ASEN 4057 Final Project - Optimization and Parallelization of C Code for Matrix-Matrix Operations

Joseph Beightol[*], Lara Lufkin [†]

Submitted: May 03, 2018

[*]SID: 104227031
[†]SID: 105927615

University of Colorado Boulder

Contents

List of Figures

List of Tables

I. Description of C Program to Improve

For the final project, the students of the ASEN 4057: Aerospace Software class were given the task to optimize and parallelize C-code from one of the C source assignments given earlier in the semester. The students were given five options to optimize and parallelize matrix operations, the Apollo-13 simulation, two cases from the two-dimensional steady heat equation, or any C-code of ones choosing. The class split up into groups of two to share their knowledge and tackle the final project together.

The C program chosen for this group retained the the optimization and parallelization of the matrix operations. As Aerospace Engineers, matrices are used daily to help simulate and design models, therefore, improving the functionality and timing of code is greatly important in the industry.

Earlier in the semester, the students were given the task to use the compiled language *C*, to accomplish simple matrix operations such as multiplication, addition, and subtraction of matrices. This assignment utilized the built-in library *cblass*, which is a library used to compute matrix operations. The *cblass* library was created to help the functionality of matrix operations as well as speed up the process of computing. While *C* is a compiled language and is already known for its superior speed compared to interpreted languages, the objective of this assignment was to optimize and parallelize the code to make it even quicker.

II. Part I: Serial Code Optimization

The first part of the project was to optimize the original C code for the chosen objective to improve the performance of the code. For this case, the performance of the matrix operations was chosen to be improved. To accomplish such a task, it was recommended that the students optimized their code by either implementing an external library, improving the operation count, or improving the utilization of a memory subsystem. As discussed above, the original C code developed earlier in the semester for the matrix operations already utilized the *cblass* external library. Therefore, it was chosen to write a worse code that computed the matrix operations without and external library, as well as optimize the non-external library code using cache-oblivious algorithm. These results were then compared to see the effect of optimization of code.

A. Profile Report and Discussion of Performance of Original C Code

The original code was determined to be the code that did not utilize any external library or cache oblivious algorithms. This code ran through two different functions, one for multiplication and the other for addition and subtraction. The code ran through many for loops in order to compute all the matrix operations. To see the full effect of the optimized code, the matrices that were implemented were chosen to be 1000x1000 random matrices. Using the built-in function *gprof* the profile summary for the original code was created. The profile summary regarding the timing of the code can be seen in figures 1 and 2

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
100.00     38.84    38.84        6     6.47     6.47  Matrix_Multiplication
  0.08     38.87     0.03                             main
  0.05     38.89     0.02        4     0.01     0.01  Matrix_Addition

```

Figure 1: Flat profile of original code.

