Parallel Strassen's Matrix Multiplication with OpenMPI

15618 Final Project Report

Yunke Cao (yunkec) Yuhao Lei (yuhaol)

Summary

We implemented a parallel Strassen-Winograd's matrix multiplication algorithm with OpenMPI on the latedays cluster.

Background

~~• What are the key data structures?~~

~~• What are the key operations on these data structures?~~

~~• What are the algorithm's inputs and outputs?~~

The Matrix Multiplication algorithm takes two matrix files as input and generate a matrix file that is the result of the multiplication of two input matrices as the output file. We wrote a python script to generate input matrices of different sizes and the correct results for verification.

Strassen-Winograd's matrix multiplication algorithm is a variation of Strassen's algorithm. It performs a 2 x 2 matrix multiplication using 7 multiplications and 15 additions. It can be recursively applied to in a matrix multiplication. The version of Strassen-Winograd's matrix multiplication algorithm is, for matrix $A \cdot B = C$:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

$T_0 = A_{11}, T_1 = A_{12}, T_2 = A_{21} + A_{22}, T_3 = T_2 - A_{11}, T_4 = A_{11} - A_{21}, T_5 = A_{12} - T_3, T_6 = A_{22}$

$S_0 = B_{11}, S_1 = B_{21}, S_2 = B_{21} - B_{11}, S_3 = B_{22} - S_2, S_4 = B_{22} - B_{12}, S_5 = B_{22}, S_6 = S_3 - B_{21}$

$Q_0 = T_0 \cdot S_0, Q_1 = T_1 \cdot S_1, Q_2 = T_2 \cdot S_2, Q_3 = T_3 \cdot S_3, Q_4 = T_4 \cdot S_4, Q_5 = T_5 \cdot S_5, Q_6 = T_6 \cdot S_6$

$U_1 = Q_0 + Q_3, U_2 = U_1 + Q_4, U_3 = U_1 + Q_2$

$C_{11} = Q_0 + Q_1, C_{12} = U_3 + Q_5, C_{21} = U_2 - Q_6, C_{22} = U_2 + Q_2$

The operations on matrices includes multiplication, addition, subtraction and memory layout redistributing as we will discuss further in the following sections. The multiplications (calculation of Qi), can be recursively performed by calling another step of strassen's multiplication algorithm. In each step, the size of the matrix to multiply decreases, e.g. if the size of matrix of A, B, C is n x n, the size of T, S, Q … is n/2 x n/2. Each of the matrix multiplications to get a Qi is a subproblem. When we perform enough steps of BFS so that each processor can get a subproblem, it perform a local Strassen's matrix multiplication on each processor.

In order to optimize the performance of our program, for each matrix, we've used one single continuous memory address to represent it. This could help increase the memory hit rate as well as decrease the complexity for doing matrix operations. The details would be further discussed in the approach section.

• What is the part that computationally expensive and could benefit from parallelization?

• Break down the workload. Where are the dependencies in the program? How much parallelism is there? Is it data-parallel? Where is the locality? Is it amenable to SIMD execution?

There are two types of workloads that can be parallelized. The first type is additions (including subtractions), the second one is multiplication (the calculation of Qi).

The multiplication is the most expensive and obvious workload to parallelize. Ideally, we want to assign a subtask of multiplication to each of the processor. For example, if we have 7 processors, we should call the recursive algorithm once, each of the processor will then calculate each Qi locally. If we have 49 processors, we should the the recursive algorithm twice, the first recursive call will generate 7 subproblems, the second call will further divide each subproblem in to 7 smaller subproblems. Thus we will have 49 small multiplications for each of the processors. This approach to traverse the subproblems is called BFS in [1]. As you may imagine, DFS is another approach. As discussed in [1], when memory is the limit, a few DFS steps need to be taken before BFS. The reason is that, for each BFS step, we need to allocate memory for each of the T, S and Q matrices. This can exceed the amount of memory of the processor. A DFS step will let all the processors dive into one of the subproblem (multiple one of Ti*Si). In each step, the problem size decreases by 2. When the problem size is small enough, we take the BFS steps to assign workloads to each of the processors. This avoids allocation 7 of each the upper levels T and S matrices. More information of BFS and DFS will be discussed in the Approach section.

