EC527: High Performance Programming with Multicore and GPUs
Programming Assignment 5

Objectives
Learn about and practice using:
(0) Another simple workhorse application: Successive Over-Relaxation (SOR)
(1) A seemingly minor variation of SOR, which can cause a major difference in performance. This discovery (the use of a tuned parameter *OMEGA*) revolutionized scientific computing by showing that good numerical analysis can effect a vast change in performance.
(2) Multithreaded programming for performance; in particular, finding the cost of thread overhead and, by simple extension, the break-even point of using multithreading for performance
(3) Optimizations used for stencil codes (the family of applications of which SOR is a part), starting with a serial reference code
(4) Parallelizing the reference code using pthreads.

Prerequisites (to be covered in class or through examples in on-line documentation)
HW – A multiprocessor with shared address space. The cost of sharing data, but with no model (yet) of cache coherence.
SW – Basics of parallel processing, especially the implications of different *partitioning* strategies (i.e., strategies in *decomposing* the problem into tasks and *assigning* those tasks to processes).
Programming – Essential Pthreads: create, passing parameters, join, sync, barrier.

Assignment

Lab setup: Not required, but suggested (for better results and to practice for the project) you can use eng-grid nodes for more significant scalability studies. See using_eng-grid.txt for details.

Part 1 – Finding Optimal OMEGA
Reference code: **test_SOR_OMEGA.c**

> *Overall: You always need to be thoughtful about your experiments, but that is especially true here!! You will run your codes on a range of **array sizes** and **OMEGAs**. You will find two things: (i) outputs are noisy for small arrays and (ii) large arrays take a long time to converge, especially when OMEGA is far from optimal. We therefore suggest the following. For small arrays, run multiple iterations, say, 10, and take the average; this will give you smooth graphs but add little to the execution time. For large arrays a single run may be sufficient. Also please note: for OMEGA that is far off from optimal you may never converge at all! Use your results for small arrays to notice the pattern and constrain the range for large arrays (for which you might only need a couple of data points).*

The reference code contains a basic serial implementation of SOR. Note that it counts iterations to convergence rather than time. Also note that the program iterates on values of OMEGA rather than array size. You are welcome to add another loop to do multiple array sizes per run (as well as multiple OMEGAs), but you will probably find that you don't need to. You can adjust TOL (the tolerance for convergence) as long as there are sufficient iterations to show the trend.

Task: Find optimal OMEGA as a function of array size.

Method: To vary the array size, change ARRAY_SIZE and recompile. It's a 2-D array, and as given ARRAY_SIZE is 32 so it's doing a 32x32 array.

The main point of the program is to test different values of the "*OMEGA*" constant, so there is another set of three constants (START_OMEGA, OMEGA_INC, O_ITERS) that is used to test a range of OMEGAs. Change the base (OMEGA_START) and the number of increments (O_ITERS). As given these are 0.5, 0.01, and 150 respectively, so it checks 150 values from 0.50 through 1.99 inclusive.

There is also "PER_O_TRIALS": this gives the number of runs per OMEGA/array-size combination.

Each line of output tells what OMEGA value it used and how many iterations it took to converge. For example, this:

```
0.50, 1979, 2304, 1857, 993, 1291, 1596, 1980, 2273, 1959, 1967
```

means that with OMEGA=0.50 it did 10 tests, and the first test took 1979 iterations to converge, the second test took 2304 iterations to converge, ... etc.

After going through the OMEGA values it then prints the actual data to plot:

```
0.50 1819.9
0.51 1913.6
0.52 1697.9
…
```

This is the average number of iterations for each OMEGA value.

Observation (again): You will notice that for small array sizes there is substantial variance in iterations to convergence. Fortunately these runs (for small arrays) are fast.

Recommendation (again): For small array sizes, results will come back in a few seconds, even for a wide range of OMEGAs and a large number of runs per OMEGA. But as the arrays get larger, the time to convergence will get into the minutes and longer. The good news is that, as the array size gets larger, you will need to try fewer and fewer values of OMEGA, as well as fewer PER_O_TRIALS. With judicious selections of OMEGA and number of trials, you should be able to try a broad range of array sizes in reasonable time, and also get accurate results.

