ASEN 4057 Final Project: Matrix-Matrix Multiplication

Team 7
May 5, 2017
Matt Funk and Jake Ursetta
University of Colorado, Boulder, CO, 80309, United States

I. Description of C Program

The C program which was the focus of this project involved matrix-matrix multiplication. The multiplication of an n x m matrix A and an m x p matrix B into a new n x p matrix, C, can be described by Equation 1 below.

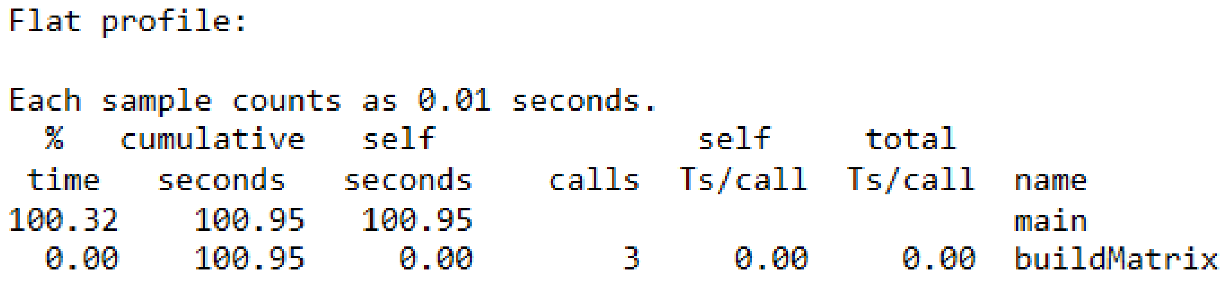
$$C_{ij} = \sum_{k=1}^{m} A_{ik} B_{kj} \tag{1}$$

Throughout the course of the semester, several serial improvements were made to the basic implementation of this equation. These improvements included both using external libraries and minimizing the cost of memory access/latency. With regards to external libraries, BLAS was utilized. This library is able to perform matrix-matrix multiplication far more quickly than most user written programs due to it being able to fine tune block sizes. The matrices being multiplied are split up into appropriate blocks to minimize cache losses by obtaining information about a machines' processor and cache sizes. To minimize the cost of access and latency, several improvements were made to the naive implementation in class. These included pre-fetching and reusing data and minimizing cache misses via blocking. However, the block sizes used were selected based on the cache size of the individual machine being used. The goal of this project was to implement a cache-oblivious algorithm which would allow the number of cache misses to be minimized for any machine. Once this was achieved, the program could easily be parallelized by splitting the required block operations between the number of cores that the machine has.

II. Part 1: Serial Code Optimization

A. Original C Code

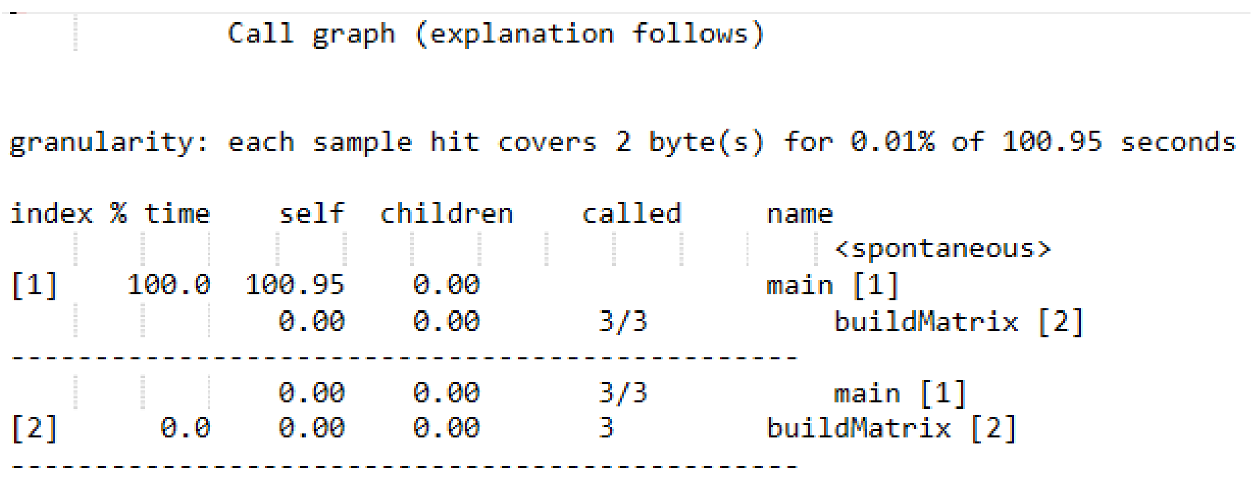
The initial matrix-matrix multiplication algorithm used was a naive implementation. This implementation does a poor job of reusing memory in cache, hiding memory latency, and has a poor access to floating point operation ratio. The basic algorithm discards the jth column of the B matrix after it is used only once. Thus, the program must load the (j+1)st column of B from memory. The time to load data from RAM takes far longer than loading it from cache; this is known as cache miss. In the worst case, (m*n*p) cache misses can occur; this can result in the run time being dominated by these cache misses rather than the actual operations. The disadvantages of using the naive implementation can be clearly demonstrated by multiplying two 2000 x 2000 matrices. The gprof flat profile and call graph for this operation can be seen in Figures 1 and 2 below.



