CS/ECE 757 Course Wiki Main » Homework 1

CS/ECE 757 Parallel Computer Architecture (Spring 2016)

Homework 1

Due Wednesday 2/05/2016

Point-of-contact Chris Feilbach <crf@cs.wisc.edu> (for questions and handin)

You should do this assignment in groups of two. 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.

You will do this assignment on **ale-02.cs.wisc.edu**, **ale-03.cs.wisc.edu**, **ale-04.cs.wisc.edu**, **ale-05.cs.wisc.edu** or **ale-06.cs.wisc.edu** - each containing two Xeon x5550 processors which have 4 cores each for a total of 8 cores (16 threads). We have given you individual accounts on this machine. Use them only for 757 homework assignments and, with instructor's permission, for your course project. The accounts and storage will be deleted at the end of the semester unless you obtain permission from the instructor for extending them. **Let the point-of-contact (see top of page) know by 2/01/2016 if you do not have access to any of these machines.**

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.

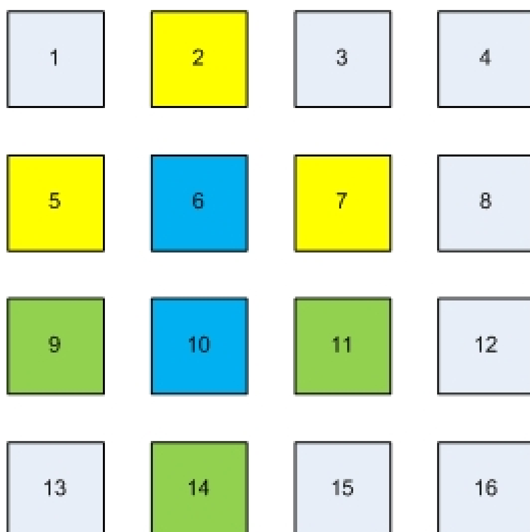

Figure 1: Calculation Dependence

To avoid this problem, two grids of the same size will be created. During an iteration, values will be read from the first grid, and then the new value will be stored in the second grid. After all values have been computed, the grids are swapped and the process is repeated.

Since the border around the grid does not change, 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 and Develop an Analytic Model (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 sense. 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 4098 4098 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)

Be sure your programs measure the execution time of the region of interest (the ocean computation), and not the initialization phase. If you are using the template provided this is already done for you.

Compare the performance of your two Ocean implementations (parallel and serial) 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 **4098x4098** 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 **8194x8194**.

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. Use OpenMP's schedule directive to statically schedule to Ocean grid so that in each time step, a thread would only update the grid cells belonging to its region.

Plot the speedups obtained with the static scheduling 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.

What to Hand In

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 **point-of-contact** (see top of page) 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 `[CS/ECE 757] 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.

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.
- Strongly consider using your own machine for verifying your program functions as specified, and the ale clusters for collecting performance results.
- 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 January 19, 2016, at 04:29 PM, visited times