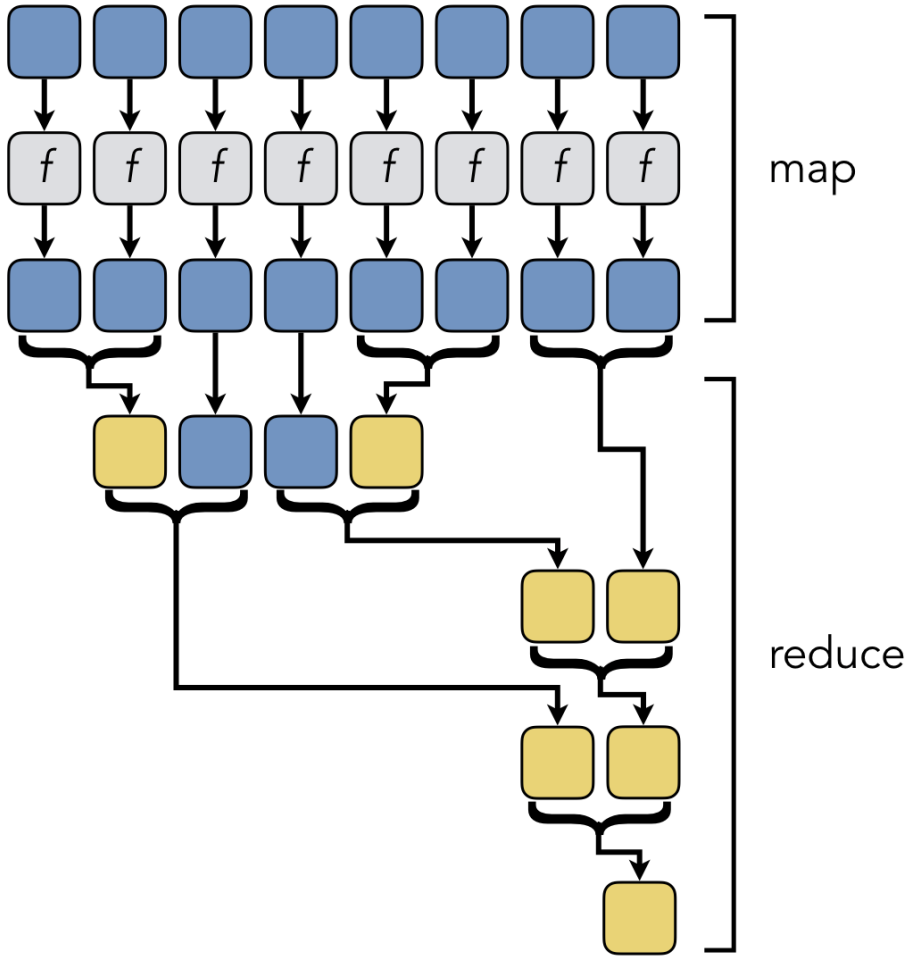
```
for (int i = 0; i < 10; ++i) // Make 10 threads
  {
    args[i].array = array;
    args[i].start = i * 10000000;
    pthread_create (&threads[i], NULL, multiply, &args[i]);
  }
```

- The OpenMP library contains macros to divide a loop between threads automatically, turning the following code into the previous

```
#pragma omp parallel for
for (int i = 0; i < 100000000; ++i)
  array[i] = i * i;
```

Map/reduce

- **Map/reduce** is similar to fork/join
- The biggest difference is a philosophical one about how the work is described
- Map/reduce has two stages:
  - Map applies a function to each piece of input data
  - Reduce combines the results to get a final answer
- Map/reduce is commonly used on clusters and distributed systems
  - The open source Apache Hadoop is a popular tool for map/reduce computing



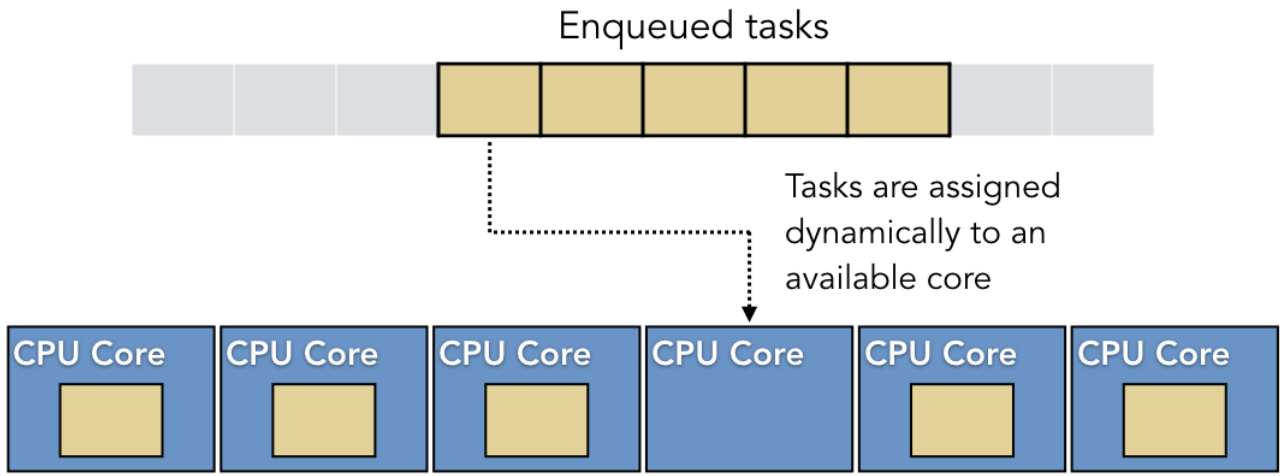
Manager/worker

- The manager/worker thread pattern is commonly used with task parallelism
- Independent tasks are given to work threads that communicate with a central management thread
  - Event handling, for example, can be viewed as manager/worker
  - Workers can also wait for a data value to change from **NULL**, as in the code below

```
void * worker (struct args * _args)
{
  struct args *args = (struct args *) _args;
  pthread_mutex_lock (args->lock);
  while (true)
    {
      while (args->data == NULL) // Wait for data
        pthread_cond_wait (args->data_received, args->lock);
      if (! args->running) pthread_exit (NULL);
        // Process data
    }
}
```

