Project 1 – Multicore Optimization

1. Matrix multiplication runtime (Build #335 runtime)

```
******Sequential*****

Test Case 1         0.00292969 milliseconds

Test Case 2         0.00512695 milliseconds

Test Case 3         0.128174 milliseconds

Test Case 4         0.131104 milliseconds

Test Case 5         3485.41 milliseconds

Test Case 6         4916.36 milliseconds

******OMP*****

Test Case 1         0.119141 milliseconds

Test Case 2         0.00585938 milliseconds

Test Case 3         0.013916 milliseconds

Test Case 4         0.0161133 milliseconds

Test Case 5         187.381 milliseconds

Test Case 6         201.597 milliseconds
```

2. K-means clustering configuration (Build #343 config)

numClusters = 32, threshold = 0.0010

Input	numObjs	numCoords
kmeans01.dat	351	34
kmeans02.dat	7089	4
kmeans03.dat	191681	22
kmeans04.dat	488565	8

3. Observed performance

Matrix multiplication speedup (combined) = ~21.6x (matrix_mul_02.dat)
k-means clustering speedup [Atomic] (combined) = 1.61x (kmeans01.dat), 5.72x (kmeans02.dat), 2.85x (kmeans03.dat), 8.12x(kmeans04.dat)

[All the above values are based on execution times on local machine, not Jenkins]

Details of performance optimization

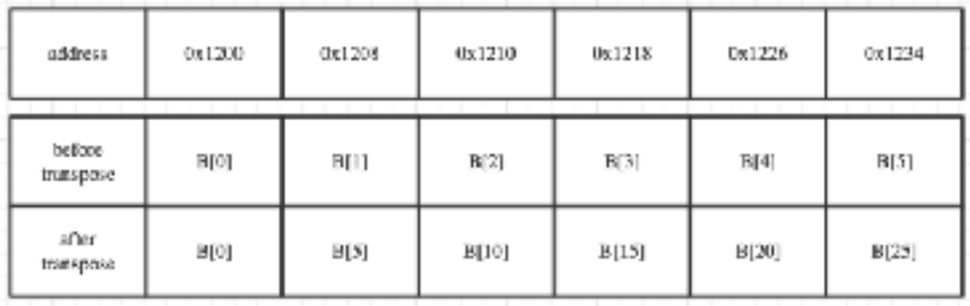
1. Matrix multiplication

The matrix multiplication have two part to optimization, the algorithm part and the operating time part(cache, parallel and hardware).

a. Goal: Optimization of cache miss rate

In this method, first, we aim to reduce the cache miss rate.
As we all know, data transferred between cache and memory with block-sized transfer units. CPU looks first for data in cache, then in main memory. When matrix A multiply matrix B, CPU reads data from B in a discrete address. By doing transpose, we can make the address of matrix data in B in a continuous order. When the CPU read data from cache, cache can return continuous address.
Secondly, we use temporary variables to reuse data in cache and reduce the cost of global variables.

Optimization process: As we can see from the project code of mutrix_multiply, product factor matrix A and matrix B's data are stored in two n*1 or 1*n arrays. At the same time, multiply results are stored in the same dimension array. When we do the matrix multiple, we choose data in B with skipping manner, the skipping coefficient is matrix dimension. By doing that, the cache stored will cost a lot of time. So, we transpose the matrix before the matrix multiply, to make the data in continuous order. The logic is shown below:



address	0x1200	0x1208	0x1210	0x1218	0x1226	0x1234
before transpose	B[0]	B[1]	B[2]	B[3]	B[4]	B[5]
after transpose	B[0]	B[5]	B[10]	B[15]	B[20]	B[25]

figure 1.2. Example of matrix transpose

The matrix transpose code is shown below:

```
for (unsigned int i=0; i< sq_dimension; i++) {
        for(unsigned int j=0; j < sq_dimension; j++) {
                m[j*sq_dimension + i] =
sq_matrix_2[i*sq_dimension + j];
        }
}
```

Meanwhile, we use temporary variables to reduce the cost of cache read and write.

The code is shown below

```
    for (unsigned int z = 0; z < sq_dimension; z++)
    {
    result_sum += sq_matrix_1[x*sq_dimension + z] * m[y*sq_dimension + z];
  }
      sq_matrix_result[x*sq_dimension + y] = result_sum;
```

Optimization results: We expect the test can speed up to 2X with original code and actually the test 6 speedup to 3X but the test case 5 speedup not that apparently. Overall, the whole speedup time is less than 5X. Maybe the data set not that large so the cache miss rate changing is affect less than what we expect.

b. Goal: Optimization of multicore

Using OpenMP to accelerate the execution speed by operating parallel execution with multi-thread.

Optimization process : The first step is to add *#pragma omp parallel num_threads(8) \ shared (sq_matrix_1, sq_matrix_2, m, sq_matrix_result)* after the declaration of the variables. This will start parallel computation for all matrix operations. Second, *#pragma omp for collapse(2) \ firstprivate(sq_dimension)* has been added to the speed up the transpose loop. Collapse(n) allows parallelism over n contiguously nested loops. It will specify how many loops in this nested loop should be collapsed into one large iteration space and divided according to the schedule clause. Third, we use *#pragma omp parallel for reduction(+:result_sum).* The *result_sum* is shared. After using the reduction, every thread will computes its own *result_sum* according to the statement of *reduction(+:result_sum).* Then the result will be added up to get the final *result_sum*.

