EC527: High Performance Programming with Multicore and GPUs

Programming Assignment 6

Part1 OpenMP basics

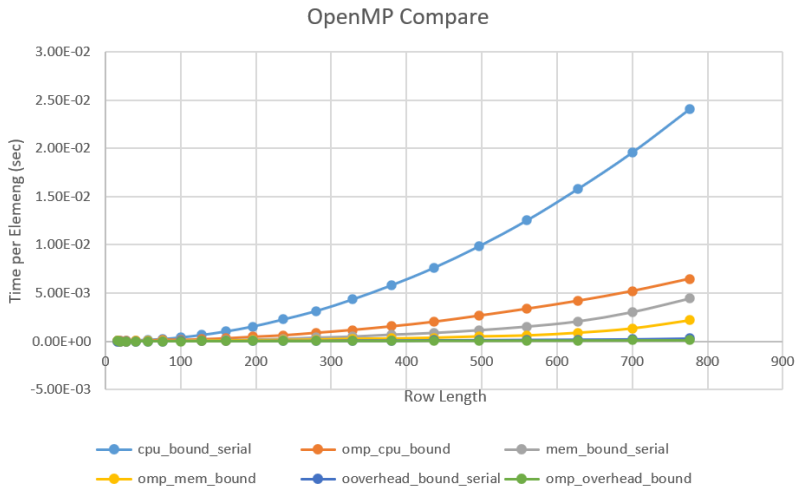
1a:

The codes are modified in test_omp.c. The result output is shown as below:

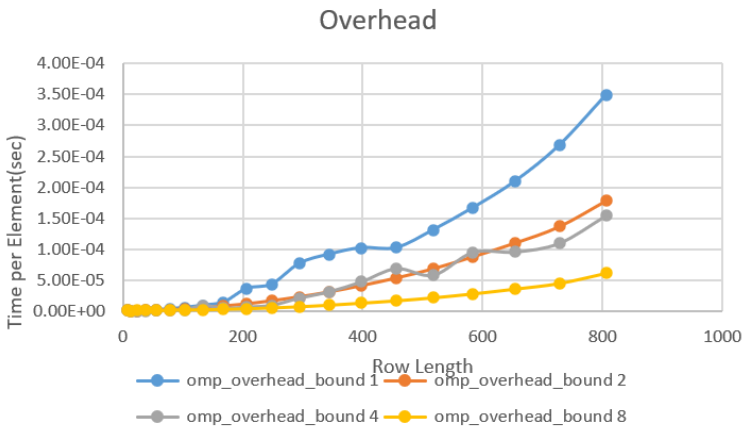
```
cloris@signals36$ ./test_omp
OpenMP race condition print test
omp's default number of threads is 8
Using 8 threads for OpenMP
Printing 'Hello world!' using 'omp parallel' and 'omp sections':

    eWolorld!Hl

Printing 'Hello world!' using 'omp parallel for':

    !WHolredoll
cloris@signals36$
```

1b:

As shown in the graph below, we can see that when using the OpenMP, it is much more efficient than the ordinary baseline way, no matter what the bound method is.

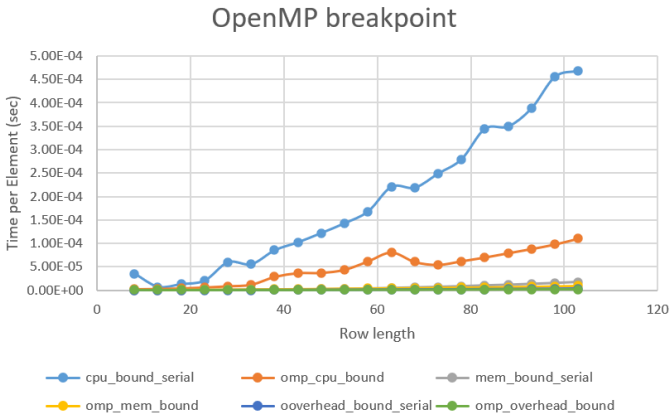


In order to find the overhead, I tried several thread numbers such as 1, 2, 4 and 8, and is shows that:



