CS 420: Introduction to Parallel Computing for Scientists and Engineers

Fall 2013

Machine Problem 1 – OpenMP

Due date: October 2, 5:00 PM

Background

For this MP you are given serial code for several problems which you must parallelize using OpenMP. The goals are for you to learn to recognize potential parallelism in code, avoid hazardous situations when sharing memory, and familiarize yourself with various OpenMP pragmas. Your code will be graded on both correctness and speed. Be sure to use the serial code given to you to check your answers.

System

You can use any computer with a multi-core processor to develop the code for this assignment, but use Taub to get the running times for this assignment. The code that you will be editing is written in C. Both EWS and Taub have the `gcc` compiler with OpenMP available. To compile a program `foo.c` that uses OpenMP, use the command:

```
gcc –fopenmp –o foo foo.c
```

Use the included Makefile to automate your compilation. Use the optimization level -O0 when compiling the code. Each program will have a section of code surrounded by the comments //BEGIN MAIN ROUTINE and //END MAIN ROUTINE. These are the sections of code you want to time and focus on parallelizing.

Taub contains dual 6-core processors. Run tests using different numbers of processors from 1 to 12 (1, 2, 4, 6, 12 are good core counts to test) when timing the code to test performance. You can set the number of OpenMP threads to use by setting the environmental variable OMP_NUM_THREADS. On Taub using the bash shell, this can be done with the command "`export OMP_NUM_THREADS=2`" (or 3, 4, 5, etc).

You will need to use a compiler that implements OpenMP version 3.0 to experiment with optimizations using the collapse directive, such as gcc version 4.4 or newer, which is available on Taub.

There is no special command needed to run an executable compiled with OpenMP. You simply use "`./foo`".

Each program will output a file with the results. You can compare the parallel results to the serial results to verify that you are getting the correct answer. You may find the `diff` command useful for comparing output files, and the Ctrl+C key combination useful if the terminal is non-responsive in the middle of a `diff`. You can change `#define SAVE` from 1 to 0 to make the program run faster once you are sure that your solution is correct.

Assignment

You will parallelize the following programs using OpenMP. The source code can be downloaded from the course wiki.

Program 1: Vector Addition (`addition.c`)

This is a basic vector addition code. Modify the code to sum the two vectors in parallel. Experiment with OpenMP pragmas such as schedule and any other pragmas that you think may improve the performance of the code. Comment on the results for different variations.

Program 2: Matrix Multiplication (`multiply.c`)

This code multiplies two matrices using triple nested `for` loops. Develop at least two parallelizations using only 1 parallel `for` loop. Also experiment with different nested parallelizations. Create one parallelization using the collapse directive and a second using pragmas on multiple levels of the nested loop.

Program 3: Matrix Transpose (`transpose.c`)

This code transposes a matrix. Similar to the matrix multiplication code, develop at least two parallelizations using only 1 parallel `for` loop. Also experiment with different nested parallelizations. Create one parallelization using the collapse directive and a second using pragmas on multiple levels of the nested loop.

Program 4: Tiled Matrix Multiplication (`tiled.c`)

This code takes the original matrix multiply and uses an optimized block implementation. Find at least one optimization you can make to the serial code to improve the serial performance. Similar to the matrix multiplication

code, develop at least two parallelizations using only 1 parallel `for` loop. Also experiment with different nested parallelizations. Create one parallelization using the collapse directive and a second using pragmas on multiple levels of the loop. Experiment with different block sizes and discuss the impact of block size on performance.

Each of the programs has been written in a straightforward, serial manner. You must decide which OpenMP pragmas you will use, where you will put them, and which variables should be shared or private. It may sometimes be useful to create additional variables or change the order of loops. Other than those changes, you should not significantly change the structure of the program.

Some constants are defined at the top of the programs. You can change them for testing purposes, with two caveats:

1. Change them back before submitting the code to ensure there are not any errors when grading your program.
2. Be VERY careful when changing the size of matrices or vectors (typically called `n`). The matrix-matrix multiplication algorithm has cubic complexity. If you multiply the dimensions by 10, your code will take 1000 times as long.

Time your code using the `get_clock` routines provided. In particular, time each time-consuming parallel section of code.

4-Credit Hour:

For students signed up for four credit hours, you will have one additional problem.

1. Develop a tiled version of the `transpose.c` example (called `tiled_transpose.c`) similar to the `tiled.c` matrix multiply example. Make any changes anywhere in the code necessary to make the loop that computes the transpose run as fast as possible. Discuss multiple different improved serial optimizations and improved parallel implementations. Experiment with different block sizes and discuss the impact of block size on performance.

Submission

Create a report (`netID_mp1.pdf`) which contains:

- A table showing the average running time of each program as well as the calculated speedup of each program: $\frac{Time\ on\ 1\ processor}{Time\ on\ p\ processors}$. Include the timings for the serial code as well as different parallel

implementations of the code in this table.

- Use varying core counts from 1 to 12 to test the OpenMP code (pick interesting core counts such as 1, 2, 4, 6, and 12).
- A brief discussion of reasons for the performance.
- A brief explanation of your choices in parallelizing the programs and why your choices result in the best possible performance. Be sure to include any discussion for each program requested above.
- Submit the code for all of your parallel implementations, but comment out the code for all implementations except for the fastest. In order words, when you run the code, only your fastest parallel implementation should be executed.

Copy this report, all of your parallel source code, and your Makefile into a folder named mp1. Either zip the folder (Windows) or `tar` and `gzip` it (Linux). Be careful when using `tar` as you can accidentally overwrite your files if you put the arguments in the wrong order. Submit your .zip or .tar.gz file to the Subversion repository. The timestamp given by Subversion will be used to determine whether an assignment is on time, so be sure to upload it by the deadline.

If you are unsure of how to create a .pdf file, most popular word processors (MS Word, LibreOffice, etc.) have the functionality built in.