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
100.32    100.95   100.95                             main
  0.00    100.95     0.00        3     0.00     0.00  buildMatrix
```

Figure 1: Naive Implementation Flat Profile



```
            Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) for 0.01% of 100.95 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0  100.95    0.00                 main [1]
                0.00    0.00       3/3           buildMatrix [2]
-----------------------------------------------
                0.00    0.00       3/3           main [1]
[2]      0.0    0.00    0.00       3         buildMatrix [2]
-----------------------------------------------
```

Figure 2: Naive Implementation Call Graph

As can be seen, the total time to run this program was 100.95 seconds. Essentially all of this time is spent in the main function although the buildMatrix function is called 3 times. This function simply dynamically allocates space for the A, B, and C matrices so it makes sense that its time is insignificant. The timing tool in C was used to determine the actual time that it took to perform the matrix-matrix multiplication. This time was 100.56 seconds, meaning that reading in the A and B matrices and printing the C matrix only took a total of 0.39 seconds. The calculation performance is very sub-par as it takes over a minute and a half to multiply only relatively large matrices. This would be extremely inconvenient when it is important to obtain results in a small amount of time. This poor performance helps motivate the need for serial improvements to this naive implementation.

B. Proposed Serial Code Improvements

The most viable serial improvement which could be made to this program would be making it cache-oblivious. To do so, it is first necessary to identify the largest matrix operations which could take place in any cache, regardless of the machine. Upon doing some research, it was determined that this base size is typically 16 x 16 x 16. This means that no matter what machine the program is being run on, the cache will be able to perform matrix-matrix multiplication for two 16 x 16 matrices. Thus, given two very large matrices, it is necessary to separate each into blocks which can be handled in cache. It is possible to develop a simplified algorithm to do this for only square 2^n matrices, but it was desired that the program be able to handle all arbitrarily sized matrices. To implement this algorithm, the base case and three recursive cases have to be considered. The base case only occurs if m, n, and p are less than 16; the recursive cases occur otherwise. If the maximum dimension between m,n and p is n, then A is split horizontally. This results in Equation 2 below.

$$C = \begin{pmatrix} A_1 \\ A_2 \end{pmatrix} B = \begin{pmatrix} A_1 B \\ A_2 B \end{pmatrix} \tag{2}$$

Otherwise if the maximum dimension is p, then B is to be split vertically, resulting in Equation 3.

