Machine Problem 1

CS 484 - Parallel Programming

Due: March 3, 2019 @ 23:59

(Typeset on : 2019/02/19 at 00:59:04)

Introduction

Learning Goals

You will learn the following:

- More about how to schedule OpenMP parallel for loops.
- How to synchronize OpenMP threads with data dependencies.

Assignment Tasks

The basic workflow of this assignment is as follows (there are more details in the relevant sections):

- Clone your turnin repo to VMFarm [1]
  Your repo should be visible in a web browser and clonable with git from
  `https://gitlab.engr.illinois.edu/sp19-cs484/turnin/YourNETID/mp2.git` .
- You may need to iterate (especially for part1):
  - Implement the algorithms / code for the various programming tasks.
  - Build and test on VMFarm (see README.md) (It should pass all tests.)
  - Check in and push your complete, correct code.
  - Benchmark on Campus Cluster (`scripts/batch_script.run`)
- Check in any benchmarking results that you wish to and final versions of code files.
- Write and check in your writeup.

1 OpenMP Parallel Loop Scheduling

In this part of the assignment, you will gain some experience choosing the right schedule for parallelized loops.

Imagine (it's easy if you try) that you have been given an input of several computational tasks to do in parallel. Among these tasks, there are some (most in fact) that are trivial, but one or a few may be absurdly difficult (time-consuming). In general, you may or may not know in advance which problems are the hard ones.

There is a schedule that is best for these sorts of problems (assuming scheduling overhead is small in comparison to problem size). We have constructed a problem that, with the correct choice of schedule, will get very good, nearly linear speedup. While other choices of schedule will see speedup, only the correct one should see nearly-perfect speedup.

It is unlikely to see this behavior in real life, and there will be tradeoffs between different schedules. It is important to consider average behavior over all inputs your program is likely to encounter. However, for this assignment, we wanted the effect to be dramatic.

1.1 task

The code you have been provided (part1/src/misbalanced.cpp) is already correct and passes all unit tests. Your job is to alter the OpenMP pragma to choose the best loop schedule. (You should *only* alter the pragma line.)

Compilation will create a binary that only benchmarks this part of the assignment at
YOUR_BUILD_DIR/bin/part1_bench . You can run the benchmark by
submitting the batch script scripts/batch_script.run on Campus Cluster, which will put results files into the writeup directory.

[1] You are free to use the Docker container uiuccs484parallelprog/cs484_student , but we will provide no support for Docker itself.

Alter/compile/benchmark the code with various choices of schedule and chuck size.

Achieve a parallel efficiency of at least 0.93 **on the benchmark named** `BM_OpenMP_primes_parallel` **when run with 2 threads.**

1.2 Deliverables

In your final report, graph the performance of the schedule you decided was best, along with two others that you tried. Commit the best version of your altered pragma to part1/src/misbalanced.cpp .

2 Gauss-Seidel Synchronization

The Gauss-Seidel algorithm is one method for solving the Discrete Poisson Equation. We have provided starter code for two implementations of 2D Gauss-Seidel:

- `gauss_seidel_serial()` is a serial implementation for your reference. You will use the sequential version as a baseline for comparing performance of your parallel implementations. (Found in part2/gs_serial/gsserial.cpp .)
- `gauss_seidel_parallel()` is a parallel implementation which incorrectly uses pipelined computation. (Found in part2/gs_parallel/gsparallel.cpp .)

Both versions of the code work correctly when run on a single processor; however, running the provided code for `gauss_seidel_parallel()` with multiple threads produces incorrect results. Your task is to modify `gauss_seidel_parallel()` to work correctly with multiple threads
(you can check this by running YOUR_BUILD_DIR/bin/part2_tests), and run tests comparing the two implementations, while varying the number of threads and tile-size for `gauss_seidel_parallel()`. We have provided some benchmarking and correctness testing tools
to run these experiments (see YOUR_BUILD_DIR/bin).

For instructions on compiling/building your code, see README.md.

2.1 Parallelization

Modify `gauss_seidel_parallel()` in part2/gs_parallel/gsparallel.cpp so that it produces the same results as the serial version by using appropriate OpenMP synchronization clauses. You should not need to edit the code in any of the other files. You may **NOT** use the `depends` clause. Please do not otherwise optimize or alter the code, simply handle appropriate synchronization.

2.2 Experiments/Deliverables/Discussion

After you have fixed `gauss_seidel_parallel()`, run the benchmarking tool YOUR_BUILD_DIR/bin/part2_bench (This is done automatically in the batch script.). The tool will benchmark your code using a 1000x1000 matrix and the following parameters:

1. Threads: 1, 2, 4, 8
2. Tile Sizes: 8, 16, 32

Plot the execution time against the number of threads. Your plot should contain:

- X axis: number of threads
- Y axis: time taken
- Separate labeled lines for sequential and parallel methods. Please plot separate lines for each tile size for the parallel method.

After implementing the function, answer the following questions.

1. In which cases did the sequential method finish faster? What about the parallel method? Interpret your results.
2. For the parallel method, which tile 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. (In particular, what happens when there are more threads than cores?)

3 Submission

Your writeup must be a **single PDF file** that contains the tasks outlined above.

Please save the file as "`./writeup/mp2_<NetID>.pdf`*" , commit it to git, and push it to the* **master** *branch of your turnin repo before the submission deadline.*

You must also commit at least the following code files. These files, and only these files, will be copied into a fresh repo, compiled, and tested at grading time.

- part1/src/misbalanced.cpp
- part2/gs_parallel/gsparallel.cpp

Nothing prevents you from altering or adding any other file you like to help your debugging or to do additional experiments. This includes the unit test and benchmark code. (Which will just be reverted anyway.)

It goes without saying, however, that any attempt to subvert our grading system through self-modifying code, linkage shenanigans, etc. in the above files will be caught and dealt with harshly. Fortunately, it is absolutely impossible to do any of these things unaware or by accident, so relax and enjoy the assignment.