Summary:

In the RAM computing model, data is stored in memory, cache, and the registers throughout the process cycle. Knowing how these data stores operate, we can develop more efficient programs. In this assignment, you will compare four different algorithms for multiplying two matrices.

Deliverable:

Turn in one pdf file having 2 graphs (1) BSIZE vs. Time (for finding "ideal" block size) and(2) N vs. Time (for finding "fastest" algorithm). This file should also have a report answering the questions outlined below. This pdf file will be named "asgn4_[your_eid].pdf", where eid is your VCU eID.

Goals

1. Is the time returned correct for all the algorithms? If not, which algorithm or algorithms are incorrect and why?

The time for the blocked algorithm was accurate to within about 10% of the actual time. I used a stopwatch for the processes that took multiple minutes and my time and the printed time were relatively close.

The time for BlockedMulti was not accurate for some values. In these cases the program would print very small times such as 0.05 sec when the actual time the program took to run was over a minute. This inaccuracy was caused by the algorithm's use of forks. The printed time is the CPU time used by each fork. If the BSIZE was larger than the value of n, the printed time was accurate. An example of these different accuracies can be seen with n = 2000 with BSIZE = 2000 and BSIZE = 4000. When BSIZE was 2000, the printed time was 0.19 sec but the actual time was about 5 min 50 sec. When BSIZE was 4000, the printed time was 0.20 sec, but in this case the time was accurate (or at least seemed to be since the program finished almost instantly). In the cases where the BSIZE was larger than the values of n, the program would only have to use one block, which was sometimes faster. The largest value of n that I was about to run quickly was n = 10000 with BSIZE = 100000 and it took about 3.5 sec. Larger values caused errors with memory size.

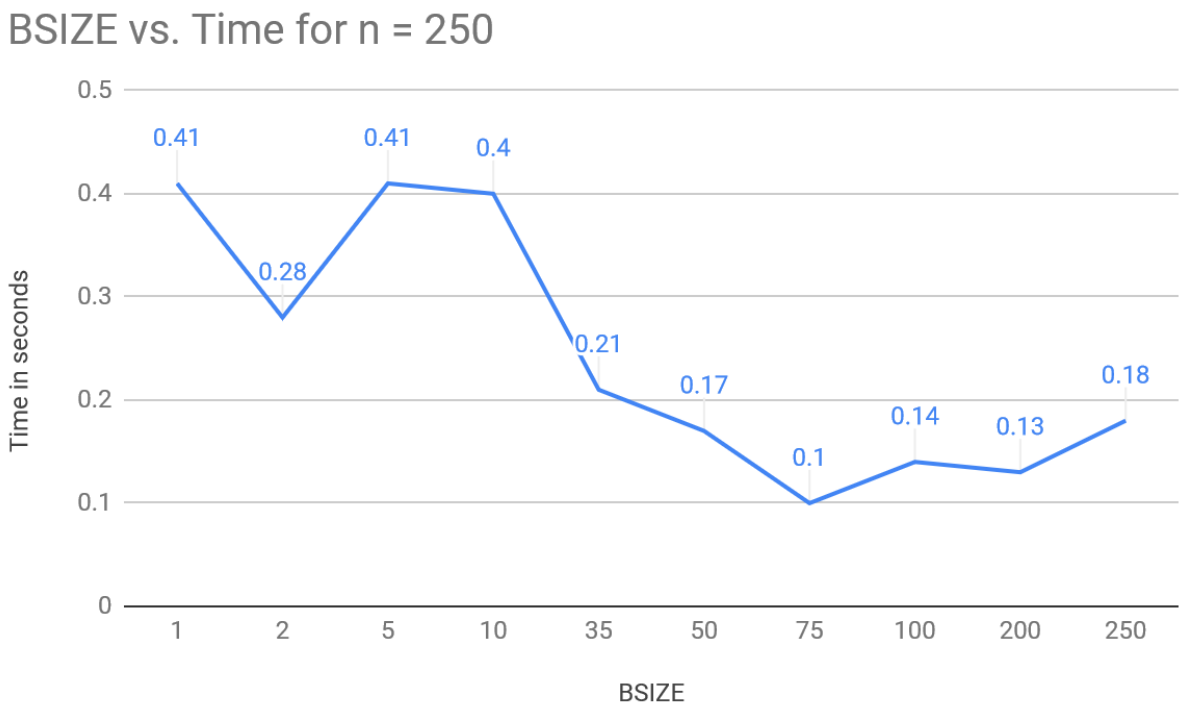
The time for the normal algorithm was accurate for all of the values that I tried.

