Mikhail Andreev

EC527 Assignment 7

Using bme-compsim-55 (3.6 GHz)

Part 1: “Hello World!”

The code that accomplishes this is:

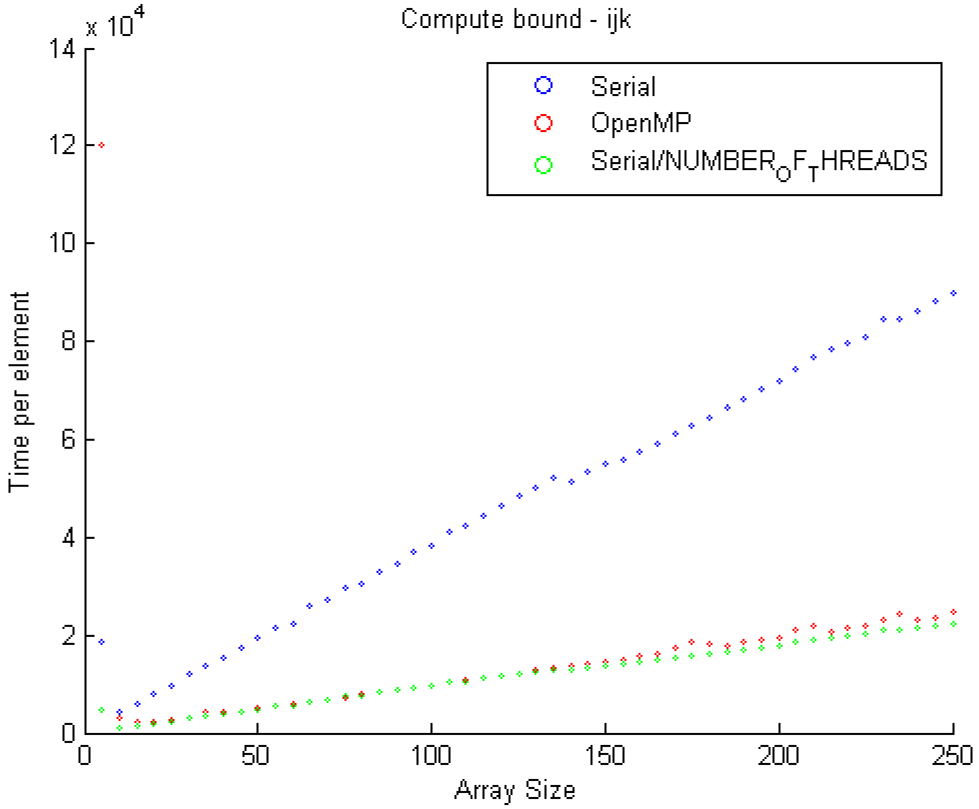
```
  char out[] = "Hello World!";

  printf("\n Hello World -- Test OMP \n");

  omp_set_num_threads(4);

#pragma omp parallel for
  for(i = 0; i < 12; i++)
    printf("%c",out[i]);
```

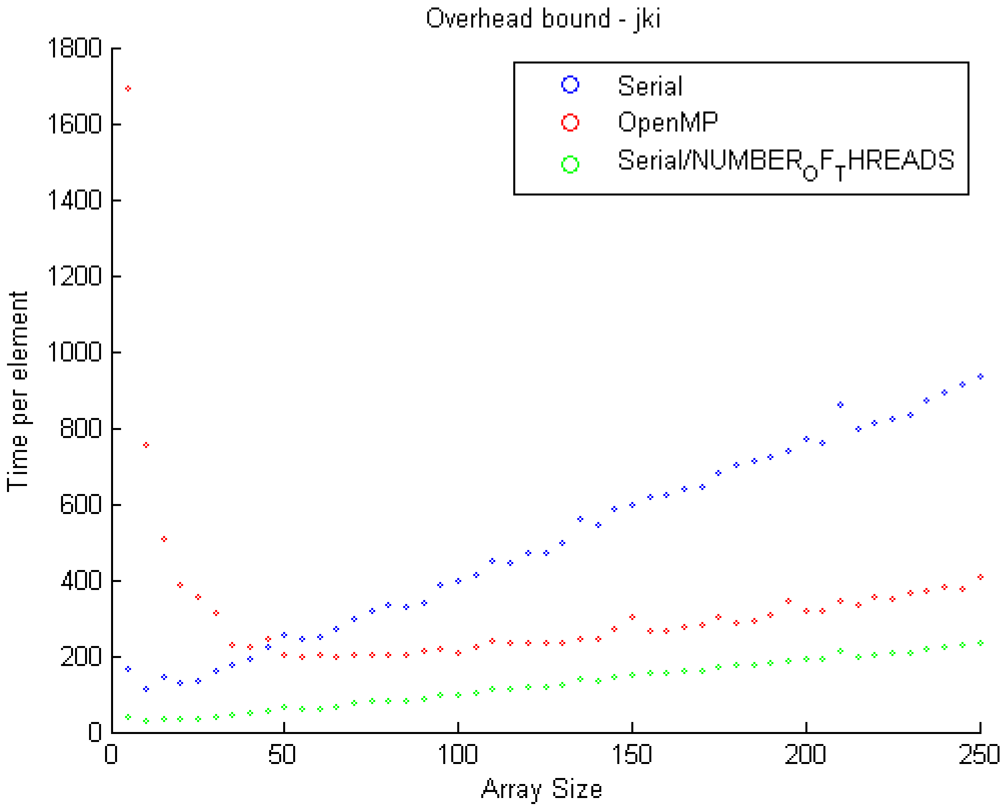
The output given by this code is:

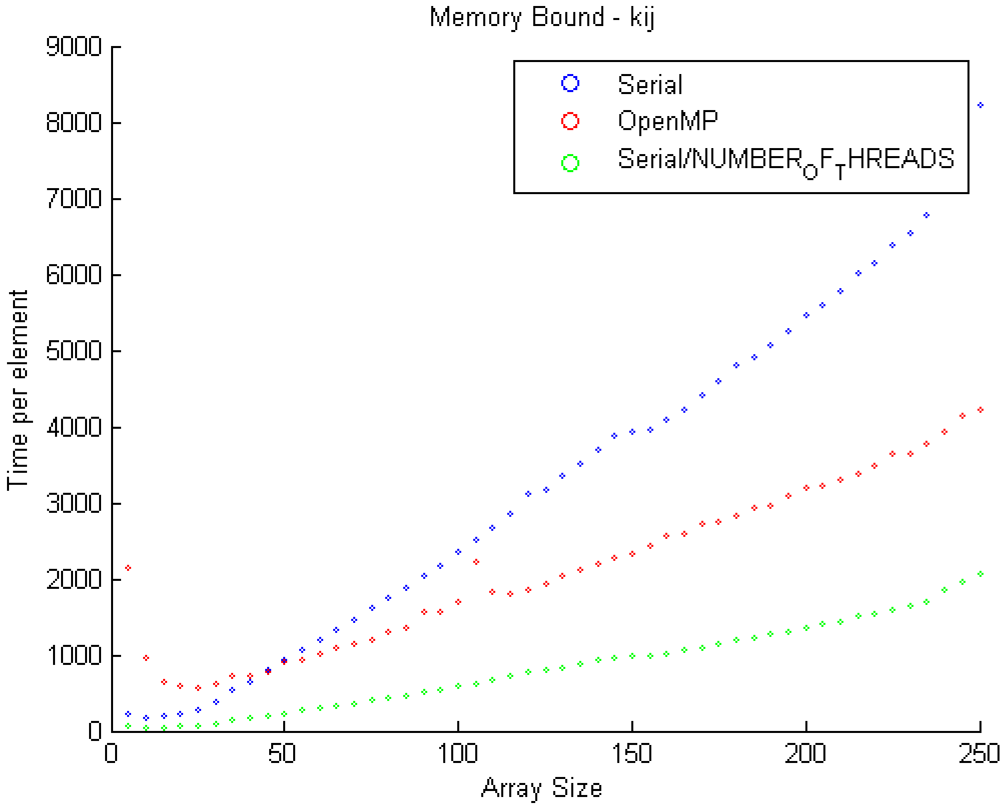
```
Hello World -- Test OMP
WorHleld!lo
```

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.


