Tzi Siong Leong (20753794)
Jason Chao (2139 7806)

CITS3402 Project 1 Report

For our matrix collisions project, we sought many ways to improve our program's runtime. We even implemented our project in C++ to take advantage of the STL data structures that it provides to create a more efficient program from the ground up to improve our runtime. We used data structures such as vectors and hashmaps to efficiently retrieve and insert our data. Then, we decided to parallelize our code in the mainfileOMP2.cpp but not finalNeighbors.cpp due to it being slower than sequential. We mainly parallelized our for loops to share work among all the four threads that we set. We also tried different synchronization techniques to parallelize our pushToCollisionTable function, which inserts blocks to the hashmap of collisions. We attempted to use the critical directive, but we resorted to using the lock directives to ensure that only one thread can access the collisionTable hashmap due to the lock routine having a slight increase in performance. This is done to prevent colliding access with the same key among the threads so data can be inserted safely by one thread at a time.

The primary directives that we used to increase the performance of our program are the worksharing constructs for for loops and clauses such as schedule and numthreads. With the numthreads clause, we requested the operating system to give us a set amount of threads in which to parallelize the work across the CPU cores, in our case we used the maximum amount of threads our computer's CPUs could offer, which is four. The schedule clauses are used to divide loops into equal-sized chunks per thread assigned. The chosen type of scheduling was the static type, this type of scheduling pre-calculates the chunks per thread at compilation time, thus lowering the overheads when our code ran. This is in contrast to dynamic scheduling, where threads get an unpredictable amount of iteration chunks and the scheduling happens during at run time rather than at compilation time so therefore we justified using static scheduling for our program.

Considering all our parallelization efforts to improve our code's runtime, we managed to make our sequential program run faster with openMP parallelization by 14.4% on average with a dia of 0.000001 (Table 1). We have conducted a ten sample experiments on both sequential and parallelized code to get an average of improved speed runtime of 9.0454 seconds faster as shown in Table 1. In conjunction to the numerical observations of the program's runtime performance, graph data was also gathered showing the CPU usage during the program's execution to showcase real time runtime CPU usage. Figure 1 shows the CPU usage of the sequential matrix collision code. From the figure, it was observed that only one CPU was primarily used to process the program, as was predicted with sequential code. Figure 2 shows that multiple cores were used for the openMP parallelized code. It shows that all four cores were successfully used to participate in the execution of the program to reduce runtime duration.

Table 1. Sequential and parallelized matrix collision code with dia as 0.000001 running on a machine with Intel core i7-4790k @4.6GHz,16GB RAM, 4 threads non hyperthreaded. The code ran 9.0454 seconds faster on average. It has a 14.4% decrease in programming runtime.

Sequential Code (seconds)	**Parallelized code (with lock routines) (seconds)**
62.700	54.902

62.765	53.978
62.770	52.943
62.559	53.867
62.488	54.366
62.431	53.452
63.117	51.958
62.443	53.660
62.666	54.028
63.296	53.627
Average: 62.7235	**Average**:53.6781

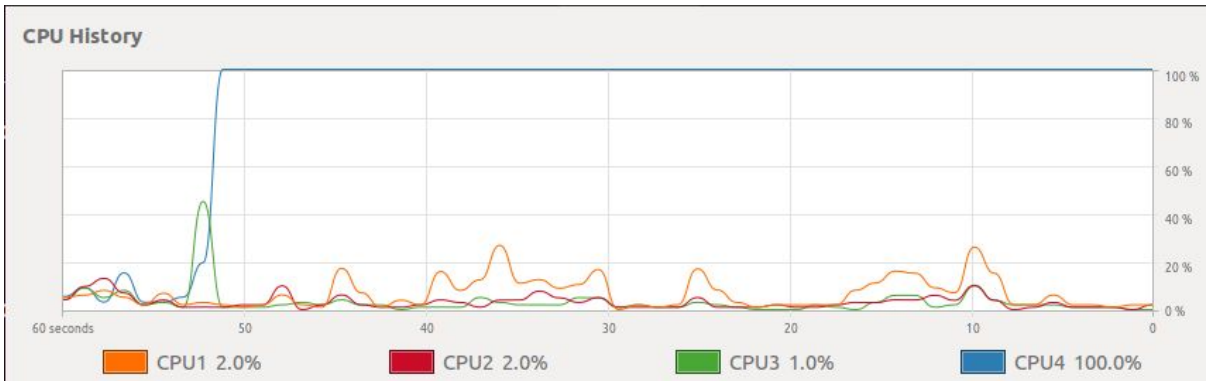

Figure 1. Sequential code runtime graph of CPU usage. This figures depicts that primarily one core/CPU is used when the sequential code executes. X axis shows the how many seconds ago the program ran. Y axis show the percentage of CPU usage.