The addition workloads are also parallelized among processors. The way of doing that is to let each of the processor “owns” a part of elements in the matrices. A processor owns the same part of the matrix in its four “quarters” (A11, A12, A21 and A22). So that it can calculate that part of T, or S matrices by performing additions locally. For example, $T_2 = A_{21} + A_{22}$. In order to perform multiplication, we need to merge the subsets of T or S matrices from each of the processors into the whole matrix. We also need to send the subset of multiplication result matrix back to each of the processors. This is the main source of communication. We adopted the block-cyclic layout given in [1] as our memory layout for matrices to make satisfy these needs. More information of block-cyclic layout will also be discussed in the Approach section.

In terms of dependency, the calculation of Qi depends on the result of Ti and Si. And the calculation of the result matrix C depends on Q matrices. The addition (and subtraction) workloads can be considered data parallel, where the multiplication workload requires a reshuffle of data and communication among all processors. Most of the matrix addition, subtraction workloads are amenable to SIMD. But we will not be optimizing our program from that perspective. The reasons are: first, the addition workloads is minor comparing the multiplications and communication costs (as we will show in the result); second, we are focusing on the MPI model and the assignment of workloads among the processors.

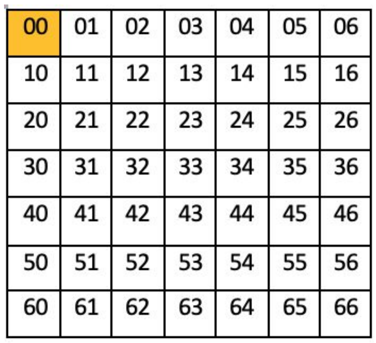
Approach

• Describe the technologies used. What language/APIs? What machines did you target?

We used C programming language and OpenMPI to implement our program. Our program is running on CPU and does target the cluster machines that support message passing method to communicate. Our program attempts to make full use of these cluster machines to accelerate the calculation of the matrix multiplication.

• Describe how you mapped the problem to your target parallel machine(s). IMPORTANT: How do the data structures and operations you described in part 2 map to machine concepts like cores and threads. (or warps, thread blocks, gangs, etc.)

Given P processors and two n x n matrices A, B, our program would evenly distribute the elements of these two matrices into P processors. To achieve high performance, we used the block-cyclic distribution layout [2]. Thus, let's take a deep insight into our layout first.



00	01	02	03	04	05	06
10	11	12	13	14	15	16
20	21	22	23	24	25	26
30	31	32	33	34	35	36
40	41	42	43	44	45	46
50	51	52	53	54	55	56
60	61	62	63	64	65	66

Figure 2.1 Grid layout

Figure 2.1 shows the layout of a grid. In our project, ***grid*** is defined as a submatrix of a complete matrix. In figure 2.1, there're 49 blocks assigned to 49 processors marked with corresponding numbered base 7. The upper left corner orange block is assigned to the processor #0 and the lower right corner to processor #48. For each block assigned to each processor, though it is a submatrix of the grid, we adopt a continuous array to represent it. This layout reduces communication among processors and enables efficient local matrix addition and reduction.