$$C = A \begin{pmatrix} B_1 & B_2 \end{pmatrix} = \begin{pmatrix} AB_1 & AB_2 \end{pmatrix} \tag{3}$$

Finally, if m is larger than n and p, both A and B must be split. A should be split vertically, while B should be split horizontally. This can be described by Equation 4.

$$C = \begin{pmatrix} A_1 & A_2 \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \end{pmatrix} = A_1 B_1 + A_2 B_2 \tag{4}$$

If more than one of the dimensions are the same, the recursive cases occur in the order above. Each time a recursive case occurs, the A and/or B matrix as well as the C matrix are blocked into smaller pieces.

This recursion continues until the base case is met. From there, the naive algorithm for matrix-matrix multiplication occurs, leading to minimal cache misses. From here the C matrix is built back up going in the reverse order that the recursive cases occur. The algorithm calculates small blocks of the C matrix at a time and uses pointer notation to place these values back in the correct place in the full C matrix.

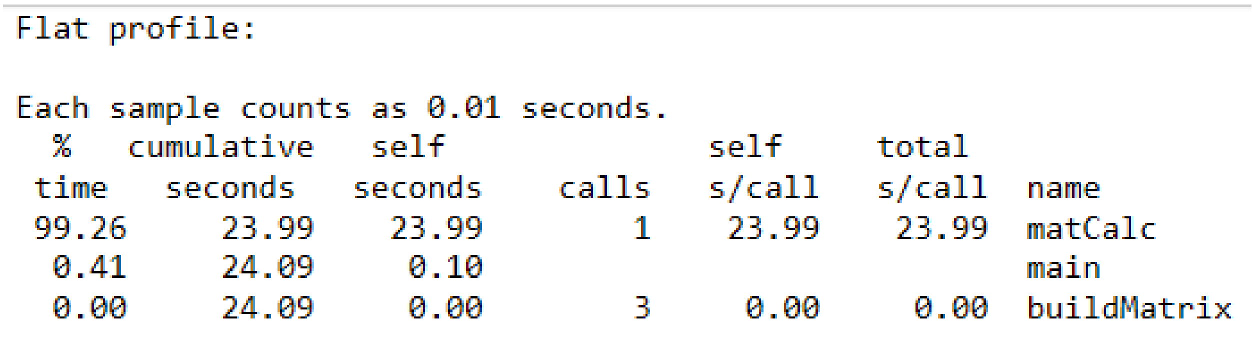
In addition to creating this cache oblivious algorithm, the program should also be modified to reduce the number of floating point operations per memory access. To do so, data pre-fetching can be utilized for the actual matrix-matrix multiplication. In the case where m,n, and p are all less than 16 and n and p are also divisible by 2, 3 data pre-fetches can occur when the (i,j)th element of C is being calculated. These are: the (i+1,j)th element, the (i,j+1)st element, and the (i+1,j+1)st element. By loading these elements when the original element is being calculated, the operations per memory access is increased while the number of memory accesses which must occur is reduced. In the case where n and/or p is not divisible by 2, pre-fetching is not used to simplify the coding process. Both of these improvements should greatly reduce the amount of time for which the matrix-matrix multiplication takes.

C. Timing Comparison

After implementing the above serial improvements into the C code, the multiplication of the same 2000 x 2000 matrices for the naive implementation was done. The time to complete for this case was 23.99 seconds. Compared to the naive implementation, this is an improvement of 76.57 seconds. This means that making the algorithm cache oblivious and using data pre-fetching leads to a 420 % improvement in performance. This performance difference will only continue to increase as larger matrices are used; the cache misses of the naive implementation will increase exponential. However, in the interest of time, a larger comparison between these two algorithms was not done. Regardless, it is far more effective to utilize a cache-oblivious algorithm for matrix-matrix multiplication, as expected.

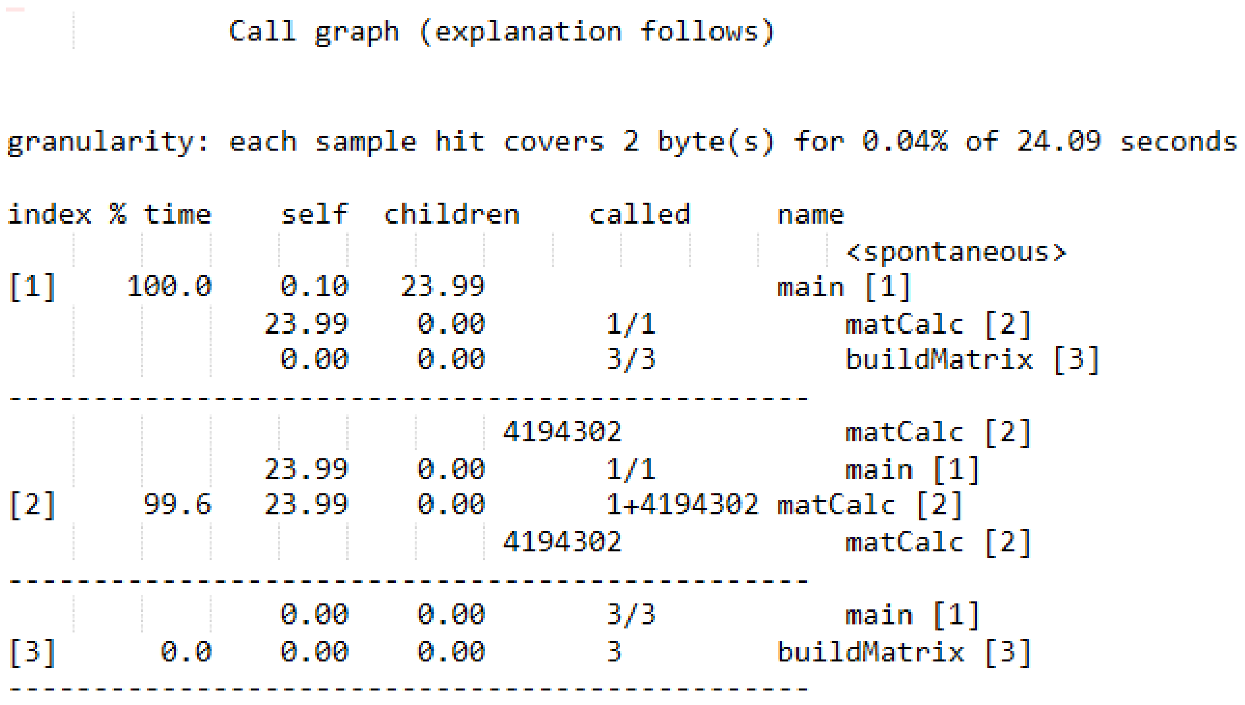
D. Optimized C Code

The performance of the serial optimized code can be further explored by looking at the gprof profile report. The flat profile is displayed in Figure 3 while the call graph can be seen in Figure 4.