Compute bound - ijk
x 10^4
14
12
10
8
6
4
2
0
Time per element
0
50
100
150
200
250
Array Size
Serial
OpenMP
Serial/NUMBER_OF_THREADS


Overhead bound - jki
Serial
OpenMP
Serial/NUMBER$_O$F$_T$HREADS
Time per element
1800
1600
1400
1200
1000
800
600
400
200
0
0
50
100
150
200
250
Array Size


Memory Bound - kij
Serial
OpenMP
Serial/NUMBER$_O$F$_T$HREADS
Time per element
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
0
50
100
150
200
250
Array Size

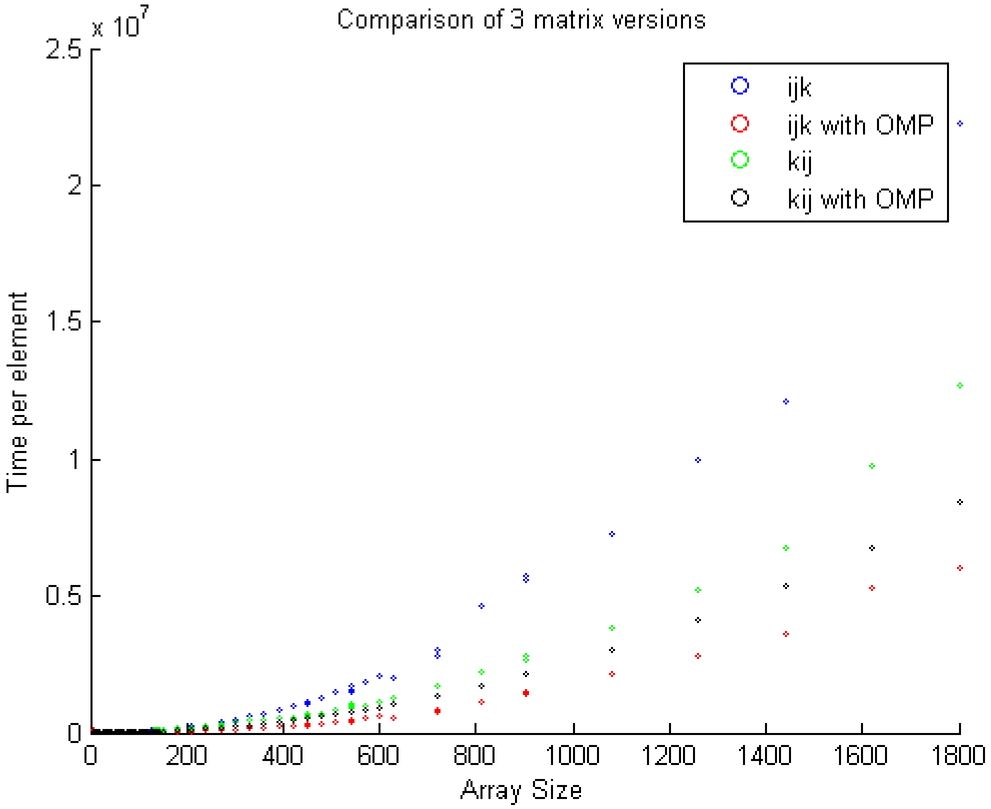
These graphs can tell us the overhead values by averaging the difference between the parallel version and the no-overhead parallel version.

```
Compute Bound Overhead = 3.213478e+03
Memory Bound Overhead = 1.300667e+03
Overhead Bound Overhead = 1.928206e+02
```

