CS 758 Course Wiki: Fall 2016 Main » Homework 1

CS 758: Programming Multicore Processors

Homework 1

You should do this assignment alone. No late assignments.

Filelist for the assignment:

- Woo et. al. SPLASH-2 paper
- Intel's Closing the Ninja Gap paper
- Dan Gibson's OpenMP intro
- Dan Gibson's OpenMP examples
- Template files

The purpose of the assignment is to give you experience writing simple shared memory programs using OpenMP. This exercise is intended to provide a gentle introduction to parallel programming and will provide a foundation for writing and/or running much more complex programs on various parallel programming environments.

While you are supposed to use the ale machines to get your code working, for the final speedups and other results, you have to use our new Xeon Phi system. Instructions for usage: Instead of creating individual accounts on our new Xeon Phi (Knights Landing) machine (godel.cs.wisc.edu), we have developed a lightweight job queueing system. This system allows you to submit your jobs, which will be run individually on godel by an automated user, and receive logs of the stdout and stderr of the execution.

Before you schedule your job on godel, make sure that it works on a machine in the instructional labs or the ale nodes. If you face any major issues, please post on piazza with sufficient details.

Follow the following steps to schedule an execution on godel:

1) Create an executable of your program. While the software environment on godel is not exactly the same as on the CSL instructional machines, there shouldn't be any major issues that you won't be able to debug.

2) Create a runscript named 'run.sh' containing the command lines needed to launch your program. Use the following command to give it execute permissions.

```
chmod a+x run.sh
```

3) Make sure the program only writes output to stdout, or to any file in its current working directory. Also, ensure that ‘net:cs’ has read permissions for any input files. Check here for how to do this, using listacl/setacl (https://csl.cs.wisc.edu/services/file-storage)

4) Place the executable, run.sh, and any other input files required by your program into this location - /p/course/cs758-david/public/godel/<your_cs_username>

5) Check the contents of the above folder after a minute or so (perhaps ten minutes on the day before the due date). A file named 'running' will be created if your job was picked up and is currently being run. If your program was completed, 'run.sh' will be replaced by 'ran.sh' and two log files - stdout and stderr will be created in the same folder.

6) Using the contents of stdout and stderr, debug any errors, and repeat steps 1-6.

Common Errors - Required permissions are missing from the executable or other input files. Program executes for more than 1 minutes. The scheduler enforces a strict timeout of 1 minute and kills any program running longer. This is because every program is individually run on the machine, and we want to keep response times small for everyone. Some difference in the software environment of our machine and CSL labs. Post such problems on piazza for the TAs to fix.

The original version of this assignment and the reference OpenMP programs were developed by Dan Gibson for CS838 offered in Fall 2005.

OpenMP

OpenMP is an API for shared-memory parallel programming for C/C++ and Fortran. It consists of preprocessor (compiler) directives, library routines and environment variables that determine the parallel execution of a program.

For this assignment, you will use the GNU implementation of OpenMP that is already installed on the ale nodes. A set of sample OpenMP programs are available here. A presentation on using OpenMP by Dan Gibson is available here. It is strongly recommended that you download and run them to get a hands-on experience of compiling, linking and running OpenMP programs. Remember to

- Include omp.h in all the source files that use OpenMP directives or library calls.
- Use the flag -fopenmp for compilation and linking of your source files.

Programming Task: Ocean Simulation

OCEAN is a simulation of large-scale sea conditions from the SPLASH benchmark suite. It is a scientific workload used for performance evaluation of parallel machines. For this assignment, you will write two scaled-down versions of the variant of the Ocean benchmark.

Ocean is briefly described in Woo et al. on the Reading List. The scaled-down version you will implement is described below.

Our version of Ocean will simulate water temperatures using a large grid of **integer** values over a fixed number of time steps. At each time step, the value of a given grid location will be averaged with the values of its immediate north, south, east, and west neighbors to determine the value of that grid location in the next time step (total of five grid locations averaged to produce the next value for a given location).

As illustrated in Figure 1, value calculations for two adjacent grid locations are not independent. Specifically, the value of the grid location number 6 is calculated using the values of the grid locations 6, 2, 7, 10, and 5, and the value of the grid location number 10 is calculated using the values of the grid locations 10, 6, 11, 14, and 9. Because 6 depends on 10 and 10 depends on 6, the resulting values for 6 and 10 will depend on the order in which the values for 6 and 10 were calculated.