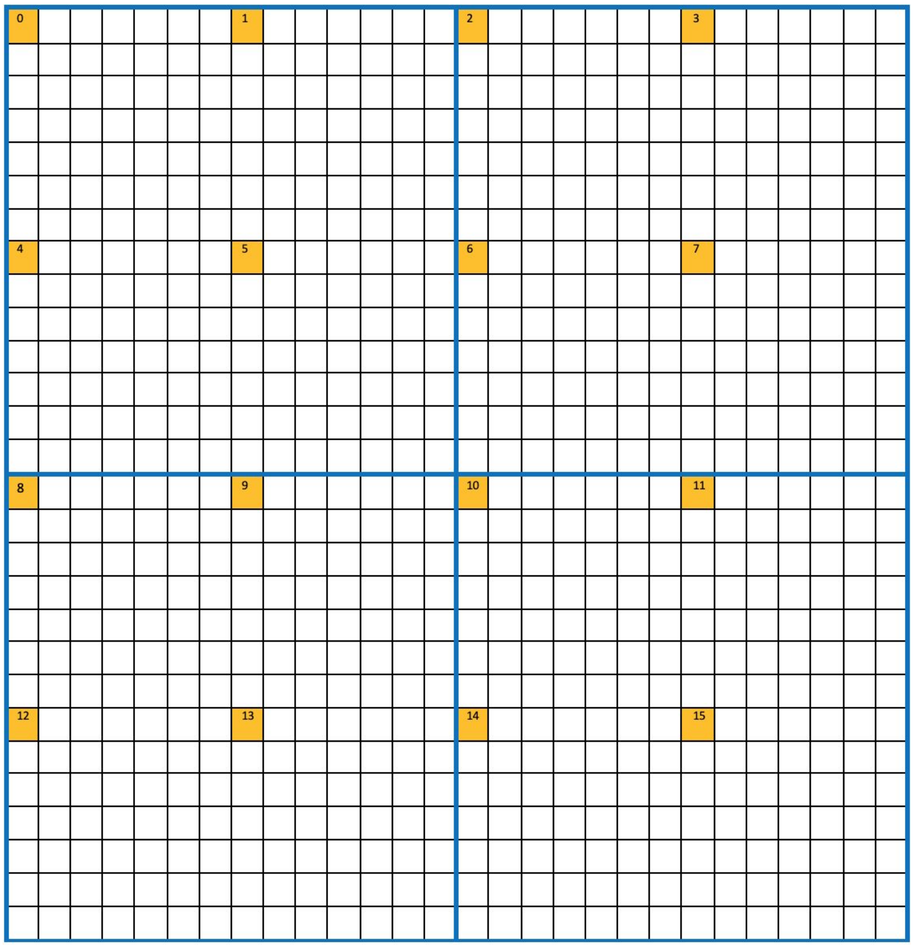


Figure 2.2 Complete matrix layout

Figure 2.2 shows a complete matrix which is divided into 16 grids. Take processor #0 as an example, as we use a single array to store all assigned blocks of one complete matrix on each processor, each upper left corner block of a grid layout is marked with a number from 0 to 15 in figure 2.2. This is to indicate the block's order in the memory and it's very intuitive as it is almost the same as the original order of the complete matrix except that each block should occupy a continuous memory within that single array.

Another thing figure 2.2 shows is that, if we're going to use the previous mentioned Strassen-Winograd's matrix multiplication, we should divided the matrix into 4 submatries which are indicated by the deep bold blue lines in figure 2.2. Take matrix A mentioned in Background section as an example, it should be divided into A11, A12, A21, and A22. Then, the issues comes when calculating $A_{11} - A_{21}$. As figure 2.2 shows, the memory address of block #4 is not following block #1, so does the block #12 with block #9. This issue complicates the calculation of the start memory address of the block and leads to a poor memory hit. Thus, we implemented a feature to transform the order of blocks to guarantee the blocks within a submatrices are continuous in each processor.