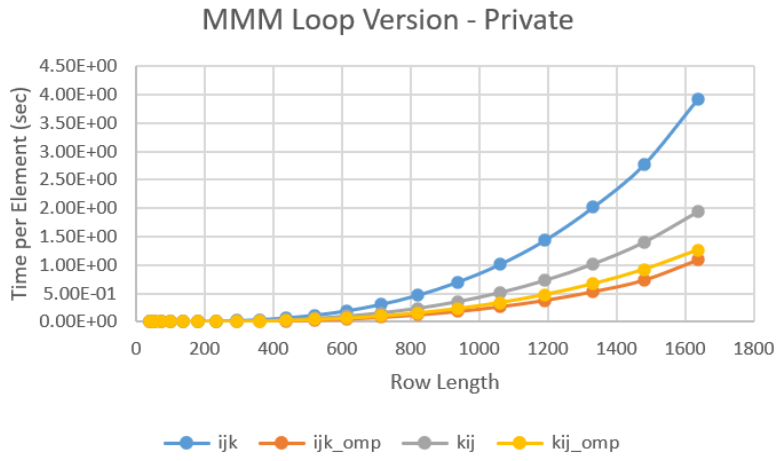
In order to find the OpenMP overhead, I tried to calculate the time by testing 2, 4 and 8 threads, and using the formula
Overhead = time_2_threads-time_1_thread or
Overhead = (time_4_threads-time_1_thread)/3 or
Overhead = (time_8_threads-time_1_thread)/7
For making it easier to see, it prefers to use some very small array size, and I chose the array size to be 24. So, the overhead time is 2.98E-07sec, 1.15E-07sec and 1.07E-07sec.

To find the break point, I tried to use the very small row length to see the result, and as shown in the graph below.



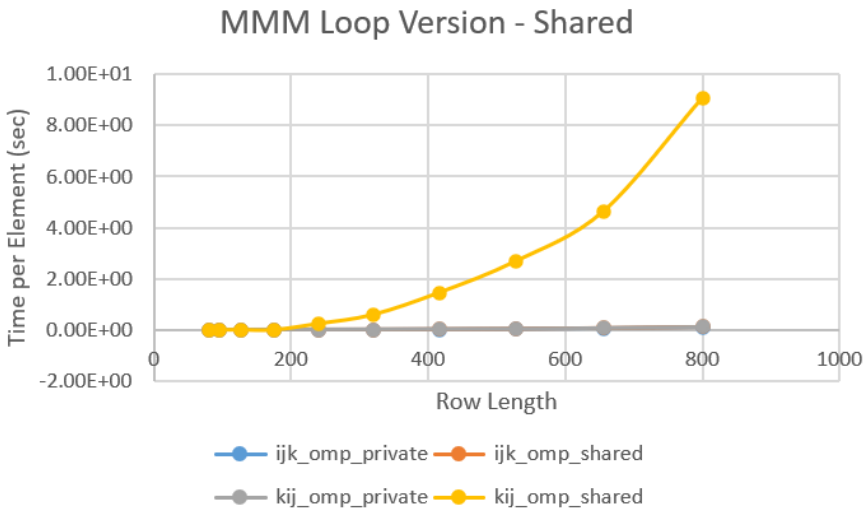
The break-even point for the CPU bound is at 18 array elements. The break-even point for memory bound is at 60 array elements. The break-even point for the overhead bound is at 65 array elements.

