Mikhail Andreev

EC527 Assignment 7

Using bme-compsim-55 (3.6 GHz)

**Part 1: “Hello World!”**

The code that accomplishes this is:

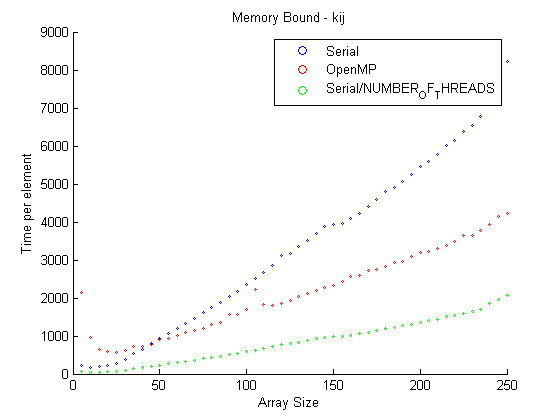
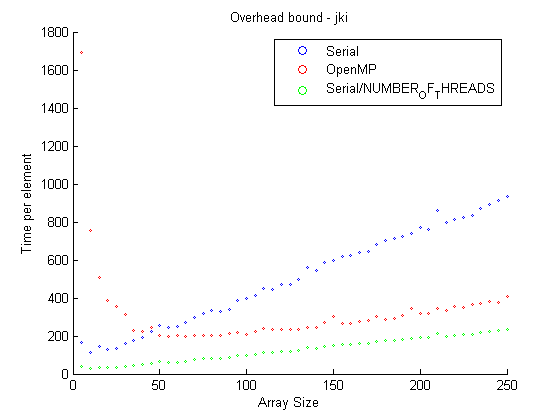
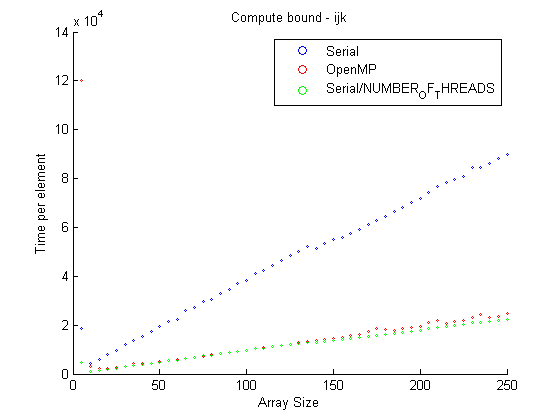


The output given by this code is:



**Part 2: Parallel For**

By examining the output for the 3 code versions, we can graph the output of the serial values versus the parallel versions, which in turn will tell us the overhead and the break-even point. In the following graphs, the blue graph is the serial version, the red graph is the OpenMP version, and the green graph is the time it would take to do computations without thread overhead.



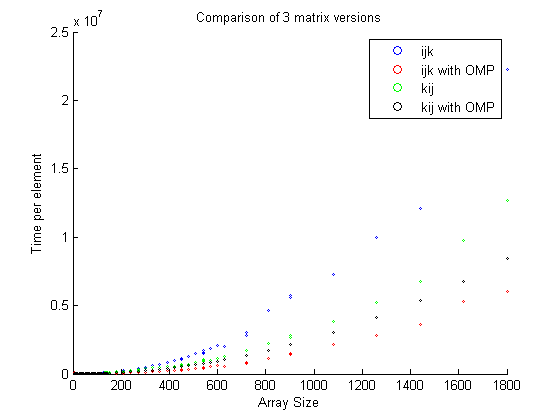
These graphs can tell us the overhead values by averaging the difference between the parallel version and the no-overhead parallel version.



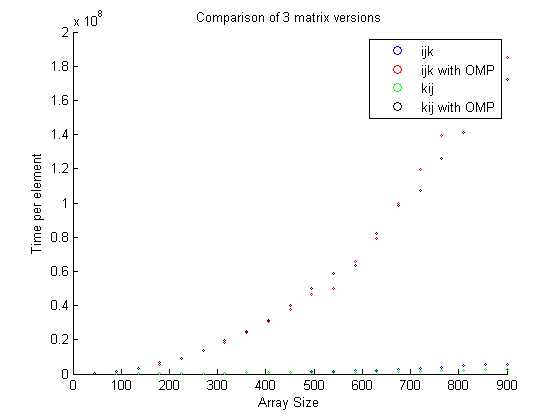
From these graphs we can also see the break-even points. The break-even point for the compute bound is at 10 array elements. The break-even point for memory bound is at 50 array elements. The break-even point is at 45 array elements.

**Part 3: MMM, 3 loop version**

1. Running the code with different array sizes generates the following graph:

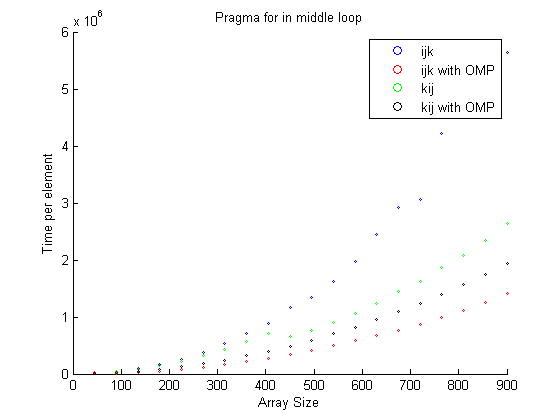


1. After removing the indices from the code the resulting graph is shown here:

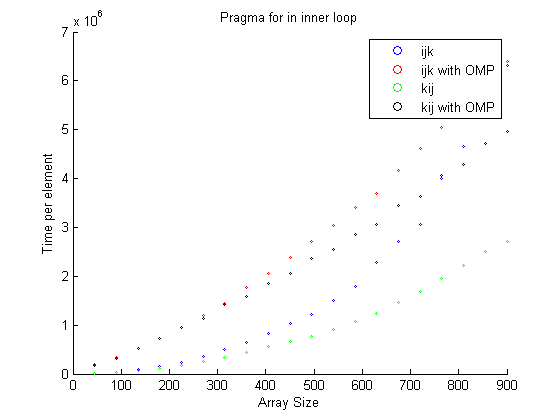


This clearly shows that removing the indices greatly inhibits performance of the OMP code, causing it to slow down considerably compared to the baseline. The reason for this is that since the variables are no longer private, they have to be shared between the threads, causing a lot of waiting.

1. When moving #pragma for into the middle loop we get:



When moving the #pragma for into the inner loop, we get the graph:



As can be seen the fastest implementation is when #pragma for is in the middle loop. This occurs because if the parallelization occurs on the outer loop, the amount of work done by each thread is still very high, decreasing the effectivity of separating the work. If the pragma for is put on the inner loop, the overhead from threading drives up the speed, making it inefficient. However, the middle loop provides the balance needed to achieve fastest return.

**Part 4: OpenMP on real programs**

1.

The following SOR code implements OpenMP:



And produces an output of:



Where the last value is produced by the OpenMP code.