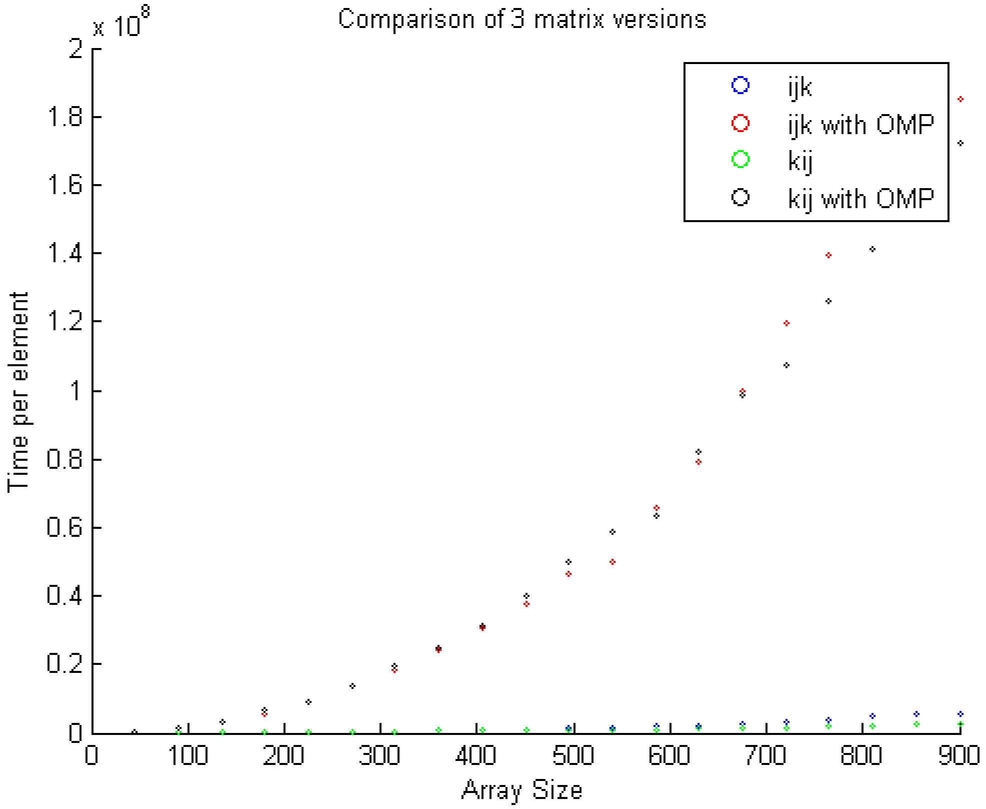
From these graphs we can also see the break-even points. The break-even point for the compute bound is at 10 array elements. The break-even point for memory bound is at 50 array elements. The break-even point is at 45 array elements.

