CS 4444/6444 Fall 2011

Assignment 3: Heated Plate with Pthreads

Due by 5:00 PM on Friday, October 14th in Andrew's mailbox

Introduction

Your task in this assignment is to implement a parallel version of the heated plate problem using Pthreads. You will run your parallel program on Blacklight, a shared-memory supercomputer at the Pittsburgh Supercomputing Center.

Files needed

To start, download from the course Collab site the following files:

- heated_plate.c or heated_plate.f: this is the file that you will need to parallelize. Pick your flavor depending on whether you want to code in C or Fortran.
- ground_truth.ppm: a snapshot generated by the sequential implementation of a 10,000 x 10,000 heated plate after 10,000 iterations. Use this to verify that your parallel implementation is producing the correct results.
- create_jpegs.pl: run this Perl script to convert PPM snapshots of the heated plate into JPEG images.

The program

The program in this assignment models the flow of heat through a two-dimensional plate. The plate is divided up into a grid of cells, and the temperature of each cell is computed in discrete time steps. The interior cells are all initialized to the same temperature (50 degrees), and the border cells are fixed at a specific temperature (0 degrees along the top and left sides and 100 degrees along the bottom and right sides). In each time step of the simulation, the temperature of each cell is computed by averaging the temperatures of the four neighboring cells in the previous time step.

The original version of the program provided to you is sequential; all of the computation is performed by a single thread. After simulating a specified number of iterations, the final state of the plate is output as an image in the PPM format. You can use the script provided (create_jpegs.pl) to convert the PPM image(s) into JPEG format for easier viewing.

To run the program, specify the size of the plate and the number of iterations as follows:

```
./heated_plate <columns> <rows> <iterations>
```

For example, to simulate a 500 x 500 plate for 1,000 iterations, use the following command:

```
./heated_plate 500 500 1000
```

The command-line arguments are optional; default values will be used if they are missing.

The task

The task in this assignment is to parallelize the sequential version of the program using Pthreads. The number of threads used by your program should be controllable via a command-line argument. Please use the snapshot output from the sequential implementation to verify that your parallel implementation is executing correctly. We have provided the snapshot output for a 10,000 x 10,000 plate after 10,000 iterations (ground_truth.ppm) so you can verify correct execution of your final runs. For testing of your initial (shorter) runs, you can use the sequential implementation to generate your own PPM files to compare against.

Once you have a working Pthreads implementation, measure the performance impact of increasing the number of threads (and the number of processor cores). You should measure performance for thread counts of all powers of 2 from 2 through 128 (2, 4, 8, 16, 32, 64, and 128). Make sure to request at least as many processors (*ncpus* in PBS) as threads. Processors must be requested in multiples of 16 on Blacklight; for example, when running with 8 threads you would request 16 processors.

To save time, you do not need to measure the performance of the sequential implementation. You can instead use the value that we measured (53 minutes) when computing your speedup.

When measuring the final execution times that you will provide in your report, use the following parameter values: a grid size of 10,000 x 10,000 and 10,000 iterations. For initial testing, you will most likely want to use a smaller number of iterations.

Blacklight

For initial testing, feel free to use any system available to you. The performance measurements provided in your report, however, must be obtained on Blacklight. More information about using Blacklight (how to log in, transfer files, compile, submit jobs, etc.) can be found here:
http://www.psc.edu/machines/sgi/uv/blacklight.php

When submitting jobs to be run on Blacklight via PBS, it is very important to provide an accurate walltime request. For the data size we are using, the single-threaded implementation takes roughly 53 minutes; assuming your Pthreads implementation is no slower than the sequential implementation (not always a reasonable assumption, unfortunately), your walltime request in your PBS script should probably never exceed one hour (60 minutes). There are two reasons for this. First, if your program deadlocks or contains an infinite loop, you do not want your program to run indefinitely (and thereby waste a large portion of the compute time allocated to the class). Second, smaller walltime requests may result in your job being scheduled sooner, as described in the web site linked above. For the latter reason, when submitting multiple jobs, it is probably beneficial to submit each job in a separate PBS script, rather than combining all of the jobs into a single script.

Optimization

A straightforward Pthreads implementation of this problem running with 16 threads only achieves roughly a 4.2x speedup over the sequential implementation. Performance is even more miserable with 128 threads, with the speedup dropping to 2.5x. To improve the scalability of this program, we need to modify our code to take into account the fact that Blacklight is a NUMA machine.

Recall that Blacklight is composed of a large number of blades, each containing two CPU sockets. Each socket contains 8 CPU cores and has its own physical memory. If we allocate all of our memory at once using `malloc` (as in the sequential implementation), it is likely to all end up residing in the memory attached to a single socket. This will result in a severe memory bottleneck as all of the memory accesses from all of the threads get routed to only one physical memory.

Ideally, as we increase the number of cores/blades allocated to our program, we would like to spread the memory accesses across all of the available physical memories. One approach to achieving this is to leverage some of the functions provide in the *libnuma* library (especially the various `numa_alloc_*` functions). More information on the library can be found here:
http://linux.die.net/man/3/numa
http://www.halobates.de/numaapi3.pdf

What to turn in

Please turn in a written report that adheres to the guidelines specified in the HomeworkReportGuidelines.pdf file on Collab. Do not forget to attach your source code.

Turn in a hard copy of the report and source code to Andrew's mailbox in the mail room on the 5th floor of Rice Hall by 5:00 PM on Friday, October 14th.