CS 418, Spring 2008
Assignment 1: Intro to Parallel Programming Using the Shared Memory Model

Assigned: Thursday, Jan. 24
Due: Thursday, Feb. 7, noon

The purpose of this assignment is to get some hands-on experience with optimizing parallel program performance. Although the task which you are asked to parallelize is relatively simple, there are a number of subtle issues related to achieving high performance.

1 Policy and Logistics

Please work in groups of 2 people to solve the problems for this assignment. (Hand in one assignment per group.) There will be both electronic and hard copy hand-ins, as described below. Any clarifications and revisions to the assignment will be posted on the class assignments web page (i.e. the "*Assignment and Exam Information*" link off the main web page). In the following, *HOMEDIR* refers to the directory:

`/afs/cs.cmu.edu/academic/class/15418-s08/public`

and *ASSTDIR* refers to the subdirectory *HOMEDIR*`/asst/asst1`.

2 Written Assignments

Please answer problems 2.6 and 2.7 (parts a, b, and d only) in the textbook.

3 Coding Assignment: Parallelizing a Prime Number Generator

The file *ASSTDIR*`/primes.c` contains a uniprocessor program written in C for determining prime numbers. The program works by testing each odd number (up to a specified limit) for divisibility by all of the factors from 3 to the square root of that number. The algorithm is not very smart, but it is easy to parallelize.

Your assignment is to parallelize this existing algorithm in a number of different ways to run on an SGI Altix system at NCSA and an HP AlphaServer system at PSC. Note that you should *not* change the underlying algorithm. Instead, you should view it as a workload that is to be parallelized.

The program takes two main parameters, which are read in from the command line:

P: the number of processors (`numProcs` in the code); and

N: the problem size (`size` in the code).

In addition, the program takes arguments that output all of the primes generated, either to a file or standard out. This should assist you in ensuring that the parallel version of the algorithm works correctly. Precise usage instructions can be seen by running `./primes -h`.

The main data structure of the program is an array which holds boolean values indicating whether the corresponding numbers are prime. The array only holds odd numbers, since no even numbers (except 2)

are prime. The core code is executed 200 times to provide a reasonable runtime. You should parallelize only the actual core code, not this outer loop! Also, you should not change this data structure.

You are to parallelize these applications using OpenMP to create a shared-memory application. Further details on how to use the OpenMP can be found in *ASSTDIR*/`openmp_tutorial.pdf`, along with example C programs in *ASSTDIR*/`examples`.

We would like for you to implement two different versions of the parallel code and compare their performance:

1. **Statically scheduled:** each processor (thread) is assigned a fixed number of loop iterations to work on.
2. **Dynamically scheduled:** the loop iterations are divided up between processors (threads) dynamically at run-time.

The trick is to achieve good "load balancing" with minimal overhead while achieving good memory performance. You should be able to achieve very good speedups on this assignment (i.e. close to linear).

Please use `#defines` and `#ifdefs` if possible so that there is only one source file (i.e. the static version should be generated if `STATIC` is defined, and the dynamic version should be generated if `DYNAMIC` is defined).

4 The Parallel Machines

You will collect your numbers on the NCSA SGI Altix machines (named "*Cobalt*") and the PSC HP AlphaServer machines (named "*Rachel*"). You have each been given accounts on these machines. (If you do not pick up your login information form in class, see Prof. Mowry immediately to get it.) Note that you must sign and return the "responsibilities" form in order to continue using the account.

To develop, compile, and debug your code, as well as making sure that it gets reasonable performance on small numbers of processors, you should `ssh` into the front-end of *Cobalt*, which is called `cobalt.ncsa.uiuc.edu` or the front-end of *Rachel*, which is called `rachel.psc.edu`. These machines are reserved for running interactive tasks. Note that in addition to using `ssh` to log into these machine, you can also use `klog` to access your AFS directories here at CMU from *Cobalt*. (*Rachel* will not permit you to access your CMU AFS space) You can run jobs on the front-end machine from AFS; however, do not run a batch job from your AFS directories. The batch system does not handle foreign AFS cells properly. Instead, you should copy any AFS files onto the local filesystems on these machines to execute your code and collect your performance results. *Cobalt* is also larger system than *Rachel*, so it supports running larger experiments and may give faster response times for smaller experiments. For these reasons, you will likely find it to be easier to develop your application primarily on *Cobalt*. For more information on using these machines, please see the following document: *ASSTDIR*/`ncsa_psc_tutorial.pdf`

5 Measuring Performance

While it may be helpful to compile the `-g` flag when you are debugging your code, **please be sure to use the `-O` flag to generate any programs that you will be timing!** There can be significant differences in performance between `-g` and `-O`, and we are only interested in speeding up optimized code.

To evaluate the performance of the parallel program, measure the following times using `gettimeofday` (see *ASSTDIR*`/examples/sqrt.c` for an example):

1. *Initialization Time:* the time required to do all the sundry initialization, read the command line arguments, and create the separate processes. Start timing when the program starts, and end just before the main computation starts.

2. *Computation Time:* this is strictly the time to compute the answer. Start timing when the main computation starts (after all the processes have been created), and finish when all of the answers have been calculated.

Note that: *Total Time* = *Initialization Time* + *Computation Time*. *Speedup* is calculated as $\frac{T_1}{T_p}$, where T_1 is the time for one processor, and T_p is the time for P processors. *Computation Speedup* uses only Computation Time, and *Total Speedup* uses the Total Time.

6 Performance Analysis

The goal of this assignment is for you to think carefully about how real-world effects in the machine are limiting your speedup, and how you can improve your program to get better performance. If your performance is disappointing, then it is likely that you can restructure your code to make things better. I am especially interested in hearing about the thought process that went into designing your program, and how it evolved over time based on your experiments.

Your report should include the following items:

1. A detailed discussion of the design and rationale behind your approach to parallelizing the algorithm.

2. The number of primes, and the last prime for N = 200,000; 1,000,000; 2,000,000; and 5,000,000.

3. A plot of the *Total Speedup* and *Computation Speedup* vs. *Number of Processors* (P) for N = 200,000; 1,000,000; 2,000,000; and 5,000,000. Use P = 1, 2, 4, 8, 16, 24, and 32 on *Cobalt* and P = 1, 2, 4, 8, 16 on *Rachel*.

4. Discuss the results that you expected and explain the reasons for any non-ideal behavior that you observe.

5. A plot of the *Total Speedup* and *Computation Speedup* vs. N for P = 16. N = 200,000; 1,000,000; 2,000,000; and 5,000,000. (A larger value of P is also acceptable, and may be more interesting.)

6. Discuss the impact of problem size on performance.

7 Hand In

Electronic submission:

- Your version of `primes.c`. Do this by naming your file *lastname*`-primes.c`, where *lastname* is the last name of one of your group members, and copying this file to the directory

  `/afs/cs.cmu.edu/academic/class/15418-s08/public/asst/asst1/handin`

Include as comments near the beginning of this file the identities of all members of your group. Also remember to put comments in your code.

Hard-copy submission:

1. Answers to written problems specified in Section 2.
2. Answers to items listed in Section 6.
3. A listing of your code.