Parallel execution patterns

- There are additional decisions to be made beyond the algorithmic strategy pattern and the implementation strategy pattern
- How threads are mapped onto hardware is another issue
- Rather than worrying about creating too many threads initially or dynamically creating threads, one approach is a **thread pool**
  - A thread pool is a fixed number of threads with a queue of tasks
  - When a thread finishes its work, it can dequeue a new task

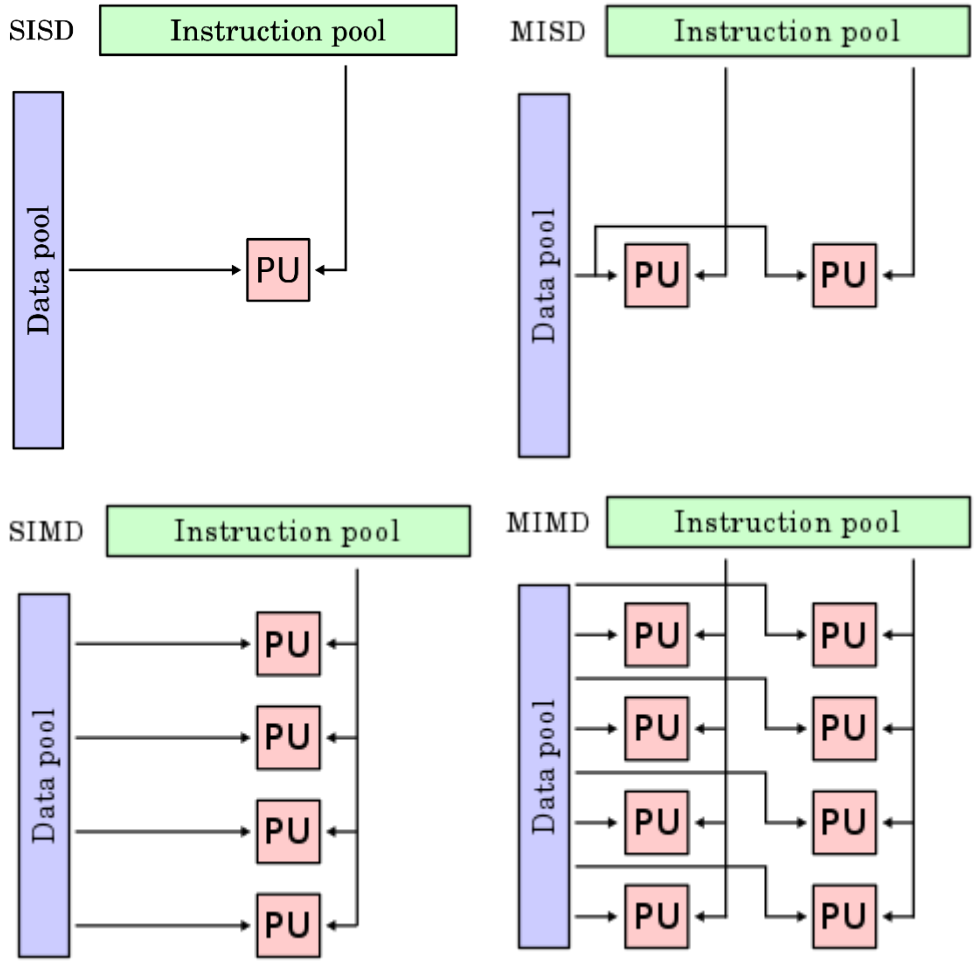


Thread pools

- Thread pool advantages:
  - The cost of creating threads is only paid once
  - Resource consumption is more predictable because there won't suddenly be a lot more threads
  - Each thread self-manages the load by getting more work when it finishes
- Thread pool disadvantages:
  - Cache performance can be poor because there's no coordination between which thread is doing what
  - Crashes and errors can be hard to recover from since we won't know which thread was doing the thing that failed
  - Managing the task queue requires synchronization that could slow things down

Flynn's taxonomy

- Flynn's taxonomy divides hardware into how they can deal with multiple instructions and multiple pieces of data
  - **Single Instruction Single Data (SISD)** is sequential processing of one piece of data with one instruction
  - **Single Instruction Multiple Data (SIMD)** is processing several pieces of data with the same instruction, like the vector processing done in graphics cards
  - **Multiple Instruction Single Data (MISD)** isn't used commonly, but it can allow for fault-tolerance because different instructions are executed in parallel on the same data
  - **Multiple Instruction Multiple Data (MIMD)** is processing different instructions on different data at the same time



Images from Wikipedia

Upcoming

Next time…

- Limits of parallelism
- Timing in distributed environments
- Reliable data storage and location

Reminders

- Finish Project 3
  - **Due by midnight!**
- Read sections 9.4, 9.5, and 9.6