Deliverable: Graph showing of iterations as a function of OMEGA for several array sizes (best to show all in one graph: for example, a set of blue dots for array size 32x32, a set of green dots for array size 64x64, etc.). A few of your array sizes should be large enough to not fit in L2 cache.
Describe what you find:

- What are the overall shapes?
- How does OMEGA change with array size?
- What is the sensitivity of OMEGA selection on iterations to convergence?
- Given a (near) optimal OMEGA, qualitatively, what affects the number of iterations to convergence?

Part 2 – Serial SOR optimizations
Reference code: **test_SOR.c**

The reference code has three more implementations of SOR: one using the "red/black" method, one with the indices reversed, and one that is blocked. In order to get fair comparisons, the

init_array_rand() function uses srandom() to ensure that the starting data is the same "random" pattern each time.

Task: Find how these basic optimizations affect performance.

Method: This code has fixed OMEGA. Change the value of OMEGA to a good one (as determined by you in Part 1). Like previous assignments, we want to do a range of array sizes where the smallest fits in L2 cache (less than 256KB) but the biggest is too big for L3 cache (typically 8MB). You need to pick an OMEGA that is fairly good across the range of array sizes. Also, for blocked SOR, try a few different block sizes.

Note that for any particular array size there are two extra rows for the "ghost zone", but these do not count for meeting the requirement to have the size be a multiple of your block size(s). To accomplish this, leave GHOST equal to 2, and choose A, B, and C to all be a multiple of the block size.

It will take a few minutes to run each of the larger sizes, so you may wish to do the large sizes in a separate run, using an OMEGA that is suitable for larger array size (as you found in part 1), and with larger coefficients A B and C, and smaller NUM_TESTS.

Deliverable: Find the time per innermost loop iteration for the four methods (the number of innermost loop iterations is the number of "iters" it took to converge, times the array height, times the array width; and the "iters" is probably different for each of the four methods).

Are you surprised by the results? Does blocking make as big a difference as it does for MMM? Can you explain why or why not?

Part 3 – Cost of multithreading

Reference code: **test_pt.c**

The reference code computes a computationally intensive function (several transcendentals) on elements of the input array. There are two versions, serial (for baseline) and pthreaded. You can also try a function that is NOT computationally intensive – there is a sample commented out in both functions.

Task: Find the overhead of Pthreads: How long does it take to do the minimum necessary operations for threading (**creating threads, passing parameters,** and **join**) ? The way to find this out is to make the array size so small that a 4-threads version takes almost twice as long as the 2-threads version. You could fit your time measurements to Amdahl's law, with the overhead being the "serial" part of the program; or just compute how many extra microseconds it is taking for each additional thread.

Find the break-even point with respect to array size and work-per-thread between the serial and multithreaded versions. The more work in the computation (array size, work per element), the more likely it is that a parallel implementation will give you better performance than single threaded implementation. The break-even point is the size at which more threads is just about the same speed as the serial version.

You have some flexibility here, but it would be good to see two workloads (e.g., a complex function and a simple function), and two or more thread counts (e.g., 2 and 4 but no more than the number of physical cores). For each scan through a range of array sizes (starting at least as small as 10) to find how large the array needs to be for multiple threads to be faster than 1 thread).

Optional is to try this out on a machine with more than 8 cores. (On ENG Grid using "qlogin -q interactive.q" you can get a machine with 10 cores or more.) You can also try using twice as many threads as cores, to see whether Intel's Hyperthreading, or AMD's CMT or SMT, helps here.

Deliverable: Results and brief explanation.

Part 4 – Multithreaded SOR

Task: Create and evaluate, with respect to the serial baseline, at least two multithreaded versions of SOR. Test two array sizes: one where the array fits in L3 cache and one where it does not. For each case, choose a good OMEGA. For maximum cache utilization you want to avoid powers of 2 in your array row lengths. You should plot your results with respect to the number of threads.

Definitely try: decomposition by strips (groups of adjacent rows) and decomposition in some other way, e.g., nonadjacent rows or vertical strips.
Optional is to also try decomposition by squares/rectangles.
Optional is to also try different starting points from Part 2 like SOR or blocking.

Deliverable: Your code, results, and interpretation.