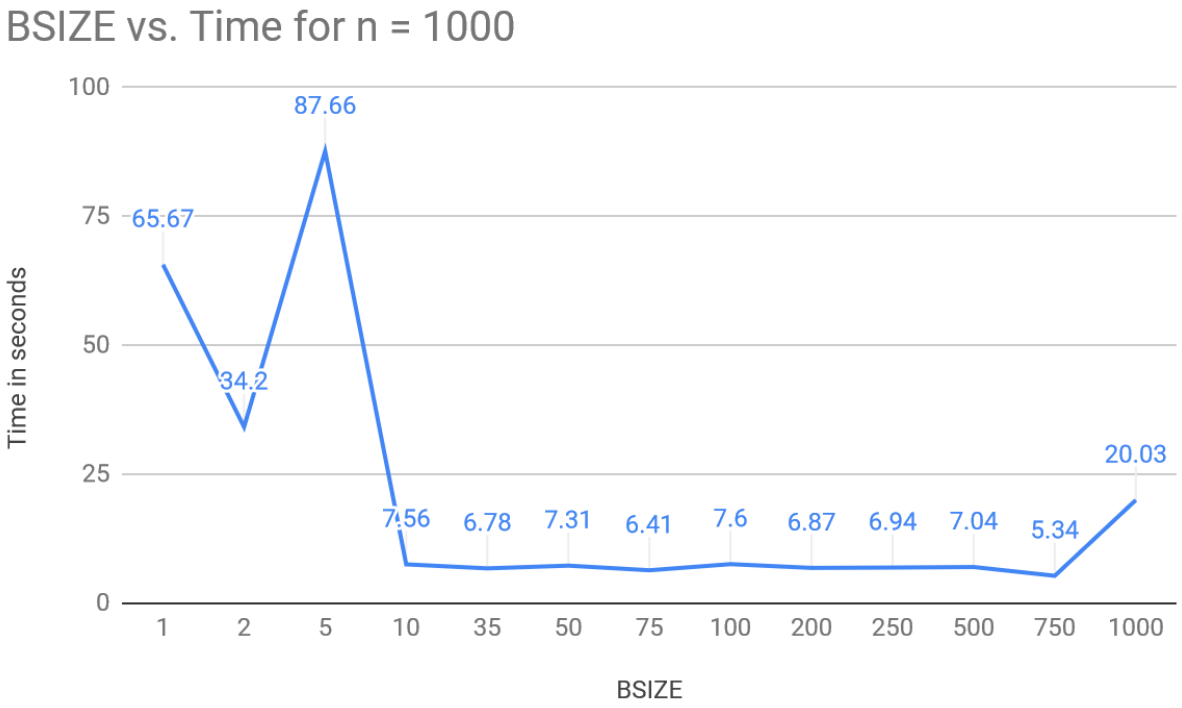
The time for the transposed algorithm was accurate.