```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 99.26     23.99    23.99        1    23.99    23.99  matCalc
  0.41     24.09     0.10                             main
  0.00     24.09     0.00        3     0.00     0.00  buildMatrix
```

Figure 3: Serially Optimized Flat Profile



```
                    Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) for 0.04% of 24.09 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.10   23.99                 main [1]
               23.99    0.00       1/1           matCalc [2]
                0.00    0.00       3/3           buildMatrix [3]
-----------------------------------------------
                             4194302             matCalc [2]
               23.99    0.00       1/1           main [1]
[2]     99.6   23.99    0.00       1+4194302 matCalc [2]
                             4194302             matCalc [2]
-----------------------------------------------
                0.00    0.00       3/3           main [1]
[3]      0.0    0.00    0.00       3         buildMatrix [3]
-----------------------------------------------
```

Figure 4: Serially Optimized Call Graph

In the two reports above, it can be seen that the total time to run the entire program was 24.09 seconds. The matCalc function performs all of the recursive blocking and the base case matrix-matrix multiplications. This explains why in takes up 99.26 % of the total time with its one call. The main function reads in and prints the matrices which does not take a significant amount of time with respect to the total. Again, the time spent in the buildMatrix function is insignificant. The improvements greatly increase the performance of the program, however, it leaves some things to be desired. 24 seconds is not especially long for one matrix multiplication operation, but this would add up if several operations are required. This motivates further optimization of the code. To do so, parallelization must be considered.

III. Part 2: Parallelization of Optimized Code

A. Proposed Parallelization

Following the implementation of the serial optimization, the code could be parallelized to allow multiple processors to be used at once. To do so, a similar algorithm for the serial case can be utilized along with some MPI commands. After initiating MPI (with MPI_Init), the first step is to determine the number of processors available as well as their rank. This is done through MPI_Comm_size and MPI_Comm_rank, respectively. The number of processors can be specified when running the program with mpirun. Once the number of processors is known, the commands which only need to