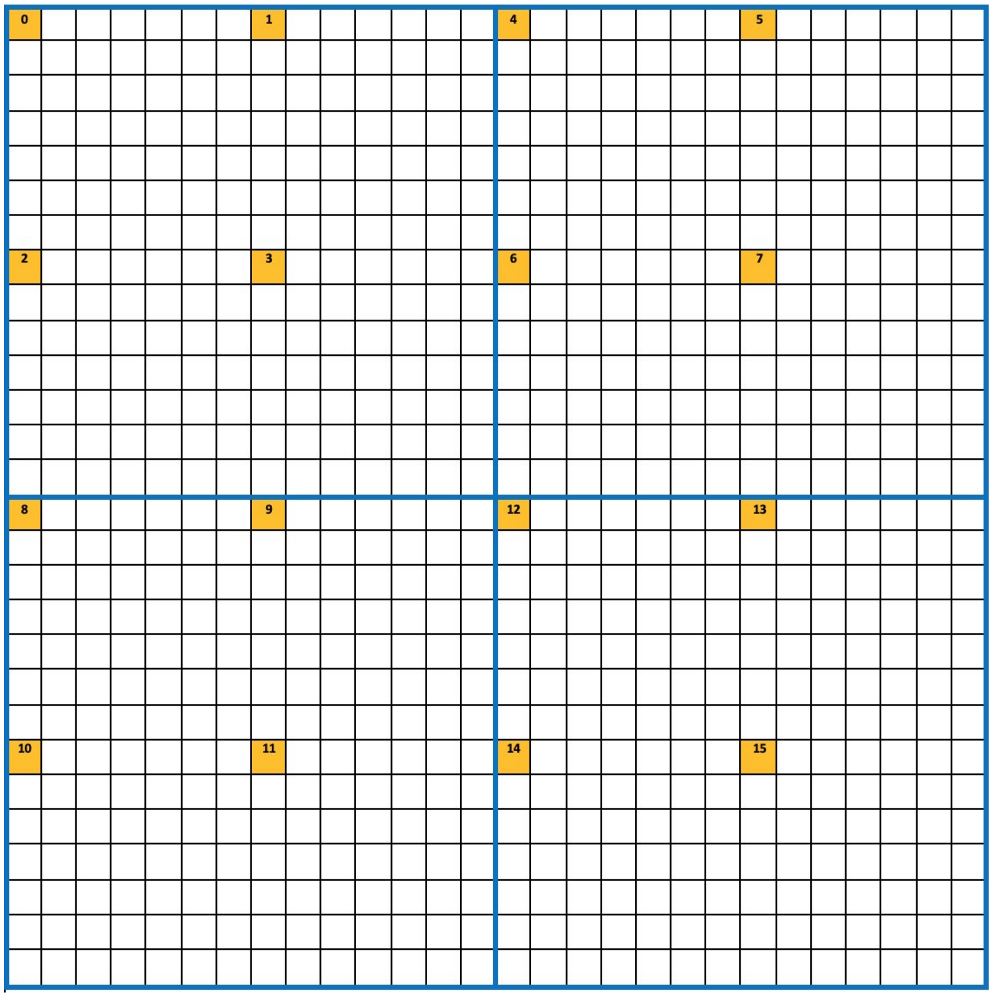


Figure 2.3 Optimized complete matrix layout

Figure 2.3 shows our final complete matrix layout. Taking matrix A as an example, this layout guarantees that each submatrix, A11, A12, A21, and A22, of A would occupy a continuous memory within the whole single array that stores all blocks assigned to each processor.

Then, let's talk about the calculation within each machine and communication among machines. There are four phases, DFS phase, BFS phase, local matrix multiplication phase, and the redistribution of results to each processor phase.

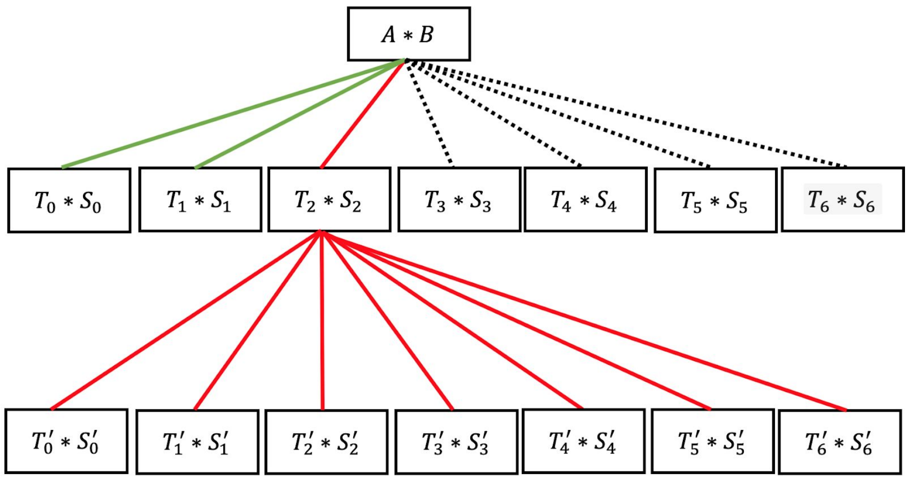


Figure 2.4 Matrix multiplication process

Figure 2.4 shows a matrix multiplication process consists of one DFS phase and one BFS phase. The first branch layer is a DFS phase and the second branch layer is a BFS phase. The green line indicates the processes that have already finished, the red line indicates the processes that are currently running in parallel, and the black dot line indicates the processes to be done in the future.

The DFS phase is used to reduce the memory usage when there is a limited memory. In this phase, each processor only needs to calculate its own local T_i and S_i using the data assigned to it. There's no communication among processors and each processor would recursively call the DFS phase until each T_i and S_i is small enough to fit into the available memory. There's one thing to emphasize that each T_i and S_i is only one fourth of the corresponding matrices A and B.

