CS 2401
Parallel Computing Lab

(100 pts.)

Introduction

Most of what you've learned so far in CS 2400 and CS2401 involves programs that execute *sequentially.* These programs execute a first step, then a second step, etc. In this lab, we will provide an introduction to programs that can execute more than one instruction at the same time. These programs will be called *parallel programs*, and, in theory, they should be able to run much faster than their sequential counterparts.

Parallel processing is becoming increasingly important in our modern society. Many modern devices (cell-phones, computers) have multiple processors (or cores). Likewise, many industrial scale computing applications (e.g., Google search), use scores of computing devices in modern data-centers. The machines that you are using also have multiple processing units (cores) within a single physical processor.

Investigating Your Lab Machine

The Solaris version of UNIX provides a command (`psrinfo`) that provides information about the number processors (both the number of physical processors as well as the number of cores). Issue the command

```
man psrinfo
```

to see how to use the `psrinfo` command.

In order to see the number of physical processors on your lab machine, issue the command

```
/usr/sbin/psrinfo -p
```

1. (10 pts.) How many physical processors are on the lab machines that you are currently working on?

Now, enter the command

```
/usr/sbin/psrinfo -v
```

2. (10 pts.) How many virtual processors are there on your lab machine?
3. (10 pts.) How fast (in terms of GHz) is each virtual processor on your lab machine?

While the `psrinfo` command provides a lot of information, it cannot distinquish between a processing "core" and a hyperthreaded virtual processor. To get that information, you'll need to use the `kstat` command.

Issue the command

```
kstat | grep core
```

and examine the output to determine how many physical cores are on your lab machine.

4. (10 pts.) How many cores are on your physical lab machine?

Exploiting Multiple Processors in C++: Multi-Threaded Applications

Next, we will see how to use these multiple CPU cores in your C++ programs to make then run faster.

First, we must discuss a little bit about how to use all of the physical cores on a machine in a single program. The way to do this is to use either multiple *processes* or multiple threads. A process is a running program. Two processes do not share the same address space. A thread is a light-weight process, and can share the same address space.

Next, we must discuss how to create a C++ program that uses multiple threads. The easiest way to do this is to use the OpenMP standard. The OpenMP standard provides a set of *compiler directives* that can be used to seamlessly create programs that execute with multiple threads. OpenMP hides many of the implementation details, and so it is easy to make mistakes. However, once you have a basic understanding of how OpenMP works, it can be very easy to write programs that use multiple threads.

The simplest compiler directive provided by OpenMP is the compiler directive

```
#pragma omp parallel for
```

If you add this compiler directive, and if you compile your code with the flag `-fopenmp`, then your program may run faster. (Remember, if you forget the flag `-fopenmp`, this compiler directive will be ignored.)

This compiler directive tells the compiler to break the following for loop into n equal size "chunks," where n is the number of processors on the machine, and to execute n separate for loops (in separate threads) to execute the for-loop. Now, notice, that not every for loop can be executed in this way, since (i) sometimes, one iteration of the for-loop may depend on previous iterations of the for-loop, and (ii) sometimes, a shared variable may appear in the for-loop. If a shared variable appears in the for-loop, and it is modified, then executing that loop using multiple threads may generate the wrong answer. Hence, the code that modifies that shared variable becomes a *critical section*, i.e., a section of code that can be executed by only one thread at a time. In OpenMP, you can denote a critical section using the compiler directive:

```
#pragma omp critical
```

Let's look at an example of how to use `#pragma omp parallel for`. can be used.

Matrix Multiplication

A matrix is simply a two-dimensional array of numbers. A single $n \times n$ matrix A has n rows and n columns. Each individual element of A is identified by it's location. So, the element $A_{i,j}$ is located in row i and column j. If we have two $n \times n$ matrices A and B, we can "multiply" these two matrices to compute a third matrix C. In particular, C is defined as follows:

$$C_{i,j} = \sum_{l=1}^{n} A_{i,k} * B_{k,j} \tag{1}$$

A straightforward implementation of matrix multiplication can be found in `labp1.cc`. An input to this program is given in the file `900.dat`.

5. (10 pts.) Compile `labp1.cc` and run it on the input `900.dat`. What is the output?

6. (10 pts.) Run your program again. This time, use the UNIX `time` command to see how long it takes. How long did it take to run?

Insert the line

```
#pragma omp parallel for
```

at line 13 in the program `labp1.cc`.

7. (10 pts.) Compile your modified program using the appropriate compiler flag. Run this new program on `900.dat`? What was the output?

8. (10 pts.) Run your program again. This time, use the UNIX `time` command to see how long it takes. How long did it take to run?

Efficiency and Speedup

If you successfully completed the previous problems, you should have noticed that second program was much faster. The ratio of the time taken by the first program (the sequential version) and the time taken by the second program (the parallel version) is called *speedup*. Ideally, the speedup that a parallel program achieves over the sequential version should be near to the number of processors utilized. The *efficiency* achieved by a parallel program is the speedup divided by the number of processors. So, for example, if a sequential program takes 10 seconds to execute and the parallel version takes 2 seconds, then the speedup is 5. If the number of processors used is 8, the efficiency is $5/8 = 62.5$ %.

9. (5 pts.) What speedup did you achieve on the parallel version of `labp1.cc`?

10. (5 pts.) What efficiency did you achive on the parallel version of `labp1.cc`?

Putting it all together

Consider the program `labp2.cc` and the input `900_2.dat`.

11. (10 pts.) Add OpenMP compiler directives to `labp2.cc` so that it executes in parallel and computes the same value as the original code. (Hint, you may need to define a *critical section*. Also, do not try to speed up reading in the input.) Check with your lab TA to see that you've done it correctly.

    Does your code run faster than the original?