Programming I

Due: 12:29 PM, Sept 20, submit the report and source codes to canvas

100 points total

Overview

You will practice using programming models (abstractions): shared address space and data parallelism, in this assignment. Particularly, you will use Pthreads and ispc, which support the two types of abstractions respectively. Through the assignment, you will develop an understanding of the parallel execution on modern multicore architectures.

References:

1. Pthreads
2. Intel SPMD Program Compiler (ISPC)

Make sure to go through the examples, understand them, and run them.

Experiment Platform

You will need to run the codes on a multicore system in Palmetto cluster. You can find user's guide at https://www.palmetto.clemson.edu/palmetto/userguide_howto_control_job_placement.html. Make sure that you place your job on a node with 8+ cores.

Part 1: BLAS saxpy (20 points)

`saxpy` computes the simple operation `result = scale*X+Y`, where `X`, `Y`, and `result` are vectors of `N` elements (in Program 5, `N` = 20 million) and `scale` is a scalar. `saxpy` is a trivially parallelizable computation and features predictable, regular data access and predictable execution cost.

Compile and run `saxpy`. The program will report the performance of ISPC (without tasks) and ISPC (with tasks) implementations of saxpy. What speedup from using ISPC with tasks do you observe? Explain the performance of this program. Do you think it can be substantially improved? (for example, could you rewrite the problem to achieve near linear speedup?) (yes or no? Please justify your answer.)

Part 2: Parallel Fractal Generation Using Pthreads (40 points)

You are provided with codes and Makefile in the `mandelbrot_threads/` directory. Make and run the code. This program produces the image file `mandelbrot-serial.ppm`, which is a visualization of a famous set of complex numbers called the Mandelbrot set. The .ppm file can be opened with command `display`. By using command option --view 2, the program generates

a different image. You can learn more about the definition of the Mandelbrot set at **http://en.wikipedia.org/wiki/Mandelbrot_set**.

Parallelize the computation of the images using pthreads. Starter code that spawns one additional thread is provided in the function `mandelbrotThread()` located in `mandelbrotThread.cpp`. In this function, the main application thread creates another additional pthread using `pthread_create`. It waits for this thread to complete using `pthread_join`. Currently the launched thread does not do any computation and returns immediately. You should add code to `workerThreadStart` function to accomplish this task. You will not need to make use of any other pthread API calls in this assignment.

1. Modify the starter code to parallelize the Mandelbrot generation using two processors. Specifically, compute the top half of the image in thread 0, and the bottom half of the image in thread 1. This type of problem decomposition is referred to as *spatial decomposition* since different spatial regions of the image are computed by different processors.
2. Extend your code to utilize 2, 4, 8, and 16 threads, partitioning the image generation work accordingly, if the processor has 8 cores supporting two hyper-threads. Use the same idea to extend to 20 threads if the processor has 10 cores supporting two hyper-threads. In your write-up, produce a graph of **speedup compared to the reference sequential implementation** as a function of the number of cores used FOR VIEW 1. Is speedup linear in the number of cores used? In your writeup hypothesize why this is (or is not) the case? (you may also wish to produce a graph for VIEW 2 to help you come up with an answer.)
3. To confirm (or disprove) your hypothesis, measure the amount of time each thread requires to complete its work by inserting timing code at the beginning and end of `workerThreadStart()`. How do your measurements explain the speedup graph you previously created?
4. Modify the assignment of work to threads to improve speedup to at **about 8.5x on a 8-core machine on both views** of the Mandelbrot set (if you're close to 8.5x that's fine, don't sweat it). You may not use any synchronization between threads. We are expecting you to come up with a single work decomposition policy that will work well for all thread counts---hard coding a solution specific to each configuration is not allowed! (Hint: There is a very simple static assignment that will achieve this goal, and no communication/synchronization among threads is necessary.). In your writeup, describe your approach and report the final 16-thread speedup obtained. Also comment on the difference in scaling behavior from 4 to 8 threads vs 8 to 16 threads.

Part 3: parallelize with ISPC (40 points)

Before proceeding, you are encouraged to familiarize yourself with ISPC language constructs by reading through the ISPC walkthrough available at **http://ispc.github.com/example.html**. The example program in the walkthrough is almost exactly the same as Program 3's implementation of `mandelbrot_ispc()` in `mandelbrot.ispc`. In the assignment code, we have changed the bounds of the foreach loop to yield a more straightforward implementation.

1. Compile and run the program mandelbrot ispc. **The ISPC compiler is currently configured to emit 4-wide SSE vector instructions. (although you're encouraged to experiment with changing that compile flag to emit AVX, please answer this question for the SSE configuration.)** What is the maximum speedup

you expect given what you know about these CPUs? Why might the number you observe be less than this ideal? (Hint: Consider the characteristics of the computation you are performing? Describe the parts of the image that present challenges for SIMD execution? Comparing the performance of rendering the different views of the Mandelbrot set may help confirm your hypothesis.)

ISPCs SPMD execution model and mechanisms like `foreach` facilitate the creation of programs that utilize SIMD processing. The language also provides an additional mechanism utilizing multiple cores in an ISPC computation. This mechanism is launching *ISPC tasks*.

See the `launch[2]` command in the function `mandelbrot_ispc_withtasks`. This command launches two tasks. Each task defines a computation that will be executed by a gang of ISPC program instances. As given by the function `mandelbrot_ispc_task`, each task computes a region of the final image. Similar to how the `foreach` construct defines loop iterations that can be carried out in any order (and in parallel by ISPC program instances, the tasks created by this launch operation can be processed in any order (and in parallel on different CPU cores).

1. Run `mandelbrot_ispc` with the parameter `--tasks`. What speedup do you observe on view 1? What is the speedup over the version of `mandelbrot_ispc` that does not partition that computation into tasks?
2. There is a simple way to improve the performance of `mandelbrot_ispc --tasks` by changing the number of tasks the code creates. By only changing code in the function `mandelbrot_ispc_withtasks()`, you should be able to achieve performance that exceeds the sequential version of the code by about 16-18 times! How did you determine how many tasks to create? Why does the number you chose work best?

Submission Instructions

Submission will be through canvas.

Please place the following files in your submission:

- Your writeup, in a file called `writeup.pdf`
- Your source codes in a single tar file. All code must be compilable and runnable!