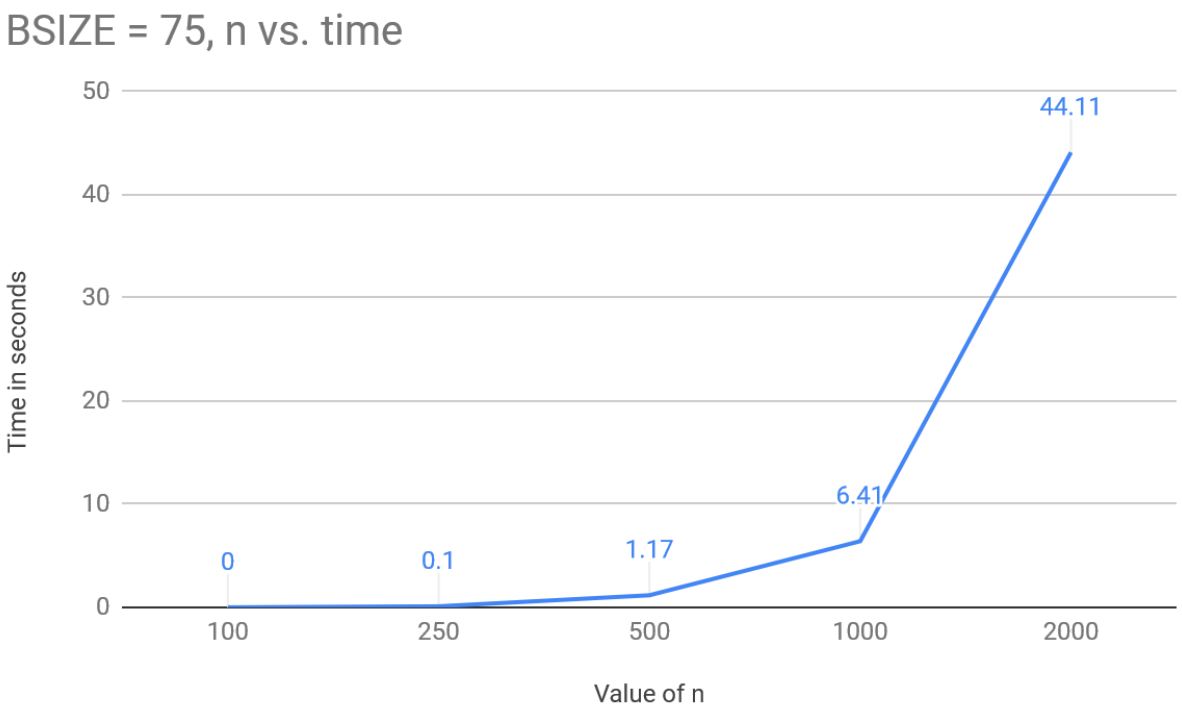
2. Determine "ideal" block size. Run several test cases for Blocked Matrix Multiplication with different values of BSIZE. Use these test cases to determine the ideal BSIZE. These tests should be represented as a graph of BSIZE vs. Time.

In the example of n = 1000, BSIZEs of 750 and 75 are the best. Because 75 was the best in the case of n = 250 and BSIZE of 750 wouldn't work for n = 250, I am using 75 as the ideal size.

Using a BSIZE of 750 for n = 250 doesn't work correctly because the program tries to use a block size larger than the matrix, leading to the algorithm functioning like the Normal algorithm or working incorrectly. I found issues with the program when I made the block size significantly larger than the value of n such as BSIZE = 100000 and n = 10000 runs in a few seconds but this doesn't make sense since the program should be taking a huge input and running as efficiently as the Normal algorithm.