be done once, such as reading in the matrices, can be sent to only the server (processor with rank=0). From here, the rows of the A matrix can be divided based on the number of processors. Due to the fact that the number of rows in A will not necessarily be perfectly divisible by the number of processors, the remainder from the division must be considered. The following algorithm was implemented to take care of this. Note that n is the number of rows in A, as before, while N_p is the number of processors.

$$Divisions = floor\left(\frac{n}{N_p}\right), Remainder = n - N_p * Divisions \tag{5}$$

From here, the processors can be looped through to give each a division. If the number of the loop iteration is less than the remainder, then the processor will also receive an additional row of A. Otherwise, the processor will only receive its division. The necessary information for completing the calculations are then sent to each of the processors. Note that the server will also be doing a portion of the calculations. MPI_Send is used to send the allocated locations of A, the inner dimension of A, the outer dimension of B, the actual data in the allocated A rows, and the entire B matrix to each of the other processors. The server then utilizes the previously described cache-oblivious algorithm to calculate its portion of the full C matrix. At the same time, all of the other processors use MPI_Recv to receive all of the data sent from the server. These processors implement the same cache-oblivious algorithm to calculate their portions of the full C matrix. Once these calculations are done, the calculated blocks of the C matrix are sent back to the server. The server loops through each of the addition processors, receives their data, and compiles the entire C matrix. Finally, MPI_Finalize is used to finish to program.

B. Scalability Study

Amdahl's Law places a limit on the speed-up of parellelizing a program; it is given by Equation 6 below.

$$t_N = \left(\frac{f_p}{N} + f_s\right) t_1 \tag{6}$$

Here, f_p and f_s are the fractions of parallel and serial code, respectively. N is the number of processors and t_1 is the time to run on one processor. By implementing the proposed parallelization in the section above, the scalability of the program with respect to the number of processors can be studied. This was done for 1, 2, 4, 8, and 16 processors. It was desired that very large matrices be used for this study. However due to the fact that this program was being run on a virtual machine from a USB, some performance issues resulted from attempting both 10000 x 10000 matrices and 5000 x 5000 matrices. These are further explained later. Thus, the same 2000 x 2000 matrices were used for each of the cases. The times to run each of the cases are summarized in Table 1 below.