```
                     Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) for 0.03% of 38.89 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.03   38.86                 main [1]
                38.84    0.00       6/6           Matrix_Multiplication [2]
                 0.02    0.00       4/4           Matrix_Addition [3]
-----------------------------------------------
                38.84    0.00       6/6           main [1]
[2]     99.9   38.84    0.00       6         Matrix_Multiplication [2]
-----------------------------------------------
                 0.02    0.00       4/4           main [1]
[3]      0.1    0.02    0.00       4         Matrix_Addition [3]
-----------------------------------------------
```

Figure 2: Call graph of original code.

In figure 1, it is noticed that the matrix operations using 1000x1000 matrices is very time costly. The total time for the program to run takes 38.39 seconds to complete. Further analyzing the code the profile summary, most of the time is spent in the Matrix Multiplication function, which makes sense since the code runs through a 1000 by 1000 nested for loop to multiply each element. This needs to be improved by using either optimization or parallelization. For this part of the project, optimization was utilized to improve the performance of the code.

B. Proposed Serial Code Improvements

In regards to optimizing the code, the group proposed to implement two strategies for serial code improvements: the already developed code utilizing the external library *cblass*, and the cache oblivious algorithm. It was chosen to implement both of these methods since the external library was already implemented in an earlier assignment, but still wanted to see the overall effect of all methods. The built-in library *cblass* uses basic linear algebra subprograms to preform basic vector and matrix operations such as addition, subtraction and multiplication. This is seen to improve the performance of the code greatly and can be seen in the following sections.

Utilizing the cache-oblivious algorithm, the code was able to minimize the cost of memory access and latency. The code was edited to reuse the memory in cache instead of storing the memory and saving it. The code did this by breaking up the multiplication into chunks that reused the stored elements to compute the final product. This also was seen to have a great effect on the performance of the code and can be seen in the following sections as well.

C. Timing Comparison of Original C Code and Optimized C Code

The three models: original, cache, and built-in library codes were then compared to see which code was ultimately faster. All the codes were run with the same 1000x1000 matrices and used 8 processors for the computation. The results regarding the timing of each code is displayed below in table 1. In this table it

is seen that the Original Code takes the longest at 44.43 seconds, which was to be expected. Implementing the cache optimization, the execution of the code is cut in half to 22.16 seconds. Finally, utilizing the *cblass* library, the code was able to reduce to 17.98 seconds. This further proves that optimization has a great impact on coding and should be utilized while computing large tasks. Overall however, the built-in library was the most efficient way to improve the original Code.

Table 1: Timing Comparison Table

	Origonal Code	Cache Optimization	External Library Implementation
Time (seconds)	46.240114	22.1614	17.9787

D. Profile Report and Discussion of Performance of Optimized C Code

While optimizing the Original Code, two techniques were employed to optimize the code. The first method was using the cache-oblivious algorithm. The profiling reports for the cache-oblivious algorithm are depicted below in figures 3 and 4. In the profiling report, it is seen that it takes significantly less amount of time to execute compared to the original code. It still spends most of it's time in the Matrix Multiplication function, but the speed inside this function is greatly increased by reusing the memory stored in cache.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 99.80     18.62    18.62        6     3.10     3.10  Matrix_Multiplication_OptCache
  0.21     18.66     0.04                             main
  0.11     18.68     0.02        4     0.01     0.01  Matrix_Addition
```

Figure 3: Flat profile of cache-oblivious algorithm.

```
                     Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) for 0.05% of 18.68 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.04   18.64                 main [1]
                18.62    0.00       6/6           Matrix_Multiplication_OptCache [2]
                 0.02    0.00       4/4           Matrix_Addition [3]
-----------------------------------------------
                18.62    0.00       6/6           main [1]
[2]     99.7   18.62    0.00       6         Matrix_Multiplication_OptCache [2]
-----------------------------------------------
                 0.02    0.00       4/4           main [1]
[3]      0.1    0.02    0.00       4         Matrix_Addition [3]
-----------------------------------------------
```

Figure 4: Call graph of cache-oblivious algorithm.