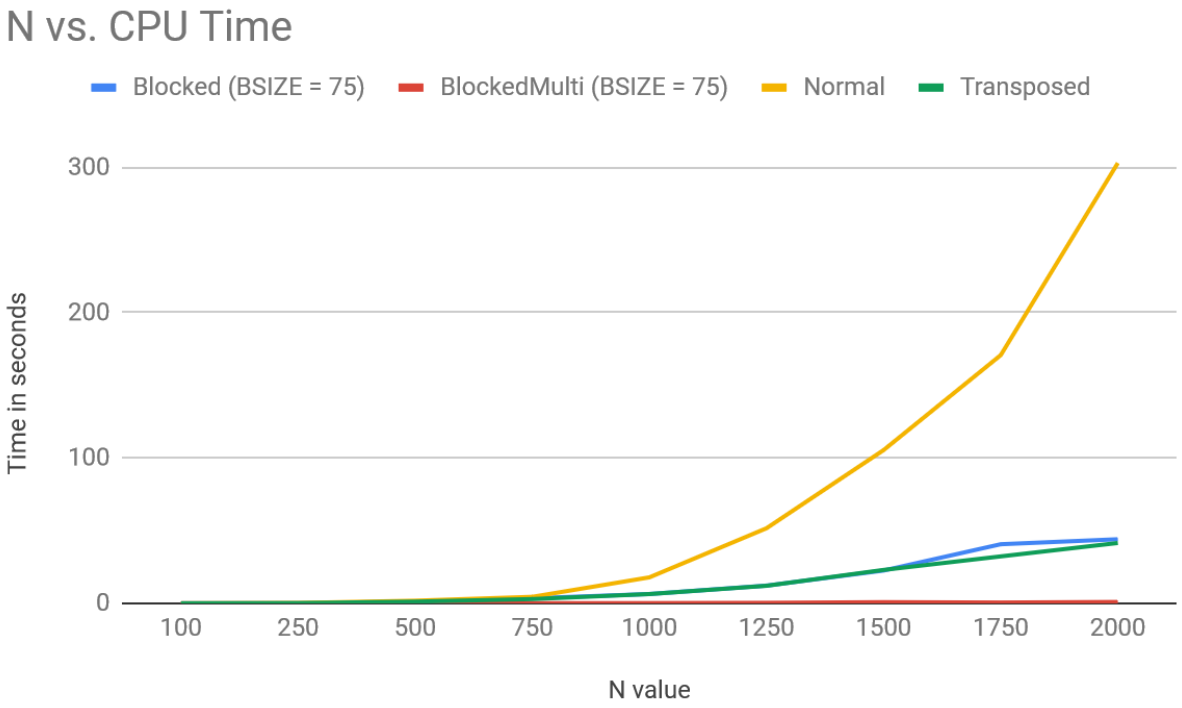


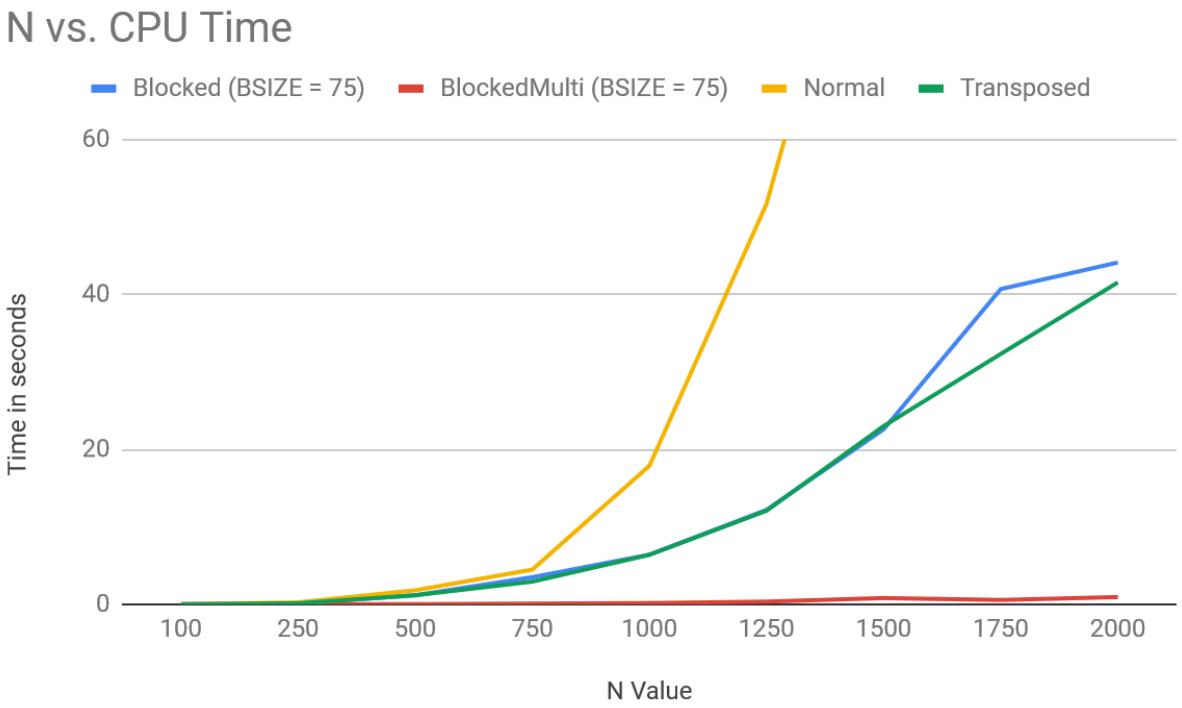




3. Determine “fastest” algorithm. Run several test cases for Normal, Transposed, Blocked, and Forked Blocked with different values of n. Use these test cases to determine the

fastest algorithm. These test should be represented as a graph of N vs. Time, with all four algorithms shown in one graph.