Table 1: Parellelization Run Times

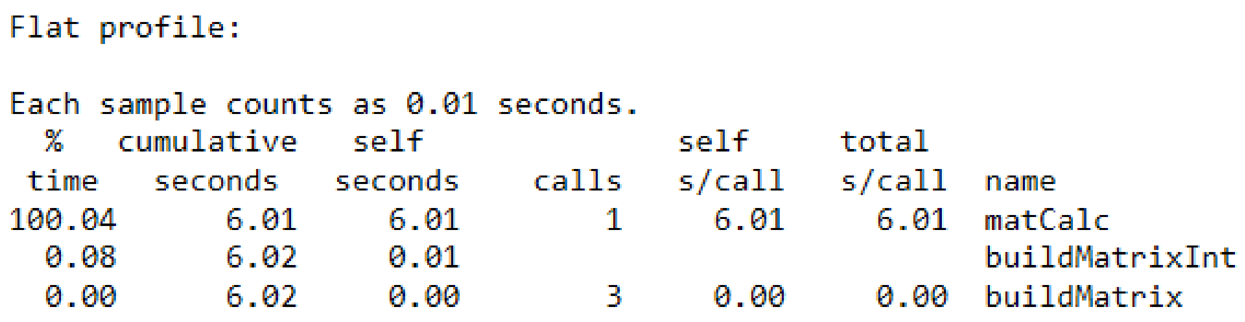
Number of Processors	MM Time (s)
1	24.27
2	12.73
4	6.40
8	6.01
16	3.15

Almost all of the calculations done in the program are parallelized (other than reading in and printing matrices), thus f_p is likely about 0.99 while f_s will be about 0.01. Using these values and Equation 6, it is expected that the run time for 2 processors should be about 12.26 seconds. The actual run time of 12.73 seconds is slightly longer than this; this can be attributed to f_s not being exactly correct or the start-up time of MPI. For 4 processors, the expected run time is 6.25 seconds. This is quite close to the actual run time of 6.40 seconds. When switching from 4 processors to 8 processors, the performance increase drops off significantly. The expected run time for 8 processors is 3.25 seconds, but the actual run time is far higher at 6.01 seconds. The reason for this drop-off is likely due to the machine itself. The CAD lab computer which these calculations were being run on had a total of physical processors with a maximum of 8 total processors. Thus, by increasing the processors used to 8, the performance will likely decrease; this is the case here. However, when utilizing 16 processors, the run time improves significantly. Although the 3.15 seconds run time is far higher than the expected run time of 1.74 seconds, the improvement from 8 cores in around 190 %. So, the drop off in performance improvement is not nearly as significant for switching from 8 to 16 processors as it is for switching from 4 to 8 processors. From this study, it is clear that Amdahl's law for parallelization time scalability is followed closely for up to four processors on this machine. However, once more than 4 processors are used, the approximately two times improvement with twice as many processors tends to break down.

C. Discussion of Performance

To further assess the performance of the parrallelization of this matrix-matrix multiplication program, a profile report can be analyzed. The flat report and call graph for the 8 processor case can be seen in Figures

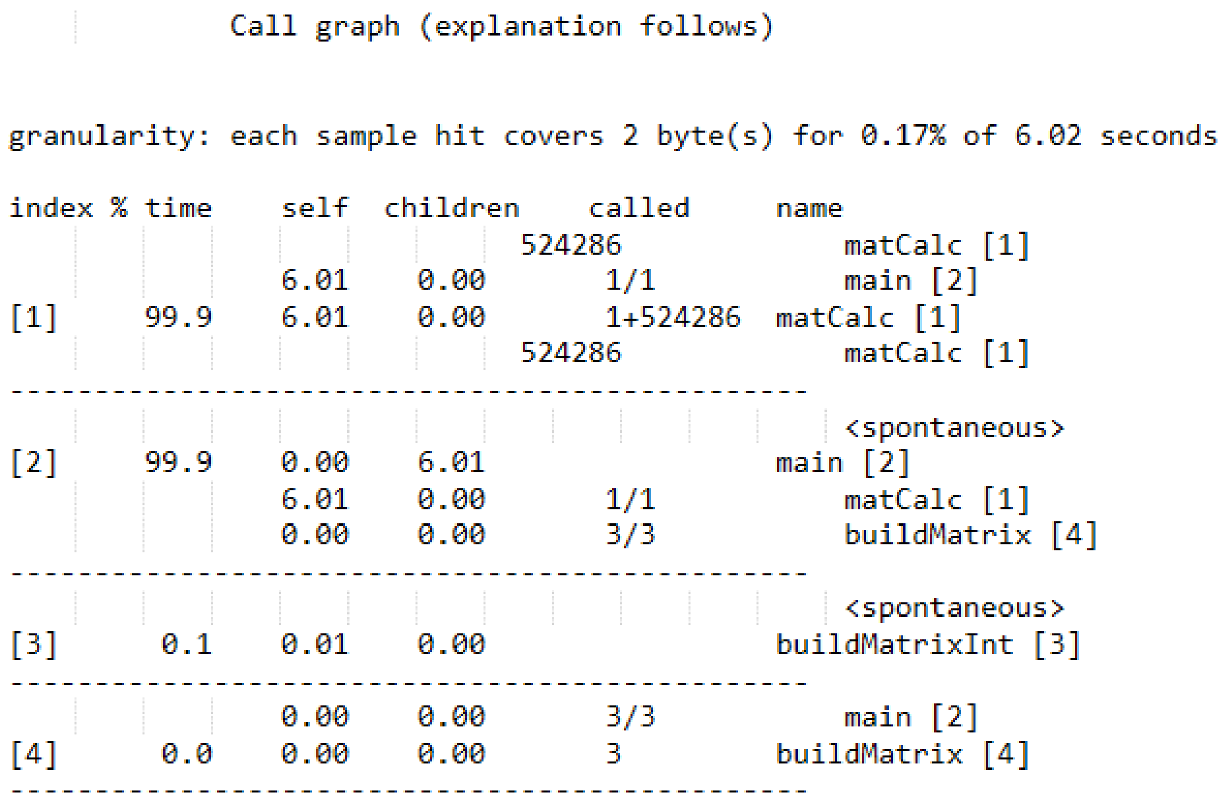
5 and 6, respectively.



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
100.04      6.01     6.01        1     6.01     6.01  matCalc
  0.08      6.02     0.01                             buildMatrixInt
  0.00      6.02     0.00        3     0.00     0.00  buildMatrix
```