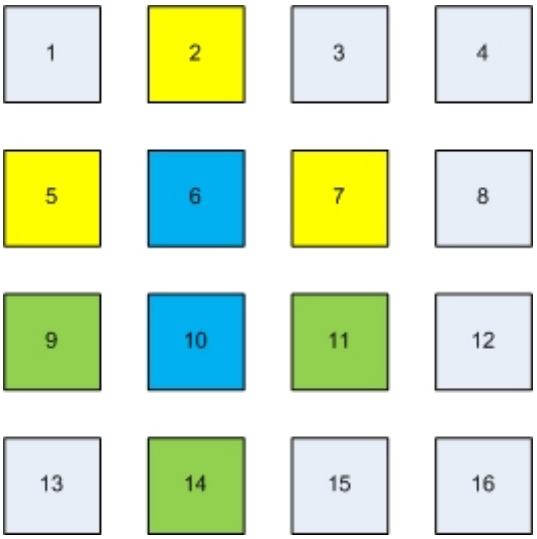

Figure 1: Calculation Dependence

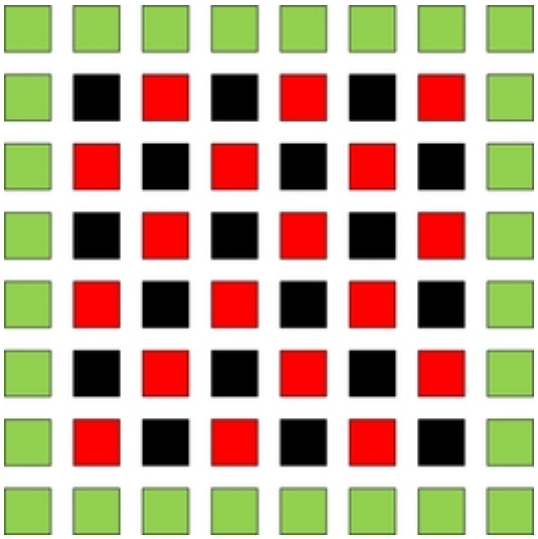

Figure 2: Red and Black Independent Sets

The grid points in the ocean grid can be separated into two independent subsets shown in red and black in Figure 2. Instead of calculating a new value for each grid location at every time step, the values for the grid points in the red subset are updated on even time steps and the values for the grid points in the black subset are updated on odd time steps.

The edges of the grid shown in green in Figure 2 do not participate in the averaging process (they contribute a value, but their value does not change). Thus, Ocean will converge (given sufficient runtime) to a gradient of the water temperatures on the perimeter of the grid.

Template for your convenience for HW1

A template is provided for your convenience with basic setup of the program. This template provides you the initialized data structure to work with. The timing instrumentations are also provided. You are free to move the timing instrumentations in case you parallelize the initialization phase. A makefile has also been added to the template that will help you in compiling the code. You might need to change the flags in the makefile during debug phase. Do revert them back to allow compiler optimizations.

IMPORTANT: You are free to not use the template. But do keep the initialization section the same, so that you have a 32-bit random number.

Download the template from: here

Problem 1: Write Sequential Ocean (10 points)

Write a single-threaded (sequential) version of Ocean as described above. This version of Ocean must take three arguments: the x-dimension of the grid, the y-dimension of the grid, and the number of time steps. You may assume for simplicity that all grid sizes will be powers of two plus two (i.e. (2^n)+2); therefore the area of the grid that will be modified will be sized to powers of two (+2 takes care of the edges that are not modified).

You are required to make an argument that your implementation of Ocean is correct. A good way to do this is to initialize the grid to a special-case starting condition, and then show that after a number of time steps the state of the grid exhibits symmetry or some other expected property. You need not prove your implementation's correctness in the literal since. However, please annotate any simulation outputs clearly.

Your final sequential version of Ocean should randomly initialize a grid of the requested size, then perform simulation for the specified number of time steps.

Problem 2: Write Parallel Ocean using OpenMP (10 points)

For this problem, you will use OpenMP directives to parallelize your sequential program. You are required to use the **schedule(dynamic)** clause on loops that you will parallelize with OpenMP. This will cause loop iterations to be dynamically allocated to threads. Please be sure to explicitly label all appropriate variables as either shared or private. Make an argument for the correctness of your implementation (it is acceptable to use the same argument as problem 1, provided it is still applicable).

The number of threads will be passed to the program based on an environment variable (OMP_NUM_THREADS). To pass an environment variable to you program in a one-off fashion you can prepend it to the command line like:

```
>>> OMP_NUM_THREADS=4 ./omp_ocean 514 514 100
```

For simplicity, you may assume that the dimensions of the grid are powers of two plus two as before, and that only N=[1,2,4,8,16] will be passed as the number of threads.

Problem 3: Analysis of Ocean (15 points)

Modify your programs to measure the execution time of the parallel phase of execution. Use of Unix's gethrtime() is recommended.

Compare the performance of your two Ocean implementations for a fixed number of time steps (**100**). Plot the normalized (versus the Sequential version of Ocean) speedups of your implementations on N=[1,2,4,8,16] threads for a **514x514** ocean. Note that the N=1 case should be the Sequential version of Ocean, not the parallel version using only 1 thread. Repeat for an ocean sized to **1026x1026**.