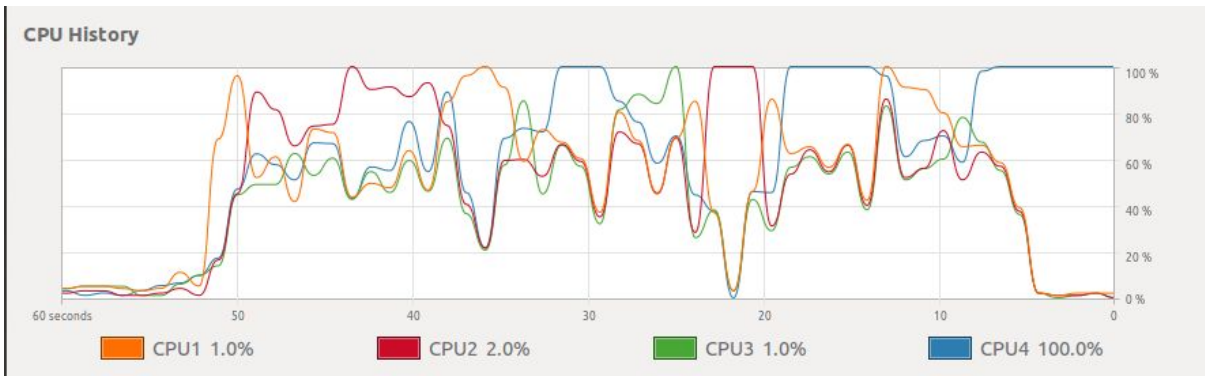

Figure 2. Parallelized code runtime graph of CPU usage. It depicts multicore usage of our parallelized matrix collision code with openMP directives.

The matrix collision code can be further parallelized using lock routines and critical directives. Another set of experiments were conducted to decide which pragma directive yield better performance: the critical and the lock directives. Table 2 results show that critical directives did not have much runtime performance differences with an average of 53.3912 seconds runtime compared to the lock routines performance results with an average of 53.6781 seconds in Table 1. In the end, we decided to use lock routines due to it being more flexible than the critical directive and allowing the parallel code to still run with multiple threads while making sure only one thread is accessing the data element. Another synchronization technique such as atomic was also considered when updating the hashmap but unfortunately the atomic does not work when updating the hashmap by reference.

Table 2. Parallelized matrix collision code running on a machine with Intel core i7-4790k @4.6GHz,16GB RAM, 4 threads non hyperthreaded) using critical directive and dia as 0.000001

Parallelized code (with critical directive) (seconds)
54.968
53.664
54.188
53.196
53.801
51.797
52.879
52.927
52.629
53.863
Average: 53.3912

Table 3. Parallelized matrix collision code running on a machine with Intel core i7-4790k @4.6GHz,16GB RAM, 4 threads non hyperthreaded) using lock routines and dia as 0.000010

Sequential Code with 0.000010 as dia (seconds)	**Parallelized code (with lock routines) and 0.000010 as dia (seconds)**
107.898	87.821
107.569	91.007

107.81	90.398
107.524	90.818
107.652	87.992
106.898	86.732
106.526	85.067
106.475	91.056
107.012	98.774
107.942	87.922
Average:107.3306	**Average**: 89.7587

Experimentations on increased dia to 0.000010 (ten times greater than the original dia of 0.000001) shown in Table 3 shows an average of 17.5719 seconds reduction in runtime compared to the sequential code. That is a 16.37% decrease in programming runtime. This results corresponds similarly to the parallelized code experiments with a dia of 0.000001.

Overall, the matrix collision code showed improvements when openMP directives were used to parallelize our code. The experiments clearly show a reduction in runtime duration along with the graphical figures showing parallelization working in real time with CPU usages. Further experiments on different directives efficiency and performance were also conducted to get the best runtime performance. Although they have minute differences, It was concluded that the lock routines were slightly better than the critical directives to parallelize our code.