Programming Assignment 6
Due Sunday, March 22nd

Objectives
Learn about and practice using:
(1) a seemingly minor variation of SOR which can cause a major difference in performance. This discovery (OMEGA) revolutionized scientific computing by showing that good algorithms can have a vast change in performance.
(2) basic multithreaded programming for performance, in particular finding the cost of basic thread overhead and, by simple extension, the break-even point of using multithreading for performance
(3a) standard simple optimizations in developing SOR serial reference code
(3b) parallelizing the reference code using Pthreads.

Prerequisites (to be covered in class or through examples in on-line documentation)
HW – We assume a basic multiprocessor with shared address space. Also that there is a cost to shared data, but with no model (yet) of cache coherence.
SW – Basics of parallel processing, especially the implications of different partitioning strategies.
Programming – Basics of PThreads: create, passing parameters, join, sync, barrier.

Assignment

Lab setup: The machines in PHO207 are dual core while those in PHO307 are quad core. These are fine for testing out the threads basics, for debugging your code. The 307 machines are also OK for basic scalability studies. 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 OMEGA
Reference code: test_SOR_OMEGA.c

> *Overall: Especially here, you need to be thoughtful with your experiments!! You will run your codes on a range of array sizes and OMEGAs. You will find two things: that outputs are noisy for small arrays and that large arrays take a long time to converge, especially for OMEGA far from optimal. This means being careful with your experiments. For small arrays, run them for 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, for OMEGA that is far off (for large arrays) 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. Notice 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 want to.
Task: Find optimal OMEGA as a function of array size.
Method: The program lets you vary the size, just change DELTA or BASE while leaving ITERS at 1 (I left these in in case you wanted to use them). To test a range of OMEGAs for a given array size, change the base (OMEGA) and the number of increments (O_ITERS). Note "PER_O_ITERS": this gives the number of runs per OMEGA/array-size combination.
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 you get larger, this 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 OMEGAs, as well as fewer points per OMEGA. With judicious selections, you should be able to try a broad range of array sizes in reasonable time.
Deliverable: Graph of iterations versus OMEGA for a range of array sizes. A few should be large enough to not fit in 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 two more implementations of SOR, one with the indices reversed, and one that is blocked.
Task: Find how these basic optimizations affect performance.
Method: This code has fixed OMEGA, the idea in this part is to vary array size as we did in previous assignments. Also, for blocked SOR, try a few different block sizes.
Deliverable: Find the time per inner loop iteration for the three methods. Graph these results versus array size. Are you surprised by the results? Can you explain them?

Part 3 – Cost of multithreading
Reference code: test_pt.c
The reference code computes a computational 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 operations necessary to all likely threaded codes – create, pass parameters, join.
Find the break-even point with respect to array size and work-per-thread between the serial and multithreaded versions. That is, 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. You have some flexibility here, but it would be good to see two workloads (e.g., a complex function and a simple function), two thread counts (e.g., 1 and 4 but no more than the number of physical cores), and a range of array sizes.
Optional is to try this out on an eng-grid machine with more than 4 cores.
Deliverable: Results and brief explanation. Bonus for a formula based on Amdahl's Law.

Part 4 – Multithreaded SOR
Task: Create and evaluate, with respect to the serial baseline, at least two multithreaded versions of SOR. You can restrict your array sizes to two: one where the array will fit in cache and one where it will not. For each case, choose a good OMEGA. You probably want to avoid powers of 2 in your array sizes.
Definitely try: decomposition by strips and decomposition by strips with blocking. Optional is to also try decomposition by squares.
Deliverable: Your code, results, and interpretation.