The BFS phase is used to distribute the calculation of matrix multiplication among different sets of processors. Assume there are P processors, for each BFS phase, there would be 7 submatrices produced, just the same as T_i' and S_i' in figure 2.4. For each of these 7 pairs of T_i' and S_i', there are P/7 processors designated to calculate their product Q_i'. The BFS would also recursively call the BFS phase until there's only one processor with each submatrix. At that time, the local matrix multiplication phase would be triggered to do the only matrix multiplication in our program.

In each BFS phase, communication among processors is needed. Let's go back and take a look at figure 2.1. As there are 49 processors and only 7 processors are needed to calculate the following $T_i' * S_i'$, we need to redistribute the T_i' and S_i' among processors. First, each of the 49 processors has a part of the T_i' and S_i', $i \; in \; [0, 6]$. Thus, in our implementation, we assigns each processor numbered with xi (base of 7) to work together to handle $T_i' * S_i'$. For instance, In figure 2.1, processor #00, #10, #20, #30, #40, #50, and #60 are assigned to calculate the $T_0' * S_0'$, processor #01, #11, #21, #31, #41, #51, and #61 are assigned $T_1' * S_1'$and so on. As for communication, the tricky thing here is that we only need to have each group of 7 processors to communicate within this group. That is, #00, #01, #02, #03, #04, #05, and #06 just send the corresponding local T_i' and S_i' to each #0i. When each processor #xi receives each partial T_i' block from other #xj, it should reassembly each block to form a larger aggregated block.

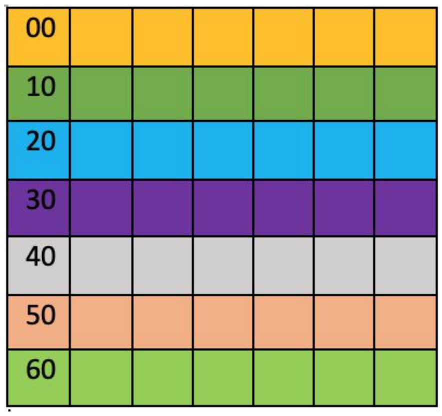


Figure 2.5 Grid layout of processors #x0 after one BFS communication

As figure 2.5 shows, after the communication among processors, the combination of processors numbered #x0 should have a the complete matrices T_0' and S_0'. In the next BFS phase, these 7 processors would communicate with each other so that finally each processor only has two complete matrices T_j'' and S_j'', which is used in the following local multiplication phase.

When each processor has a completed matrix on hand, the local multiplication phase would be triggered and our optimized local serial Strassen matrix multiplication method would be called.

Finally, our program would redistribute the results of each block to its original processor. That is, it would divide its current result matrix into 7 groups of blocks and send each of these groups back to its original owner. Then, each processor can use these blocks to calculate the final result of $T_i * S_i$. This phase actually consists of the tail of BFS phase and DFS phase. In each tail of BFS phase, the result would be send back to its original owner to calculate the result of local $T_i * S_i$. While the tail of DFS phase would calculate the result of local $T_i * S_i$ only, as there's no communication with other processors in the previous DFS phase.

• Did you change the original serial algorithm to enable better mapping to a parallel machine?

Yes, we did. We adopted the block-cyclic layout and redistribute each matrix to enable better mapping to a parallel machine. The details are described in the prior question. Besides, you can check the serialize.c and caps_bfs.c to under the parallel directory to check the difference among our serial version and parallel version.

• If your project involved many iterations of optimization, please describe this process as well. What did you try that did not work? How did you arrive at your solution? The notes you've been writing throughout your project should be helpful here. Convince us you worked hard to arrive at a good solution.

Yes we had. These iterations of optimization are listed below.

1. The appendix in the paper had typo in the variation of the Strassen's algorithm. Later we found a corrected version in the same paper published on another website.
2. In our initial design, we thought a common layout of matrix should be enough but latter Yuhao found that would be a disaster for developing and performance. Thus, we turned to block-cyclic layout to achieve our project.
3. We optimized our layout from figure 2.2 to figure 2.3. This optimization simplified the following development and speedup the matrix addition and subtraction operations.
4. Yunke optimized the memory allocation and deallocation progress, which leads to less memory usage.
5. We tried to use the SIMD based on Intel AVX2 to speedup the matrix addition and subtraction progress. However, we met segmentation fault issue because the memory addresses are not aligned. We tried to fix this issue but later found out that it was due to our memory layout approach. If we still want to leverage the SIMD, we must overthrow our layout design and redesign another one. After a precise analysis, we found that the addition and subtraction operations only occupies less than 1% of the whole computation time. Thus, we discarded this optimization.