Part 3: MMM, 3 loop version

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

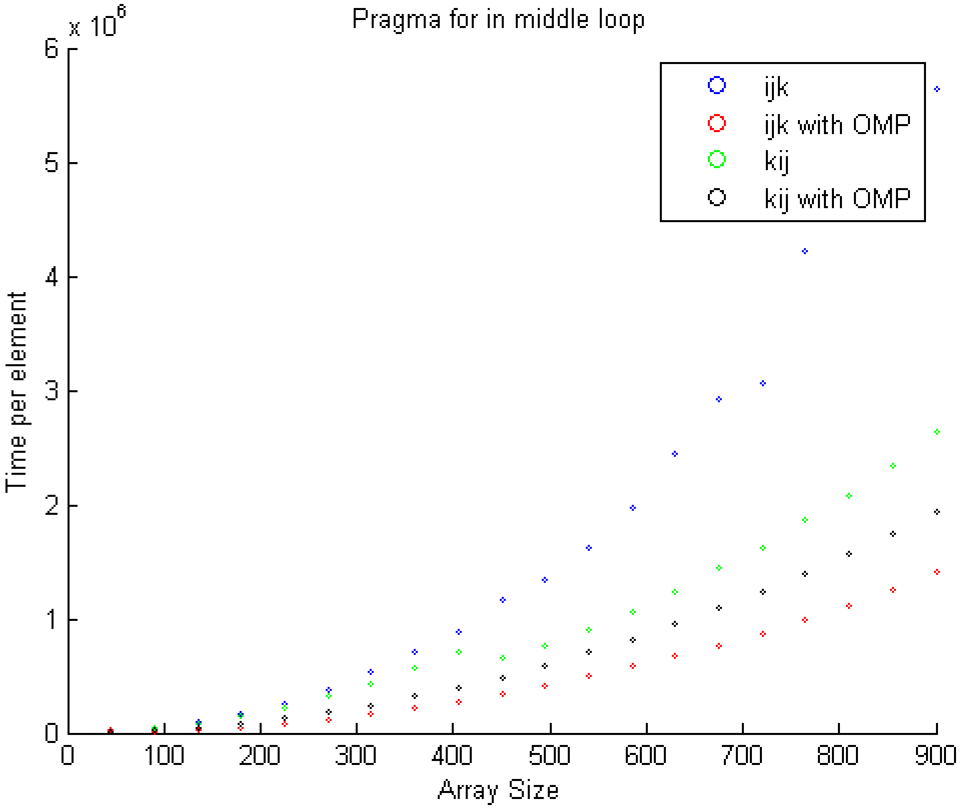


2. After removing the indices from the code the resulting graph is shown here:

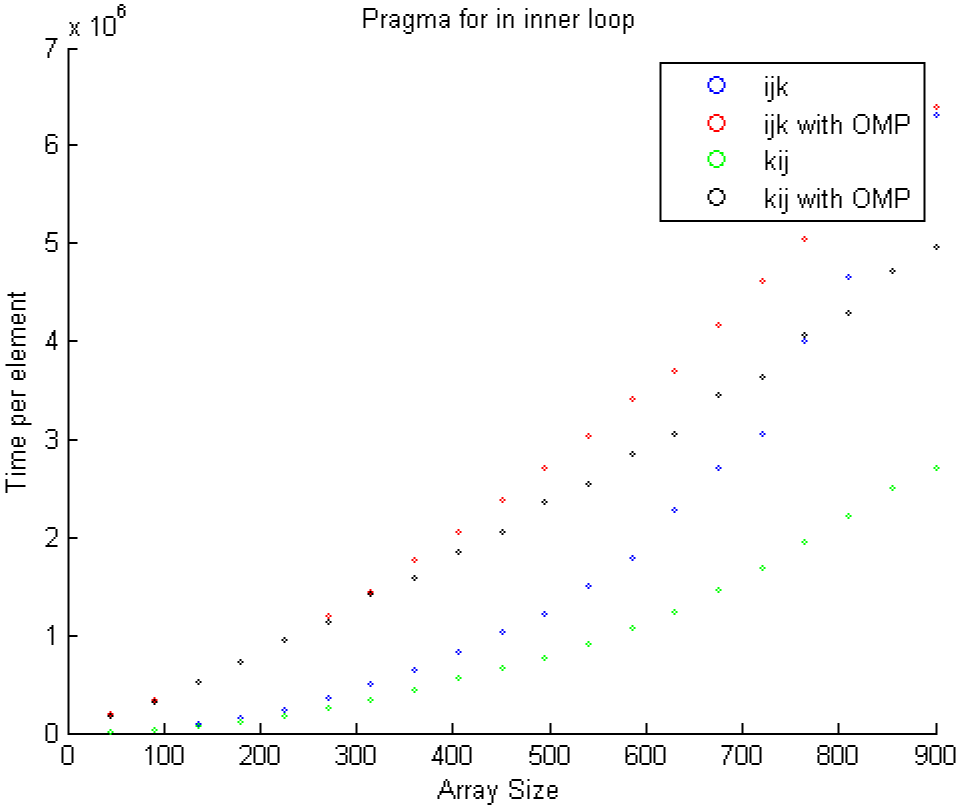


This clearly shows that removing the indices greatly inhibits performance of the OMP code, causing it to slow down considerably compared to the baseline. The reason for this is that since the variables are no longer private, they have to be shared between the threads, causing a lot of waiting.

3. 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:

```
/* SOR OMP*/
void SOR_OMP(vec_ptr v, int *iterations)
{
  long int i, j;
  long int length = get_vec_length(v);
  data_t *data = get_vec_start(v);
  double change, mean_change = 100;   // start w/ something big
  int iters = 0;

  omp_set_num_threads(4);
 #pragma omp parallel shared(data, length, mean_change, iters, change)
private(i, j)
  {
    while ((mean_change/(double)(length*length)) > (double)TOL) {
      iters++;
      mean_change = 0;

      #pragma omp for
      for (i = 1; i < length-1; i++){
        for (j = 1; j < length-1; j++) {
          change = data[i*length+j] - .25 * (data[(i-1)*length+j] +
data[(i+1)*length+j] + data[i*length+j+1] + data[i*length+j-1]);
          data[i*length+j] -= change * OMEGA;
          if (change < 0){
            change = -change;
          }
          mean_change += change;
        }
      }
      if (abs(data[(length-2)*(length-2)]) > 10.0*(MAXVAL - MINVAL)) {
        printf("\n PROBABLY DIVERGENCE iter = %ld", iters);
        break;
      }
    }
  }

  *iterations = iters;
  printf("\n iters = %d", iters);
```

And produces an output of:

```
Hello World -- SOR serial variations

iter = 0 length = 502 OMEGA = 1.60
 iters = 1859
iter = 0 length = 502
 iters = 2110
iter = 0 length = 502
 iters = 2
502, 8,
16639084969, 1859
23197438368, 2110
10023627, 2
```

Where the last value is produced by the OpenMP code.