CS 740, Fall 1998
Assignment 5: Parallel Programming

Assigned: Tuesday, Nov. 10
Due: Thursday., Nov. 19, 1:20pm

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

You may work in a group of up to 3 people in solving the problems for this assignment. There will be both electronic and hard copy hand-ins, as described below. Any clarifications and revisions to the assignment will be posted on Web page `assigns.html` in the class WWW directory. In the following, *HOMEDIR* refers to the directory:

`/afs/cs.cmu.edu/academic/class/15740-f98/public`

and *ASSTDIR* refers to the subdirectory *HOMEDIR*`/asst/asst5`.

2. Parallelizing a Prime Number Generator

The file *ASSTDIR*`/primes.U` 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 Origin multiprocessor. 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 parameters, which a read in from the command line:

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

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

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 the ANL macros to create a shared-memory application. Further details on how to use the ANL macros can be found in *ASSTDIR*`/ANL_tutorial.ps`, along with an example program in *ASSTDIR*`/example.U`.

To compare performance, you should implement two different versions of the parallel code:

1. Statically scheduled loop: the programmer statically determines which loop iterations will be executed by which processor. (This is usually computed based on the processor ID of the given processor.)

2. Dynamically scheduled loop: the loop iterations are divided up between processors dynamically at run-time. (Note that the ANL macro "`GETSUB`" can be used to help implement this approach.)

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.

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).

3. The NCSA Origin 2000 Machines

You will collect your numbers on the NCSA Origin 2000 machines. 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 the "responsibilities" form and give that back to use in order to continue using the account.

To develop your code, compile it, and make sure that it gets reasoanble performance on small numbers of processors, you should use the Origin machine at NCSA called `modi4.ncsa.uiuc.edu`. This machine is reserved for running interactive tasks. Note that you can use `ssh` to log into this machine, you can use `klog` to access your AFS directories here at CMU. You will want to use the local filesystems on these Origin machines to collect your actual results.

Information on the NCSA machines and how to use them can be found at the following URL:

- `http://www.ncsa.uiuc.edu/SCD/Hardware/Origin2000`.

In particular, look under the "*NCSA SGI Cray Origin2000 user documentation*" and "*Frequently Asked Questions*" links.

Here are some handy URLs on how to run jobs on the Origins:

- `http://www.ncsa.uiuc.edu/SCD/Hardware/Origin2000/Doc/Jobs.html`
- `http://www.ncsa.uiuc.edu/SCD/Hardware/CommonFAQ/SGIJobs.html`

To measure your final performance numbers, you will want to use the LSF batch queueing system. (Note that the interactive machine only allows jobs with up to 8 processors.) In particular, you will be `vst_sj`, `vst_mj`, and `vst_lj` "lsbatch" queues to run small (i.e. 1-8 processors), medium (9-16 processors), and large (17-64 processors), respectively.

4. Using the ANL Macros

As mentioned earlier, there is a tutorial on how to use the ANL macros in *ASSTDIR*/`ANL_tutorial.ps`. Section 2.3.5 of the Culler, Singh, and Gupta book on "Parallel Architecture" shows an example of how to use somewhat similar primitives, but you should not necessarily trust their syntax.

An example of how to compile and run these applications can be found in the *ASSTDIR*/`example_sgi` directory. In addition, the *ASSTDIR*/`example_uniprocessor` directory contains an example of how a program with ANL macros can be compiled to run on a uniprocessor. When you are debugging your code, it is a very good idea to first make sure that it works correctly on a uniprocessor before you start running it on a multiprocessor.

4.1. Some Important Pitfalls to Avoid

These things are discussed in the ANL macros tutorial, but they are worth repeating.

4.1.1. Debugging

Although you may want to compile your applications using the "`-g`" option to enable symbolic debugging, be sure to *turn off* this flag when you compile your applications to time them. You should compile your applications using the "`-O`" compiler flag to produce timing numbers.

4.1.2. Kill Those Stray Processes!

When your parallel programs crashes, the associated processes will not necessarily terminate. Always check to see if any processes are still running by typing "`ps -ef | grep <yourLoginName>`". If so, kill the associated processes by typing the following for each of them: "`kill -9 <processID>`".

4.1.3. Free Those Shared Memory Segments!

On an SGI machine, you must be very careful to release any shared memory segments that your programs reserve. If your program always executes "`MAIN_END`" when it exits the program, then you should be ok. NOTE: do *not* exit by simply saying "`exit(-1)`", since this does *not* free the shared memory segment.

You should make a habit of frequently checking whether you have any shared memory segments outstanding. To do this, type "`ipcs`" on an Origin machine. This will list whether any memory or synchronization segments are still outstanding. To free them, type "`ipcrm -m <segmentID>`" for each of them.

5. Measuring Perfomance

To evaluate the performance of the parallel program, measure the following times using the ANL `CLOCK` macro:

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 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 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 = 5000, 10000, 50000, and 100000.
3. A plot of the *Total Speedup* and *Computation Speedup* vs. *Number of Processors* (P) for N = 5000, 10000, 50000, and 100000. Use P = 1, 2, 4, 8, 16, 24, and 30 on the NSCA Origin 2000 machines. (If you can collect numbers on larger numbers of processors, that is great, but it is not a requirement.)
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 = 5000, 10000, 50000, and 100000. (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.U`. Do this by naming your file *last*`-primes.U`, where *last* is the last name of one of your group members, and copying this file to the directory

  ```
  /afs/cs.cmu.edu/academic/class/15740-f98/public/asst/asst5/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 items listed in Section 6.
2. A listing of your code.