CMSC416 Notes

Hersh Kumar

Contents

1 What is Parallel Computing?

Serial or sequential computing is doing a task in sequence on a single processor. Parallel computing takes a task and splits it into sub-tasks, and does them concurrently. This happens on a set of processors, often connected by a network. Some tasks do not need any communication, they are "built" for parallel computing, and these are pleasingly parallel. Parallel computing is useful for things like drug discovery, weather forecasting, and large-scale structure analysis of the universe.

Parallelism is important because it makes some science simulations feasible in the lifetime of humans, either due to speed or memory requirements. It is also useful since we can provide answers in realtime or near realtime for problems that are slow in sequential computing. Overall, we have two main reasons, the first being that we want to structure our program so parts run concurrently, and the second is that we can't achieve the speed required using sequential computing.

The architecture of a parallel system involves a set of nodes or processing elements, connected by a network. Each node has memory as well as multiple cores. There are different topologies for connecting nodes via a network, and the simplest is a torus, each not is connected to its neighbors, where the connections wrap around, hence the torus name. The most common modern network structure is known as a fat-tree network, where the nodes are in a tree, and to avoid traffic near the top of the tree, we add extra links to increase bandwidth. Another common network is the dragonfly network, which is a combination of torus networks, connected in a higher level network structure.

The way that parallel systems deal with input/output is with longterm storage on parallel file systems. Any file operation will happen on multiple drives, with a complicated software that keeps track of all of the data.

There are different parallel programming models. Parallelism is achieved through language constructs, or by making calls to a library, and the execution depends on the language and library used. The model we will use for programming single nodes will be based around spawning threads and running them. For distributed memory, we will use a message passing module, which deals with inter-node communication.

1.1 Hardware and Software

Let us begin by talking about hardware. The terminology that we use is that a processor can have multiple cores. A **core** is a single execution unit that has a private level 1 cache, and can execute instructions independently. A **Processor** contains multiple cores, places on an integrated circuit, which results in a multi-core processor. Chips are mounted on circuit boards, and the chips connect to **sockets**, which combine to create a **node**, which is a motherboard or PCB that has multiple sockets. These nodes are generally located on rackmount servers, which are based in cabinets that provide power and network connectivity.

Now moving to the software side, users login to a login node, and submits their program to a scheduler. The scheduler then allocates resources on the cluster, and then runs the job on the allocated nodes. The reason for not running things simultaneously is that threaded programs are as fast as their slowest thread, and thus we don't want threads competing for resources. The scheduler decides what job to schedule next, based on an algorithm, and what resources to allocate to the ready job.

The network is shared by all jobs executing, as well as the filesystem. However, the compute nodes (or at least part of a node) are dedicated to their job. This indicates the difference between login nodes and compute nodes. The compute nodes can only be accessed when they have been allocated to a user by the job scheduler, and the login nodes are shared by all users to compile their programs and submit jobs to the scheduler. There are also input/output nodes that act as intermediaries between compute/login nodes and the parallel file system.

What is the difference between a supercomputer and a regular cluster? Generally, supercomputer refers to a large expensive installation, typically using at least some custom hardware. These are generally made for high performance jobs, not regular, small jobs. They also generally have high-speed interconnectivity, as opposed to clusters that are seen at Google, etc. On the other hand, clusters are generally put together using commodity/off-the-shelf hardware.

2 Threads and Processes

Let us now consider the difference between serial and parallel code. Let us first define a thread. A **thread** is a path of execution managed by the OS. Threads can share the same memory address space. On the other hand, a **process** is more heavy-weight, they do not share resources such as memory or file descriptors. A process is essentially a chunk of memory, with one or more threads running in it. Serial or sequential code can only run on a single thread or process, while parallel code can be run using one or more threads or processes.

When we run a parallel program, we are aiming for faster runtimes. We also care about resource usage, which leads to the ideal of scaling, which is taking a parallel program, and running it on more threads or processes. A program is *scalable* if its performance improves when we give it more resources. There are two different types of scaling, weak and strong scaling. **Strong scaling** is keeping the total problem size fixed, (sorting n numbers on 1 process, 2 processes,...). **Weak scaling** is keeping the problem size per thread/process fixed, but increasing total problem size as we run on more threads/processes (sorting n numbers on 1 process, sorting $2n$ numbers on 2 processes, $4n$ numbers on 4 processes,...). **Speedup** is a measure of how much faster the program is running, and is defined as the ratio of execution time on one process to that on p processes:

$$\text{Speedup} = \frac{t_1}{t_p}$$

We also care about **efficiency**, the speedup per process:

$$\text{Efficiency} = \frac{t_1}{t_p \times p}$$

The best efficiency that we can get is 1. In general, speedup is limited by the serial portion of the code, which is an observation known as Amdahl's Law. This is often referred to as the serial/sequential bottleneck. Let us assume that only a fraction of the code (f) can be parallelized on p processes. In this case:

$$\text{Speedup} = \frac{1}{(1-f) + \frac{f}{p}}$$

Parallelization is not free, it requires more communication between processes. Each process may execute serial code independently, and when data is needed from other processes, messaging occurs. This is referred to as communication, synchronization, or MPI messages. There has to be some form of communication. The costs for intranode and internode communication are quite different. Across nodes, we have to use the network, while in one node we use the shared memory. As soon as we start using the network, we have latency, which is much slower than accessing memory. Sometimes we will se bulk synchronous programs, all processes compute simultaneously, then synchronize/communicate with each other, then continue to compute. This makes it easier for humans to think about the program.