Figure 5: Parallelized with 8 Processors, Flat Profile



```
            Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) for 0.17% of 6.02 seconds

index % time    self  children    called     name
                             524286             matCalc [1]
                6.01    0.00       1/1           main [2]
[1]     99.9    6.01    0.00       1+524286  matCalc [1]
                             524286             matCalc [1]
-----------------------------------------------
                                                 <spontaneous>
[2]     99.9    0.00    6.01                 main [2]
                6.01    0.00       1/1           matCalc [1]
                0.00    0.00       3/3           buildMatrix [4]
-----------------------------------------------
                                                 <spontaneous>
[3]      0.1    0.01    0.00                 buildMatrixInt [3]
-----------------------------------------------
                0.00    0.00       3/3           main [2]
[4]      0.0    0.00    0.00       3         buildMatrix [4]
-----------------------------------------------
```

Figure 6: Parallelized with 8 Processors, Call Graph

The total run time of the program is 6.02 seconds, of which 6.01 seconds is taken up by the matCalc function. Only 0.01 seconds are spent in the buildMatrixInt and buildMatrix functions together, which allocate the size of the matrices to be used. Looking at the call graph, it can be seen that matCalc is only called once, however, it calls itself recursively 524286 times. This means that this many divisions of the A matrix are required to implement the cache oblivious algorithm for this parallelized program.

As mentioned above, the performance on the virtual machine is degraded when matrices larger than 2000 x 2000 were used. Even generating larger matrices with random numbers took a long amount of time. Dealing with such large matrices when running the virtual machine from a flash drive proved to be an issue. This is likely due to the clock speed of the USB not being large enough to accommodate the processor speed on the computer itself. Additionally, when allocating the maximum 8 processors to the virtual machine as well as a large amount of the total RAM, the computers' operating performance likely degraded significantly. For this reason and in the interest of time, only one larger matrix multiplication was carried out. This involved two 5000 x 5000 matrices and was run on 16 processors. The resulting run time was 47.62 seconds.

IV. Summary of Findings & Potential Future Improvements

In conclusion, the run time of a matrix-matrix multiplication algorithm can be extremely improved by implementing both serial optimization and parallelization. With a basic, naive implementation, the run time for the multiplication of two 2000 x 2000 matrices is 100.95 seconds. This can be improved greatly by making the algorithm cache-oblivious. By breaking down the large matrices into blocks small enough to fit in any cache, the time spent on cache misses can be optimally reduced. This leads to a total run time of only 23.99 seconds. However, this run time can be improved even more significantly by implementing parallelization. Using a distributed parallelism model with MPI, the number of processors which perform the calculations can be increased. It was found that the run time follows Amdahl's law up for up to 4 processors; the run time decreases by about 50 % when doubling the number of processors. However, this improvement trend tends to break down when more processors (8 and 16) are used, at least on the machine used. The run times for 1, 2, 4, 8, and 16 processors were: 24.27, 12.73, 6.40, 6.01, and 3.15 seconds, respectively.

A potential future improvement for the matrix-matrix multiplication algorithm developed is to improve the implemented data pre-fetching. Pre-fetching is only done when n and p are both even. The code could be changed to allow for pre-fetching to occur for any matrix. This would involve pre-fetching when possible (at the beginning of the matrix) and regular data collection when not possible (at the end of the matrix). Implementing this improvement would move to code to being almost completely optimized. From here, any improvements would seem unnecessary, as the BLAS external library would likely be just as effective.