Optimization result: The overall program speeded up from 14.39X to 17.37X and the cases with small sq_dimension had a relatively larger speedup.

c. Goal: Hardware-based and compiler and optimizations

Optimization process: Using the data parallel technology SIMD can operate several data in one operation and SSE can enhance CPU floating computing ability, we try to apply it in our makefile. First, we change -O2 to -O3 to the Makefile and observe the run time. It improved a little bit. So we add the -msse4 to the makefile and observe the result. It improved 0.100ms for test case5 in data_02. The code is shown below:

```
SIMDFLAGS = -msse2 -msse3 -msse4 -O3 -lm
```

Optimization results: The code after adding the SIMD flag speedup to almost 5.95X with the sequential test. For the first two test cases in data_02, the speed even get down. But as the data number increase, test case 3,4,5,6 speedups a lot. Especially test case 5, the speedup almost gets 25X.

2. **K-means clustering (Lloyd's algorithm)**

   As the atomic/non-atomic portions of the computation already had parallelization components, the focus was on reducing I/O time, applying possible parallel reductions on iterative computations in the non-atomic implementation and utilization of compiler-based/hardware-based optimizations.

   a. **Goal: Optimization of I/O operations**

      The approaches used were targeted at accessing/modifying memory in a parallelized manner, and possible use of cache optimization and cache blocking.

      Optimization process: The first step was to perform assignment of initial cluster centers and membership initialization in a using the OMP parallel construct as both these operations involved repetitive assignment or sequential access of arrays. A similar parallel approach was used for the free() calls to the large arrays at the end of the function.

      Optimization results: We expected a significant reduction (15-20%) in I/O time shown but not the computation time. However, the observed reduction was 2-3s in overall execution time for the largest dataset which had an original execution time of ~130s. The optimization did not provide the expected speedup because the major portion of memory access operations are constrained to the reduction of cluster centers, re-evaluation of new cluster centers, etc. and not the memory initialization/release operations.

```
        #pragma omp parallel shared(clusters, objects, membership)
num_threads(8)
  {
     #pragma omp for \
              private(i, j) \
              firstprivate(numClusters, numCoords)
     for (i=0; i<numClusters; i++)
     {
        for (j=0; j<numCoords; j++)
           clusters[i][j] = objects[i][j];
     }

   #pragma omp for \
              private(i) \
              firstprivate(numObjs)
     for (i=0; i<numObjs; i++) membership[i] = -1;
}
```

Optimization process: The second option considered was to use cache blocking for memory access operations in the cluster center reduction and re-evaluation steps (after assigning each object to a cluster). This is based on the idea of trying to fix as much of an array within the cache rows of the L1 cache for quicker I/O and blocking was done in multiples of 8 to identify an optimum block size.

Optimization results: This approach was dropped as we did not observe any appreciable speedup although it should significantly speed up the computations. We believe the reason for absence of speedup was improper implementation of cache blocking along with the already existing OMP parallelism. The speedup due to block-based access might have been hindered by parallel access of memory by each of the threads in the computation.

Overall, the speedup obtained through I/O operation optimization was ~5% and this poor speedup may have been due to

b. **Goal: Use of reductions on iterative computation**

Optimization process: The idea was to perform array reduction of new size of each cluster using the OMP for reduction constructs. At the end of each assignment step, the array containing the number of objects in each cluster is indexed using the thread id that updated it. This array must be reduced to a single dimensional array of cluster id and the number of objects it now contains. The reduction can be done by iterating over the 2-D array with a fixed cluster id and iterative thread id, and summing up the cluster size recorded for each thread id.

```
#pragma omp parallel shared(local_newClusterSize, newClusterSize) \
                        num_threads(8)
    for (i=0; i<numClusters; i++) {
        clusterSize = 0;
        #pragma omp for \
                private(j) \
                firstprivate(i, nthreads) \
                reduction(+:clusterSize)
        for (j=0; j<nthreads; j++) {
            clusterSize += local_newClusterSize[j][i];
            local_newClusterSize[j][i] = 0.0;
        }

        newClusterSize[i] = clusterSize;
    }
```

Optimization results: There are no issues with cache evictions/misses as the number of threads is 8 and number of clusters is 32 which ensures that all data can fit in the L1 cache. The expected speedup was about 25% (I/O and computation time combined) while the observed speedup was around 15%. Again, the reduction was not sufficient to improve overall speedup as there were larger array reductions (such as the one for summation and averaging of cluster centers) that could have instead been optimized for better overall performance.

c. **Goal: Compiler and hardware-based optimizations**

Optimization process: Given that there were repetitive floating point operations on sequential set of array locations, it was best to utilize SIMD support that was available on the cluster. SIMD is known to greatly improve parallelized floating point operations for the same computation. Compiler optimization flags were also added to the Makefile to enabled compilation of code into executables with much lower computation and memory access times. Both –O2 and –O3 were tried (with only minor improvement in speedup).

```
$(CC) $(CFLAGS) $(OMPFLAGS) –O3 –msse4 –c omp_kmeans.c
```

Optimization results: The observed overall speedup of the program was nearly 6x (combined execution time for all testcases). For the first two testcases with relatively small number of data points, the speedup wasn't substantially larger. However, for testcases 3 and 4 with over 100k data points, the speedup was substantial, reducing from 120s+ to ~12s in the sp ecific case of testcase 4. We believe that the parallelized implementations, memory access patterns and loops were all optimized substantially by the GCC compiler and resulting in this dramatic speedup.

References

[1] http://bisqwit.iki.fi/story/howto/openmp/
[2] https://gcc.gnu.org/onlinedocs/gcc-3.4.4/gcc/Optimize-Options.html
[3] https://gcc.gnu.org/onlinedocs/gcc/x86-Options.html
[4] http://www.ece.northwestern.edu/~wkliao/Kmeans/index.html
[5] https://www.mcs.anl.gov/~itf/dbpp/text/node45.html