The second method was utilizing the already used, built-in external library *cblass*. In reality, the timing of the code using the external library took approximately 17.98 seconds. While implementing the *gprof* function to provide a profile report for the external library, the report did not provide any information regarding the time spent using cblass. Therefore, the report only provides the time spent in the main source, which was 0.04 seconds. This is because the built-in library is not a separate function created for the source code and is just a built-in library that computes the basic matrix operations and saves them in memory. The profiling report regarding the performance of the external library code is depicted in figure 5.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
100.15      0.04     0.04                             main
```

Figure 5: Flat profile of code using external library.

III. Part II: Parallelization of Optimized Code

The second part of the assignment required the parallelization for either the optimized code or the Original Code. Since the optimized code was done using the built-in library, *cblass*, the group decided to parallelize the Original code form the first part of the Final project.

A. Proposed Strategy for Code Parallelization

Working through the different ways of parallelization, Open MP was chosen due to its efficiency and user friendly nature. Open MP is a way for the code to split up the work onto multiple processors to accomplish the task more efficiently. If a function is called multiple times, the code will communicate to later parts of the code to complete these tasks in an orderly manner. Open MP was implemented in the matrix multiplication function to optimize the for loop used. Since there are no dependent variables in these for loops, each loop can be run independently at the same time. This makes the for loop used to compute the matrix a perfect aspect of the code to parallelize. The cache matrix multiplication was not used in this parallel code due to its dependence on stored matrix values in the cache. If each processor edits and uses these values it will make the parallelization slower than the original run.

B. Scalability Study on Virtual Machine with up to Sixteen Processes

The next step for part II of the project was to run the parallelized code utilizing different numbers of processes. Utilizing more processes enable the computation to be split up into more cores and compute quickly and more efficiently. The results displaying the time for the parallelized code to execute for different number of processes is depicted in table 2 below. Here, it is seen that the timing only using one process takes a great deal of time at 57.87 seconds. This is to be expected however, as only using one process cannot split up the function calls to execute the task. As the processes increase, the time taken to execute the program experiences a rational function as it exponentially decays to approximately 15 seconds. This is shown in figure 6. This shows that at some point, increasing the number of processes will eventually flat line and not decrease the time of the code.

Table 2: Scalability Study Table

Number of Processors	1	2	3	4	5	6	8
Time	57.87	32.05	25.25	21.75	20.69	19.94	17.48

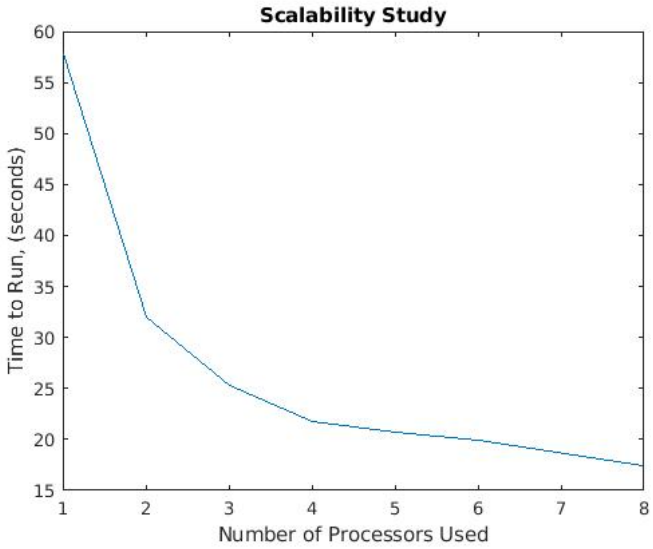

Figure 6: This Figure Plots the data shown in Table 2

C. Profile Report and Discussion of Performance on Virtual Machine

Figures 7 shows the flat profile report for the parallel code using only one processor. Figure 8 shows the call graph for this same profile report. A profile report for code using openMP and more than one processor can not be produced using gprof. This is due to the fact that gprof is not compatible with openMP. Instead of producing a profile report, the function omp_get_wtime was used to time the parallelized code. Using

this method, we found that the parallelized code took 17.484 seconds to run using 8 processors. (*NOTE: Professor Matsuo assured us that it is okay to not include the profile report for 8 processors as long as we show the run time of the code.)

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
100.08     41.18    41.18        6     6.86     6.86  Matrix_Multiplication
  0.05     41.20     0.02        4     0.01     0.01  Matrix_Addition
```

Figure 7: Flat profile of Parallel code.

```
                     Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) for 0.02% of 41.20 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00   41.20                 main [1]
               41.18    0.00       6/6           Matrix_Multiplication [2]
                0.02    0.00       4/4           Matrix_Addition [3]
-----------------------------------------------
               41.18    0.00       6/6           main [1]
[2]    100.0   41.18    0.00       6         Matrix_Multiplication [2]
-----------------------------------------------
                0.02    0.00       4/4           main [1]
[3]      0.0    0.02    0.00       4         Matrix_Addition [3]
-----------------------------------------------
```

Figure 8: Call graph for parallel code.

IV. Summary of Findings and Potential Future Improvements

Overall, this project was a great way of showing how there are multiple ways to improve code. The project taught the students about different libraries and how reusing cache memory could substantially reduce the timing of the code. The students were also introduced to the idea of parallelizing code using *MPI*.

In regards to future improvements to employ to the code, it is believed that the code could become more efficient if a different MPI method was used. With Open MPI being an easier way to parallelize the code, the performance of the code only improved by a little bit. Also, if allowed more time, it is believed that if the cache optimized code was parallelized, the code could have been much quicker.