CS484: Parallel Programming - Spring 2017
MP1
Due Date : Wednesday, February 22, 11:59PM
Submission Method : SVN

February 17, 2017

1 Introduction

The first part of this assignment introduces you to PAPI, which is a performance monitoring tool that provides access to some of the hardware counters available on processors. If the processor allows access to its counters, PAPI can be used to measure various statistics regarding your program, including instruction loads, stores, cache accesses and memory accesses. We have provided you with a quick reference manual for PAPI, that covers everything you need for this MP. A more detailed handout can be found on the PAPI website You can add PAPI to your environment in Taub with the **module load papi** command.

The second part of this MP is an introduction to shared memory programming using OpenMP. You have to implement a parallel pipelined version of the Gauss-Seidel algorithm for solving a linear system of equations. You will then compare this implementation to a parallel diagonal implementation and a baseline sequential version. You might find the following link useful: using #pragma omp flush.

2 Assignment

2.1 Matrix Transpose

Write a program to compute the transpose of a matrix, preferably in C/C++ and measure its performance using PAPI. We have provided you with the necessary functions for PAPI. Feel free to add more counters if needed. Stub code with comments is provided to you in the folder *transpose*

You have to implement the following variants of matrix transpose:

1. Vanilla - no optimizations, regular doubly-nested loop, with contiguous writes.

2. Flipped - Flip the loops for computing the transpose, so that the writes are now strided.

3. Unrolled - unroll the outer loop for the vanilla version. You can pick unroll factors of 4 and 8.

4. Blocked - Use 2D tiling to reduce cache misses

2.1.1 Experiments

The matrix size and block size (for tiling) should be variables. You have to do the following experiments :

Assume the matrices are square (symmetric) for your experiments. All experiments should be performed on the compute nodes of Taub. You have to turn-off all compiler optimizations for your experiments, use the -O0 flag.

1. Given a matrix, measure the time taken for computing the transpose for all variants. You have to repeat each experiment 5 times and report time as the average of 5 measurements. You can use the PAPI function, ($PAPI\_get\_real\_cyc$) to measure time as the number of elapsed clock cycles. You are free to use any other suitable timing routine. Plot graphs for execution time vs matrix size (number of rows/columns) for the vanilla, flipped and unrolled versions. We have provided you with a batchscript that does all the experiments in one submission for a range of matrix sizes.

2. For a fixed matrix size, determine the block size that gives the best performance. Pick the matrix size to be 4096X4096 for this experiment. Vary block sizes from [10,50] in steps of 10 to determine the size that gives minimum execution time. Compare the value with the L1 and L2 sizes of the compute nodes. Plot a graph of the tiled matrix transpose (with the best tile size you found), for various matrix sizes.

Write a report with the graphs and analysis for each experiment.

2.2 Gauss Seidel

The Gauss-Seidel algorithm is one method for solving the Discrete Poisson Equation. Pseudocode for the sequential 3D algorithm is given on the following page. We have provided starter code for three implementations of 2D Gauss-Seidel:

gauss_seidel_sequential() is a sequential implementation for your reference. You will use the sequential version as a baseline for comparing performance of your parallel implementations.

gauss_seidel_pipelined() is a parallel implementation which uses explicit synchronization through pipelining.

gauss_seidel_diagonal() is a synchronization-free parallel implementation using wavefront propagation techniques (refer to textbook and lecture slides).

All three vesions of the code work correctly when run on a single processor; however, running the provided code for *gauss_seidel_pipelined()* produces incorrect results. Your task is to 1) modify *gauss_seidel_pipelined()* to work correctly with multiple threads, and 2) run tests comparing and contrasting *gauss_seidel_sequential()*, *gauss_seidel_pipelined()*, and *gauss_seidel_diagonal()*, while varying the number of threads, size of the matrices, and (for

Figure 1: Sequential 3D Gauss-Seidel-Algorithm (from Hager and Wellein)

```
1  double precision, parameter :: osth=1/6.d0
2  do it=1,itmax   ! number of iterations (sweeps)
3    ! not parallelizable right away
4    do k=1,kmax
5      do j=1,jmax
6        do i=1,imax
7          phi(i,j,k) = ( phi(i-1,j,k) + phi(i+1,j,k)
8                       + phi(i,j-1,k) + phi(i,j+1,k)
9                       + phi(i,j,k-1) + phi(i,j,k+1) ) * osth
10       enddo
11     enddo
12   enddo
13 enddo
```

gauss_seidel_pipelined()) size of chunks. We have provided some reference code in *main()* to help you with timing and printing relevant information.

2.2.1 Parallelization

Modify *gauss_seidel_pipelined()* in gs.c to provide support for cache coherency in *lastBlockDone*, adding *omp flush* directives as appropriate. You should not need to edit the code in any of the other files (i.e., makefile, gs-helper.h, or gs-helper.c). **NOTE:** the provided code may sometimes appear to run correctly on multiple threads. To verify you have implemented the code correctly compare the results with the sequential code provided. You might find it helpful to comment / uncomment *#define DPRINT* at the top of gs.c to show / hide the results.

2.2.2 Experiments

After you have fixed *gauss_seidel_pipelined()*, edit the code in main() to time all three methods, varying the size of the arrays, the number of threads (for the diagonal and pipelined versions), and the size of the chunks (for the pipelined version). Please test using the following values:

1. Threads: 1, 2, 4, 8...(up to the number of cores available)
2. Matrix Sizes: 32, 128, 512, 2048
3. Chunk Sizes: 2, 4, 8, 16, 32

For each matrix size, plot the execution time against the number of threads i.e., you should have 4 plots (for matrix sizes 32, 128, 512, and 2048), each of which should contain:

- X axis: number of threads

- Y axis: time taken
- Separate labeled lines for sequential, pipelined, and diagonal methods

For execution time, repeat each experiment at least 5 times and report the average of the 5 attempts. For the pipelined method, it's okay just to plot a single chunk size (whichever gave you the best performance), but note the size on the plot. Write a report analyzing your results based on your plots. Things to include:

1. In which cases did the diagonal method finish faster? What about the pipelined method? Interpret your results.
2. For the pipelined method, which chunk size was most effective? Can you explain why?
3. Does performance increase as expected when using more threads, or does it level off / drop dramatically? Explain any trends you notice.

2.3 Submission

Submission is via svn. Create a folder in your svn account named "MP1". Add all the files necessary for compiling and running your code. Your report should be neatly typed, preferrably in pdf. It should contain your results, including graphs and analyses, for both the matrix-transpose and Gauss-Seidel experiments.