There are different models of parallel computation:

- SIMD (Single Instruction Multiple Data): Processor runs single instruction on multiple pieces of data. This is how CPUs and GPUs work.
- MIMD (Multiple Instruction Multiple Data): Each thread/process runs different code on different data. This is the most general parallel model, but is harder to program for.
- SPMD (Single Program Multiple Data): Each thread/process runs the same code, on different data. This is typical in HPC and parallel computing with many threads/processes.

3 Designing Parallel Programs

When we write parallel programs, the general model is to first decide a serial algorithm. Then the question becomes how to distribute data and work among threads/processes. The programmer also needs to figure out how often communication between threads will be needed.

A simple case is doing computations on a grid, such as things like heat diffusion, Jacobi method, and Gauss-Seidel method. This is know as two-dimensional stencil computation. We have that each cell of the grid takes in information from the nearest neighbor cells.

To do this in parallel, we have several choices. One idea is to divide adjacent rows (or columns) among processes. The key thing here is that we need to also ask for data along the boundaries, since we need the neighbors of the cells along the edge, and so we will need to exchange information across that boundary.

The other method is to divide the grid into 2D blocks, and give a block to each process. The benefit of this is that the surface to volume ratio of each grid block is better than if we used the row or column partitioning. This means that we have less message data to send. However, we note that each grid has to send messages to more grid blocks than the row/column case.

Another thing we care about is a prefix sum, or partial sum, which is the sum of the elements up to a certain index in an array. The sequential recursive method for this seems difficult to parallelize, but there is a very clever algorithm that can do better than the sequential $\mathcal{O}(n)$. To do this, we first add each element to the element to the right. This makes the element in index 1 correct. We then add up the elements 2 indices to the left, which doubles the number of elements that are correct. We then continue until we are done. This is logarithmic with respect to the size of the array. We have essentially reordered the additions, so that we don't need to wait for the sum of the previous elements. In practice, we have N numbers, and P processes, where $N \gg P$. We then assign a block of size $\frac{N}{P}$ to each process, and then do the calculation for the blocks on each process locally.

Another difficult problem is the n-body problem. The naive algorithm is $\mathcal{O}(n^2)$. The naive approach to parallelizing this assigns n/p particles to each process. One thing we can do is use space-filling curves to compress the multidimensional space down into 1 dimension, and then partition like we partition an array. Another thing we can do is use quad/oct trees to partition the space into trees that maintain locality among particles.

We do these things in order to maintain load-balancing, we want each thread to do about the same amount of work as every other thread. This is because we are only done when all threads are complete, so the fastest program will have the most balanced distribution. We want to bring the ratio of maximum to average load as close to 1 as possible. A secondary consideration is that we want to load balance the amount of communication. This leads to another term, the grain size, the ratio of computation to communication. Coarse-grained algorithms do more computation than communication, while fine-grained programs do more communication than computation.

3.1 Architecture and Model Overviews

There are two main architectures for parallel computation, shared memmory and distributed memory. Shared memory is based on all processors/cores being able to access all the memory, as a single address space. There can be uniform memory access, all the memory is in one chunk, but there is also non-uniform memory access, where the memory is split up, and processors can obtain different regions of the memory via interconnects.

For the distributed memory architecture, each processor or core only has access to its local memory. Writes in one processor's memory have no effect on another processor's memory. Accesses to another processor's memory must be done via message passing, different from the non-uniform memory access of the shared memory architecture. All modern large scale systems are distributed memory systems.

That covers the architecture, but there are different types of programming models. There are shared memory models, and distributed memory models. The shared memory models says that all threads have access to all of the memory. Examples of this includes `pthreads` in C, where each thread can access all the memory (not necessarily all at the same time, but they have the ability to access any segment of the shared memory), or OpenMP, which we will use in this course.

On the other hand, for the distributed memory model, each process has access to its own local memory, and no more. One process cannot directly name an address in another processes' address space. In order to do it, we pass messages between the processes. Examples of this are MPI, as well as Charm++, which is another way of passing things between threads.

There are also hybrid models, where we use both shared and distributed memory models together. For example, using a shared memory model on a node, and then having nodes pass messages between each other.

3.2 Message Passing and MPI

Our model is built on processes running, and then arbitrarily sending and receiving messages to other processes. Usually, we have that each process is running the same program/executable, but potential different parts of the program, and on different data. To start this, the processes are created by a launch/run script. This generally uses the SPMD style of programming, one program running on different pieces of data.

The history of message passing dates back to the Parallel Virtual Machine (PVM), which was developed in 1989-1993. The point was to connect workstations and have processes communicate with each other. There were many vendor libraries in the 1980s and mid-90s, all with different APIs and parallel messsage passing methods. The MPI forum was formed in 1992 to standardize message passing models, and MPI 1.0 was released in 1994. MPI is still in development, but we will be considering functionality that was included in the first version of the standard.

MPI stands for Message Passing Interface. It is not an implementation, it is an interface, it defines the operations/functions needed for message passing. It is implemented by vendors and labs/academics for different platforms, and is meant to be portable, it is made to run the same code on different platforms without modifications.

There are some popular open source implementations, such as MPICH, MVAPICH, OpenMPI, and several vendors also provide implementations optimized for their products, such as Intel and Cray.