I have two graphs here with the same data. The top graph shows how quickly Normal increases compared to the other graphs and the bottom graph shows more closely the differences between the Blocked and Transposed algorithms.

4. Explain the idea behind each algorithm (why is Transposed considered efficient? Why is Blocked considered efficient?).

Blocked – The algorithm splits the matrix into smaller matrices and does multiplication on them sequentially. Since this code involves 5 nested for loops, it can be difficult to see how it works. After researching some about block matrix multiplication online, I found that this algorithm is fast because it requires fewer transfers with memory (source: http://www.cs.cornell.edu/~bindel/class/cs6210-f12/notes/lec02.pdf)

MultiBlocked – this algorithm functions mostly the same way as Blocked, but it uses fork() to split the program into multiple processes, one process for each block. While this seems like it would be faster, because the program was written so that it uses wait(NULL) to handle collisions, it ends up running slower than Blocked. If the program worked better with fork(), it could be the fastest algorithm.

Normal – Multiplies the whole matrix with nested for loops, one for each dimension of the array. This algorithm is inefficient because it involves a lot of memory transfers that are solved in the Blocked algorithm.

```
c[i][j] = c[i][j] + a[i][k] * b[k][j].
```

Transposed – The rows and columns of the matrix are switched. The actual multiplication works the same way as for normal, except that b has its rows and columns switched (b[j][k] instead of b[k][j]). Even though this code is almost identical to that for Normal, it is much faster. After looking up some information about transposing matrices on Wikipedia and Stack Overflow, I found that it is possible to write this algorithm with 2 for loops instead of 3, which helped me visualize better why it is so much faster (source: https://stackoverflow.com/questions/19580456/why-does-transposing-matrix-before-multiplication-results-in-great-speed-up)

5. Which algorithm should be the fastest? Is it the fastest? If not, explain why.

Forked Blocked (or MultiBlocked as the file is called) is by far the fastest if you look at the times printed by the program. Even though the printed times for this program are very small, the actual time it took to compute the larger values of n were extremely long (upwards of 20 min when I was the only one on the server). This difference is actual time and printed time is because of the use of fork(). The printed time is how much time each process took, but with fork(), the program might have called a very large number of processes so taking the sum of all of these processes

would give a value close to the actual time. The actual fastest algorithm was the Transposed algorithm which was slightly faster than the Blocked algorithm.

6. Explain how you selected the “ideal” BSIZE. Why is this value the most efficient?

I started with small values of BSIZE (1, 2, 10) and values near the value of n (250, 200, 1000) to get an idea if larger or smaller values might be better. I knew that if the values were too small or greater than n, the program would be less efficient. After getting the edge values, I saw values closer to n were faster than values close to 0, so I tested multiples of 5 and 50 for the values in the middle of the range. When I tested n = 1000, the repeated the same values as for 250, then added 500, 750, and 1000 to my tests. I found it surprising that 750 was the fastest since, proportionally, it was much closer to n than my value for n = 250. As I explained earlier, since BSIZE = 750 doesn’t work for n values smaller than 750 and because BSIZE = 75 was second fastest for n = 1000 and fastest for n = 250, I chose it as my ideal value.

I imagine 75 is an ideal value because it is a balance between the number of blocks that the computer can work with efficiently (low enough to avoid causing thrashing) and the amount of memory the computer can work with efficiently (the computer doesn’t have to move around huge blocks of memory).

7. Explain how Forked Blocked Matrix Multiplication handles collisions. Is this effective? How does this impact the performance of the algorithm?

The below for loop uses the wait function to wait for other processes to finish before continuing. While this ensures that collisions don’t occur, it also makes the program much slower because it causes the forked processes to wait for each other. By making the processes wait for each other, the program becomes closer to being sequential instead of using concurrent processing, but also with the additional memory and processing requirements to fork the program into multiple processes.

```
for(i=0; i<kids; i++)
    {
        wait(NULL);
    }
```