6. In the previous version, we used the simplest traditional multiplication to implement the local matrix multiplication progress. Later, we took our serialized version of Strassen algorithm to our parallel version to replace the local matrix multiplication progress and it turned out to be much better.

Results

• If your project was optimizing an algorithm, please define how you measured performance. Is it wall-clock time? Speedup? An application specific rate? (e.g., moves per second, images/sec)

We measured performance in terms of speedup of computation time (does not contain the time to load input matrices and store output matrices) with different number of nodes/processors and varias of input matrix size.

• Please also describe your experimental setup. What were the size of the inputs? How were requests generated?

We experimented on the latedays cluster with OpenMPI. We used number of nodes, processors per node combination of (1, 1), (1, 7), (7, 1) and (7, 7). 49 is the largest number of processors we can experiment on because it doesn't have 7x7x7=343 processors in total (we don't use the hyperthread because our program is computation-intense). The input matrix sizes we experimented on are 196x196, 392x392, 784x784, 1568x1568, 3136x3136, 6272x6272, 12544x12544. 12544x12544 is the largest problem size we can experiment on because of the latedays quota. All elements in the matrices are of data type double. We wrote a python script (test_data/generateMatrix.py) to generate random matrices. For some large matrices that are too large to generate from our laptop, we ran the script on AWS to generate it. We verified results by using "diff" to compare the result generated from our code to the gold standard output matrix from the python script.

• Provide graphs of speedup or execute time.

• The importance of problem size.

