Lab 10: Parallelism

Why Use Parallel Execution

Consumers and computer experts expect computers to be more powerful every year. Moore's Law, first defined in 1965, stated that the processing power of computers would double (roughly) every two years. But in the early 2000s it was becoming increasingly difficult to continue to decrease transistor sizes while increasing clock speeds.

So instead manufacturers began to put multiple computing cores into a single system. That allowed computers to continue to follow the exponential growth curve of power without having to pack more transistors into a single chip, which was causing unmanageable power losses and issues with heat management.

Now most commercial computers have 4 to 16 computing cores. Even mobile devices come standard without multi-core CPUs. High performance computers can have many, many more processors. For example Texas State University's LEAP high performance computing cluster has 120 nodes, each with 28 CPU cores.

What this means is that code that is written to run on only one processor at a time is only taking advantage of a fraction of the power of a computing system. Modern programs need to take advantage of parallel execution to be able perform efficiently and competitively.

Parallel Vocabulary

These are some terms that computer scientists use to discuss parallel computing:

- **Runtime** : $\mathbf{T_p}$. The time that a block of code takes to run on p processors where p is some integer value.
- **Work**: the total number of operations in $\mathbf{T_1}$
- **Cost**: $\mathbf{p * T_p}$. This is used to measure the efficiency of a parallelized algorithm. Cost is always greater than or equal to **Work**. So code that runs in 20ms on 1 processor will run in 10ms or more on 2 processors.
- **Speedup:** $\mathbf{S_p = T_1 / T_p}$. So if code runs in 20ms on 1 processor but 10ms on 4 processors has a speedup of 2.
- **Linear Speedup**: when $\mathbf{S_p = p}$. So code that runs in 20ms on 1 processor and 10 ms on 2 processors has a speedup of 2. The p value(2) is the same as the speedup value(2), so we would describe this as **Linear Speedup.** Code that runs in 20ms on 1 processor but 20ms on 4 processors has a speedup of 2 with a p of 4. p != S_p so we would call this sub-linear speedup.
- **Efficiency**: $\mathbf{S_p / p}$. Used to express the amount or performance improvement per additional computing core. Linear Speedup has an Efficiency of 1.

OpenMP

This C++ library is one of the most common methods of adding parallelism to C++ programs. Like other libraries it adds to the tools that we can use while writing code. So **iostream** gives us **cout** and **cin**, and **cstdlib** gives us **rand()**. One big difference between those libraries and the OpenMP library is that many of the tools the are provided by OpenMP are in the form of "pre-processor directives". The pre-processor hasn't been discussed much in this class, but we have been using it.

Any C++ statement that starts with a '#' is a pre-processor directive. The one that we have used the most is **#include**. These statements are not turned into machine language at compile time like other C++ statements. Instead they are directions to the compiler.

In the example below we will use an OpenMP directive to parallelize linear search.

```
int parLinearSearch (const int arr[], int size, int value){

   int foundAt = -1;
   omp_set_num_threads(4);

   #pragma omp parallel for
   for(int i = 0; i < size; i++){
        if(arr[i] == value) foundAt = i;
   }

   return foundAt;
}
```

This function call starts with only one thread. A thread is a series of instructions being executed on one processor. Sequential (non-parallel) code always runs on one thread. The function **parLinearSearch** first defines a new variable and sets the number of threads that OpenMP will use. It is important that instructions that should only be done once (like declaring shared variables) be written so that they only run in one thread.

The **#pragma** pre-processor directive instructs the compiler to run the for loop on 4 threads (because we manually set the number of threads to 4). This directive will only apply to the for loop, not to any code that is written after the loop. Loops are the most common sources of parallelism in code.

The return statement is outside of the for loop so it will only be executed by one thread, like the variable declaration. This is good because a function only needs to return once.

If you would like to learn more about OpenMP there are many good articles available at: www.openmp.org.

Cuda

Up until now we have only discussed code which is running on CPUs. But anyone who has ever shopped for a new computer will be aware that most computers have another kind of processor built in as well-- the GPU. These were originally built as special purpose processors for rendering graphics, but coders in the mid-2000s realized that these processors are also a powerful tool for performing large computations.

Unlike CPUs, which generally run a program on 8 to 16 threads, GPUs can run many thousands of threads in parallel. This is because GPU computing cores are much simpler but there are a lot of them. Usually a GPU will have several hundred computing cores built in. GPUs are also capable of **Single Instruction Multiple Thread** execution, where one instruction is performed on many different threads all at the same time. This makes GPUs very useful for large number-crunching tasks were many identical computations are done on many different data elements. This also makes them less useful for running the main body of programs where there are many different types of instructions being executed.

GPUs do not share a memory space with the CPUs in one machine. This means that data that is going to be processed on a GPU has to be completely copied into the memory that is accessible by the GPUs. This is a very expensive process and means that GPU computing should only be used be the **speedup** more than compensates for this extra **cost**. The extra work that has to be done to set up a block of parallel code is called the **overhead**. Every parallel code experiences some overhead and that makes it very difficult to actually achieve linear speedup in real life. Computers that use both CPUs and GPUs are called **heterogenous**.

Cuda is the system provided by Nvidia to run code on their line of GPUs. It requires many more changes to be made to code than OpenMP does. For example this implementation of Linear Search will work in Cuda:

```
void parLinearSearch (const int arr[], int size, int value, int &foundAt){

   int t_id = blockIdx.x * blockDim.x + threadIdx.x;
   if(t_id < size){
      if(arr[t_id]) foundAt = t_id;
   }
}
```

Notice that there is no loop. Instead thousands of this function will be run at the same time, each one with a different "thread ID". Each tread checks the space in the array that is assigned to it, and sets the by-reference parameter **foundAt** if it has the search value at its location. Notice that we can search an array *without a loop* because there are many thousands of threads available.

Besides requiring major code re-writes, Cuda also requires a special compiler called nvcc. For these reasons the Lab this week will focus on OpenMP rather than Cuda. Students who are interested in Cuda can find more information at: https://developer.nvidia.com/blog/even-easier-introduction-cuda/