1c:
As it is demanded that the array size should range from less than L2 cache to larger to not fit in the L3 cache, as the L2 cache has 256K and L3 cache has 12288K and the data type used in the codes are float, we can conclude that the smallest array size is:
$\sqrt[3]{(256 * 1024/4)} \approx 40$ and $\sqrt[3]{(12288 * 1024/4)} \approx 146$.
When using the default private variables, we can see the result as graph shown below:



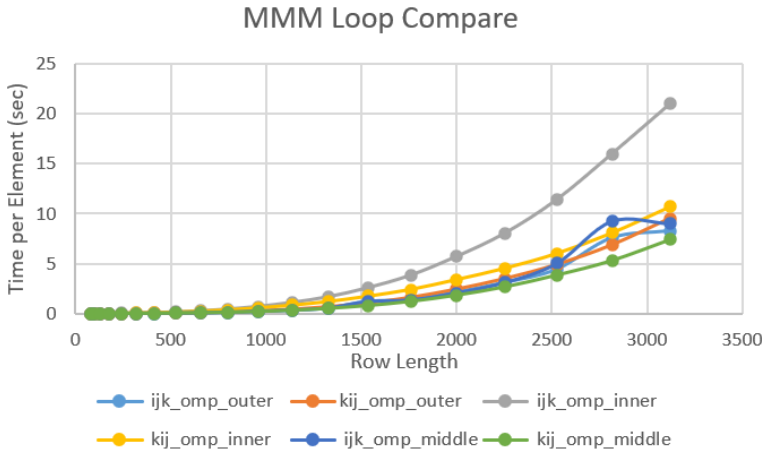
To make the result more clearly to see, I make a comparison between the private and

the shared one:



It can be seen that the versions where the private variables were moved to the shared variables the number of cycles are greater, especially when turning to the kij way. This is because there is greater work being done so synchronization is maintained between the different threads and there are no values getting corrupted.

Also, for the circumstance that when the loop is in the middle and inner, we can see from the result that when the loop is in the outside, it is the fastest. The reason behind that is the more outer pragma is written, the more synchronizations are required.



Part2

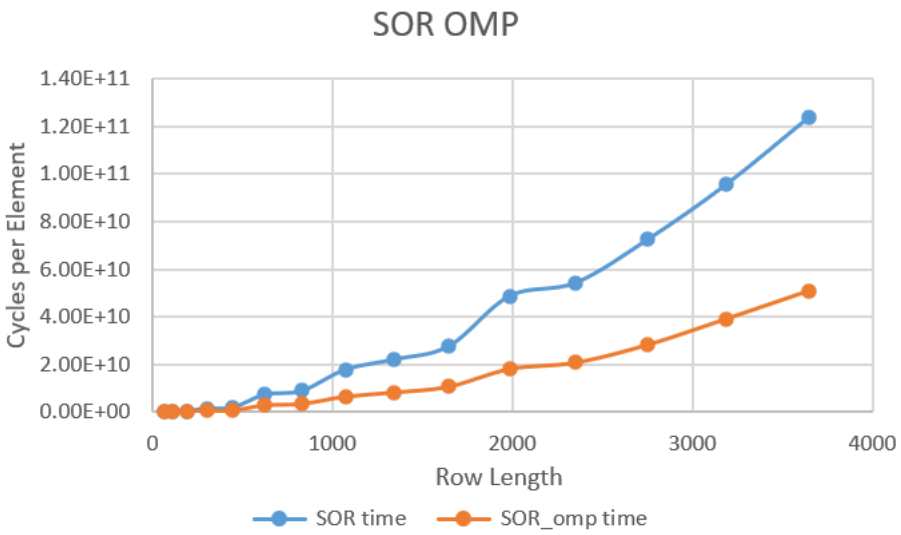
2a:

For using the OpenMP to parallelize test_SOR.c, I simply modified the codes by modifying the original SOR method. In order to achieve the goal of OpenMP, I set the number of threads to 4 and added the

```
void SOR_omp(arr_ptr v, int *iterations)
{
  long int i, j;
```

```
  long int rowlen = get_arr_rowlen(v);
  data_t *data = get_array_start(v);
  double change, total_change = 1.0e10;   /* start w/ something big */
  int iters = 0;

  omp_set_num_threads(NUM_THREADS);

  while ((total_change/(double)(rowlen*rowlen)) > (double)TOL) {
    iters++;
    total_change = 0;
#pragma omp parallel reduction(+:total_change)
    {
#pragma omp for private(i,j, change)
      for (i = 1; i < rowlen-1; i++) {
        for (j = 1; j < rowlen-1; j++) {
          change = data[i*rowlen+j] - .25 * (data[(i-1)*rowlen+j] +
                                             data[(i+1)*rowlen+j] +
                                             data[i*rowlen+j+1] +
                                             data[i*rowlen+j-1]);
          data[i*rowlen+j] -= change * OMEGA;
          if (change < 0){
            change = -change;
          }
          total_change += change;
        }
      }
    }
    if (abs(data[(rowlen-2)*(rowlen-2)]) > 10.0*(MAXVAL - MINVAL)) {
      printf("SOR: SUSPECT DIVERGENCE iter = %ld\n", iters);
      break;
    }
  }
  *iterations = iters;
  printf("    SOR() done after %d iters\n", iters);
}

}
```

The reason I set the i, j and change to be private is that the loop iterations, and also to accelerate, I used the reduction to do the overall add calculation. The result is shown below. We can see that when using the OMP for the SOR, it does accelerate.



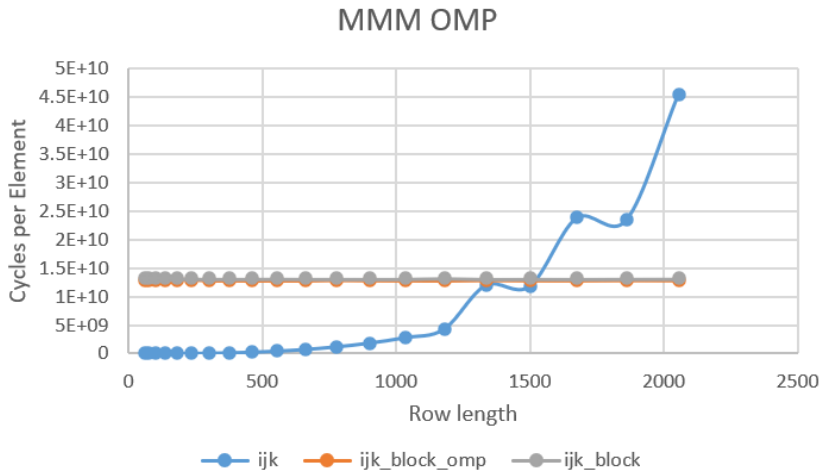
2b:

In the Lab1, I get my best MMM code in test_mmm_block.c by using function mmm_ijk_blocked(). So, I modified the codes shown below:

```
void mmm_ijk_blocked_omp(matrix_ptr a, matrix_ptr b, matrix_ptr c, long
int block_size, long int length)
{
    long int i, j, k, kk, jj;

    data_t *a0 = get_matrix_start(a);
    data_t *b0 = get_matrix_start(b);
    data_t *c0 = get_matrix_start(c);
    data_t sum;
    long int len = block_size * (length / block_size);

#pragma omp parallel shared(sum, length, len, a0, b0, c0, block_size)
private(i, j)
#pragma omp for

    for (i = 0; i < length; i++)
        for (j = 0; j < length; j++)
            c0[i * length + j] = 0.0;

    for (kk = 0; kk < len; kk += block_size)
    {
        for (jj = 0; jj < len; jj += block_size)
        {
            for (i = 0; i < length; i++)
            {
```

```
                for (j = jj; j < jj + block_size; j++)
                {
                    sum = IDENT;
                    for (k = kk; k < kk + block_size; k++)
                    {
                        sum += a0[i * length + k] * b0[k * length + j];
                    }
                    c0[i * length + j] += sum;
                }
            }
        }
    }
}
```

And the result is shown below:



We observe that when using OpenMP, we introduce a degree of parallelism in the code since the processes gets divided into threads that are operating in parallel.