Introduction to Supercomputing

Tina Benhaim    Mary Li

Computational Seminar

March 17, 2009
Happy Saint Patrick’s Day!
1 Overview
   - Intro to Parallel Computing

2 A Look Inside the Box
   - Hardware

3 Diagramming the Problem
   - Fork-join Structure

4 Potential Problems
   - Variable Use

5 Conclusion
   - Summary
What is parallel computing?

Traditionally, computer software is written to be executed serially.

- Software is to be executed on a single computer with a single central processing unit (CPU).
- A problem is broken into a set of instructions.
- Instructions are to be executed one after another.
- The next instruction cannot be run until the last one is finished.

**Figure:** Serial Processing
Parallel processing in essence allows computer programs to be broken into parts and executed simultaneously.

- Software is to be executed on multiple computers or on a single computer with multiple CPUs.
- A problem is broken into parts.
- Each part consists of a set of instructions.
- Parts are executed simultaneously by multiple CPUs.

![Diagram of parallel processing]( Parallel Processing.png)

**Figure:** Parallel Processing
Parallel Processing Benefits

The main benefit of parallel processing is the speedup in execution time since multiple tasks are being executed at the same time.

Parallel processing can be managed through:

- a single computer with multiple processors,
- an arbitrary number of computers connected by a network,
- or a combination of both.
The Computer

Figure: Inside the Computer
The CPU

When someone refers to a processor, they are talking about the CPU, but within each CPU you can have one, two, four etc. cores. So a dual core would look like the following:

![Diagram of a dual core CPU]

**Figure:** The Computer Processing Unit (CPU)

The ALU, the arithmetic logic unit, performs arithmetic and logical operations. The box next to it is called a register, which stores a small amount of information.
- If the needed information is not in the register, it pulls the info from the L1 cache.
- If the L1 cache doesn’t have the info, the information is pulled from the L2 cache into the L1 cache and then into the register, etc.
- If the L3 cache needs information, it pulls it from memory.
Interesting Sidenote

The L3 cache pulls blocks of information from memory since it is generally the case that if the register calls for this piece of information from memory, it will probably need the next bit of information next.

When there is more than one core in a processor, the cores share the cache. It is more efficient if both cores are working with the same information, else the cache must store two sets of information so each core can only use half of the cache to store the information it needs.
How is parallel processing organized?

The framework of parallel processing is as follows:

Notice the dark red line that follows through from beginning to end. This is the master thread, which will execute sequentially until told to split and it is the thread to which all other threads will defer when the threads join.
Fork

The point at which the master thread creates multiple other threads which will each take a part of the problem to work on.

Join

The point at which the other threads complete their tasks and dissolve, leaving the execution of the program back in the hands of the master thread.

Synchronize

Establishing a point within an application where a thread may not proceed further until another thread (or often multiple threads) reaches the same or logically equivalent point.
Synchronization usually involves waiting by at least one thread, and can therefore cause a parallel application’s wall clock execution time to increase.

**Lock**

A command that defines how threads are able to access the process.
Costs to Parallelization

Parallel overhead is the amount of time required to coordinate parallel tasks, as opposed to doing useful work. Parallel overhead can include factors such as:

- Task (think of a task like one thread) start-up time
- Synchronizations
- Data communications between threads
- Software overhead imposed by parallel compilers, libraries, tools, operating system, etc.
- Task termination time

There is a cost to forking and joining so focus on parallelizing the core of your program.
The computational problem usually demonstrates characteristics such as the ability to:

- Be broken apart into discrete pieces of work that can be solved simultaneously;
- Execute multiple program instructions at any moment in time;
- Be solved in less time with multiple compute resources than with a single compute resource.
OpenMP

An Application Program Interface (API) that may be used to explicitly direct multi-threaded, shared memory parallelism. OpenMP stands for Open Multi-Processing.

OpenMP tells the compiler what to do, but the compiler will only follow the instructions if it knows how. Most OpenMP parallelism is specified through the use of compiler directives which are imbedded in C/C++ or Fortran source code, thus the gcc compiler that we learned about last week is one of the appropriate compilers to use.
OpenMP operates using shared memory. This means that variables in your program can be changed by any thread unless you as the programmer declare them private.

For example, suppose that your program has a counter, but each thread incremented the counter every time the thread finished it’s task, but the task output depended on the counter value.

**Warning:**

Sometimes threads operate in a different order than you supposed and spit out incorrect output.
Variable Use

Declaring Private Variables

There are several ways to declare variables private:

1. After `#pragma omp parallel` (the main directive for initializing parallelization) define the previously declared variable x to be private, i.e.,
   ```c
   #pragma omp parallel private(x)
   ```
There are several ways to declare variables private:

1. After `#pragma omp parallel` (the main directive for initializing parallelization) define the previously declared variable x to be private, i.e.,
   ```
   #pragma omp parallel private(x)
   ```

2. Declare a variable within the text so that each thread creates and uses its own variable.
   ```
   #pragma omp parallel
   {
   x = 4;
   }
   ```
Declaring Private Variables

There are several ways to declare variables private:

1. After `#pragma omp parallel` (the main directive for initializing parallelization) define the previously declared variable `x` to be private, i.e.,
   ```
   #pragma omp parallel private(x)
   ```

2. Declare a variable within the text so that each thread creates and uses its own variable.
   ```
   #pragma omp parallel
   {
   x = 4;
   }
   ```

3. Use the command `firstprivate`, which makes a copy of the variable for all threads to use individually.
The best rule of thumb

If you didn’t declare it inside the parallel section, then it’s shared.

Another tip to remember is not to use a parallel directive within another parallel directive. Most likely, all your cores have been assigned tasks already.
In parallel programming, the programmer designates how the program is broken up between the threads. The process can be executed in blocks or cyclically. For example, in the particular case of matrix multiplication, handing corresponding rows and columns might be a better way to divide the work up.
Reductions are good for operations that are both associative and commutative such as min, max, add, and multiply. For example, if we wanted to see what the answer would be if we added 1 five hundred times, we could call the reduction command.

```
sum=0;
#pragma omp parallel reduction(+:sum)
#pragma omp for
for(k=1;k<=500;k++)
{
    sum+=1;
}
```
Suppose at run time, the program divides this sum up into four tasks, assigning the first task to run the for loop for 125 iterations, the second task to run the for loop for another 125 iterations, etc. If we programmed this process to be dynamic then the program would assign the next for loop iteration to the next thread that gets done so the first thread might get a sum of 137 iterations, the second 119, the third 126, and the forth 118. In either case each thread will sum up part of the total sum and then collapse that sum into a single value at the end.

(And in case anybody was wondering, the result is 500. :0))
Load Balancing

![Flow Structure Over Airplane Wing](image.png)

**Figure:** Flow Structure Over Airplane Wing

It wouldn’t make sense to block off portions of the wing to each thread because certain areas are more computationally concentrated. The threads at easier areas will complete their tasks much sooner and sit idle while waiting for the busy threads to finish and join with them. Assign threads carefully.
Parallel computing:

- is very useful for programs that can be broken up into parts that can be executed simultaneously.
- often speeds up computation time.
- *is the future of computing.*

Outside Resources:

- http://www.openmp.org
- https://computing.llnl.gov/tutorials/openMP/
- https://computing.llnl.gov/tutorials/parallel_comp/