Problem 4: Parallelization using static partitioning of the grid (15 points)

In problem 2, you used OpenMP's automatic parallelization capabilities. However, in order to get good data locality, you might want to statically partition the Ocean grid into different regions and assign each region to a separate thread.

In this problem, you will need to implement a parallel version of Ocean that works on statically partitioned regions of data. In each time step, a thread would only update the grid cells belonging to its region.

Plot the speedups obtained with the static sheduling compared to the speedups obtained using dynamic implementation from problem 2. For N=8 threads and 100 time steps, use various Ocean sizes to identify the grid sizes where static partitioning performs better than dynamic partitioning. Present arguments on why you think this is the case. Vary the chunk size for static scheduling, plot and explain trends.

Problem 5: Architecture specific optimizations (Only required if you took 757 with Professor Hill in Spring 2016. Optional for all other students.)

After writing the serial and parallel versions of ocean, write another, optimized_ocean, which contains some architecture focused optimization. Examples of this include, blocking the algorithm temporally or spatially to take advantage of caches, optimizing for special execution units like SIMD/SSE, and optimizing data layout for memory bandwidth. Closing the Ninja Gap, a recent paper out of Intel, describes these examples applied to many different algorithms and may help you get started.

After implementing optimized_ocean, briefly (2 to 4 sentences) describe your optimization and how you applied it to ocean. Plot the speedups obtained with you new optimization over sequential ocean for the same conditions as in Problem 3. Also, for the same number of threads, plot the speedup of your optimized version over the un-optimized parallel ocean (from Problem 3 or 4). Also include the overall "best" speedup obtained of your most optimized version of ocean compared to the serial version. **Note: We do not expect you to get 2x, or even 1.5x speedup here. In fact, getting no speedup at all is a perfectly fine result. Your answers to the questions and explanation for your results is what is important.**

What to Hand In (For most of the class)

Please turn this homework in **on paper** at the beginning of lecture. Your answers to the discussion questions should be **typed up**, no handwritten notes will be accepted. A tarball of your entire source code including a Makefile and a README file, should be emailed to the professor before the beginning of lecture. The README should include 1) directions for compiling your code, 2) directions for running your code, 3) any other comments. Use subject line `[CS758] Homework 1`, so that email filters work properly.

- A printout of the source code for the simulation phase of Sequential Ocean (this is probably just a for loop).
- A printout of the source code for the parallel phase of OpenMP Ocean (this is the code that is parallelized using **schedule(dynamic)**) and a concise description of how you parallelized the program.
- Arguments for correctness of the two programs.
- The plots as described in Problem 3 & 4 including a detailed explanation of the observed trends. Specifically, explain 1) differences between the slopes for the two grid sizes, 2) any changes in the slope for each of the grid sizes separately, 3) sources of superlinear speedups, 4) sources of sub-linear speedup.
- Optionally, include the plots described in Problem 5, and detailed explanations of the observed trends. Concentrate on how these plots differ from the ones you generated for Problems 3 and 4.

What to Hand In (If you took 757 from Professor Hill in Spring 2016)

Please turn this homework in **on paper** at the beginning of lecture. Your answers to the discussion questions should be **typed up**, no handwritten notes will be accepted. A tarball of your entire source code including a Makefile and a README file, should be emailed to the professor before the beginning of lecture. The README should include 1) directions for compiling your code, 2) directions for running your code, 3) any other comments. Use subject line `[CS758] Homework 1`, so that email filters work properly.

- A printout of the source code for the simulation phase of Sequential Ocean (this is probably just a for loop).
- A printout of the source code for the parallel phase of OpenMP Ocean (this is the code that is parallelized using **schedule(dynamic)**)
- A printout of the source code for the optimized version of Ocean.
- The plots as described in Problem 3 & 4.
- A description of your optimization for Problem 5.
- The plots described in Problem 5, and detailed explanations of the observed trends. Concentrate on how these plots differ from the ones you generated for Problems 3 and 4.

Important: Include your name on EVERY page.

Tips and Tricks

- Start early.
- Set up RSA authentication to save yourself some keystrokes. HowTo.
- Make use of the demo programs provided.

- You can use /proc/cpuinfo to learn many useful characteristics of your host machine.
- Run your programs multiple times to get accurate time measurements. This will help avoid incorrect results due to interference with other user's programs.
- While making your measurements do take care that no-one else is running his/her program on the machine. Otherwise, it will provide you as well as the other person wrong results as well as longer runtimes.
- Do not wait until the last day to run the experiments. You might not get time on the machine to get good results. Hint: The best time to run your experiments is 12-6am???

Page last modified on September 13, 2016, at 09:24 AM, visited times