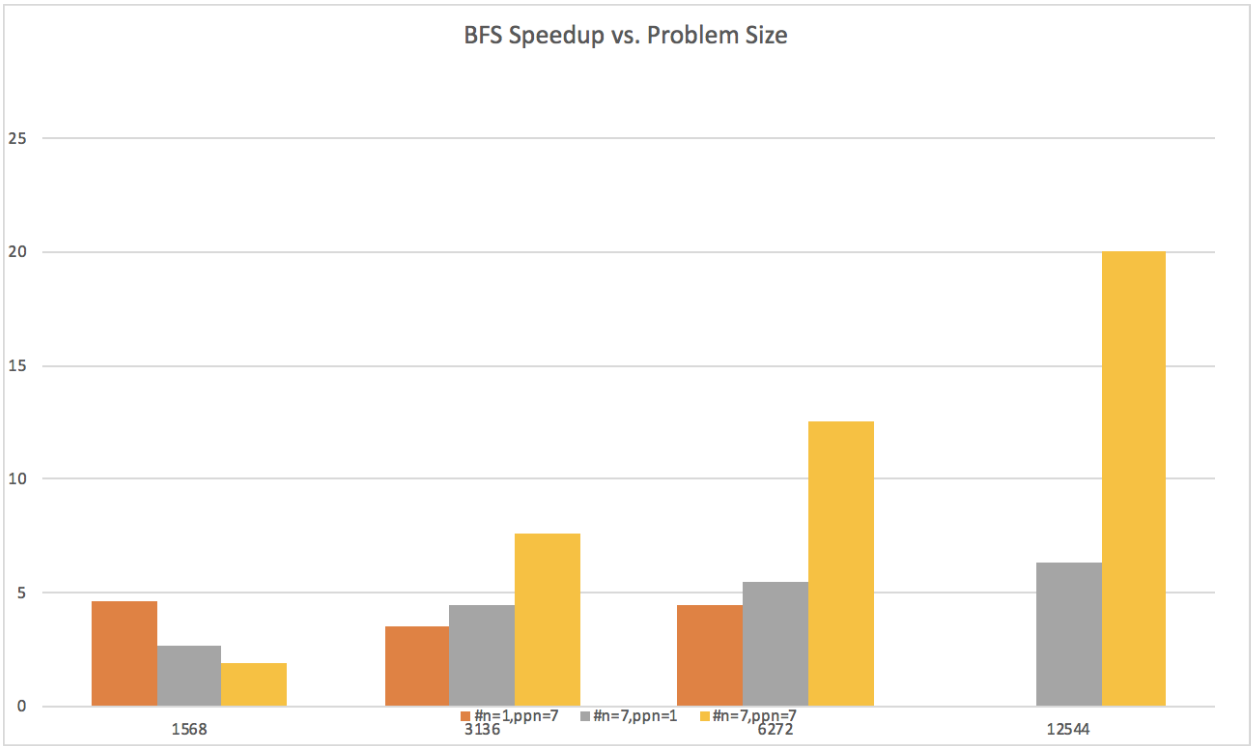


Figure 3.1 BFS Speedup vs. Problem Size

Figure 3.1 shows the speedup of BFS implementation of our program (dfs_step = 0) compared to the serialized Strassen's multiplication algorithm (equivalent to run our program on 1 processor). Overall, our program achieves satisfying speedup, especially on large problem size. On problem size 6272x6272, running on 7 processors achieved 5.48x speedup (7 nodes, 1 ppn) and 4.44x speedup (1 nodes, 7 ppn), running on 49 processors achieved 12.6x speedup (7 nodes, 7 ppn). On problem size 12544, it achieved 6.33x speedup on 7 nodes, 1 ppn, 20x speedup with 49 processors. In this problem size, the 1 nodes, 7 ppn setup will encounter the out of memory problem on the Latedays cluster and be killed by the system. We use a few DFS steps to reduce the use of memory to solve the problem (details below). When the problem size is small (smaller than 1568x1568), we observed that using 7 nodes downgrades the performance. We believe it is because that the communication overhead dominates the total computation time when the problem size is small. We'll also discuss in the following sessions.

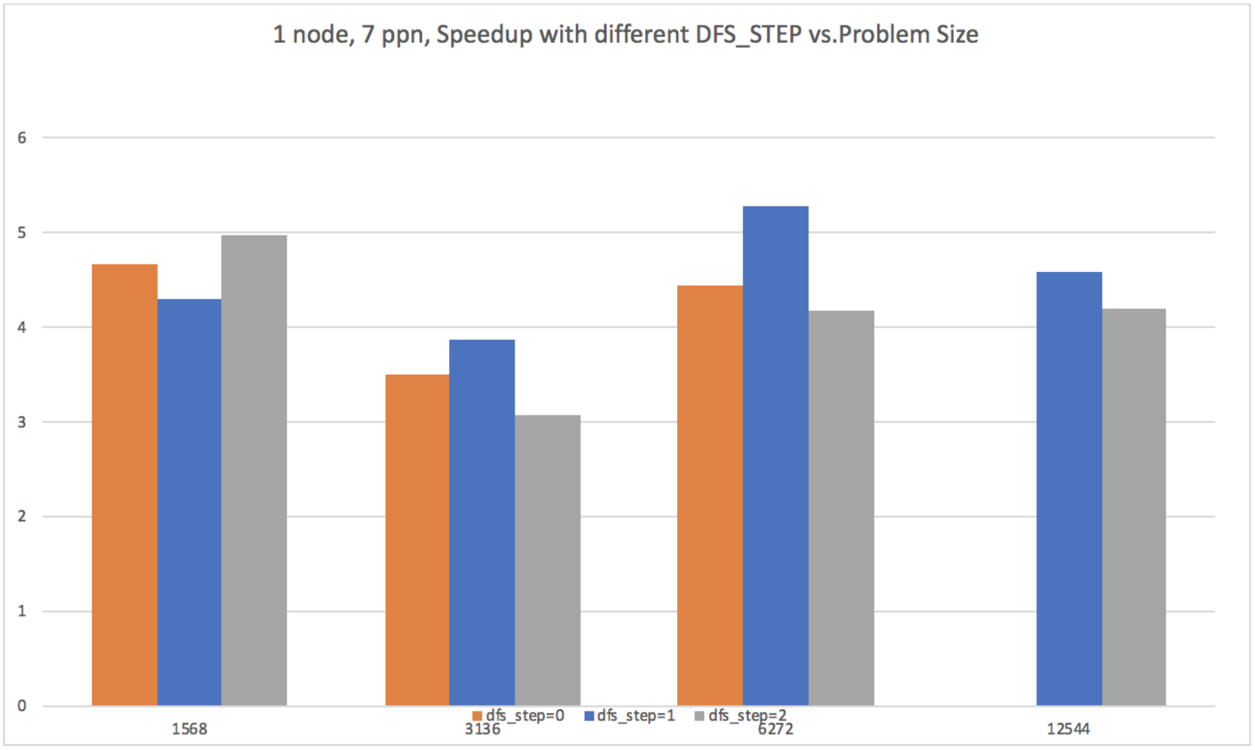
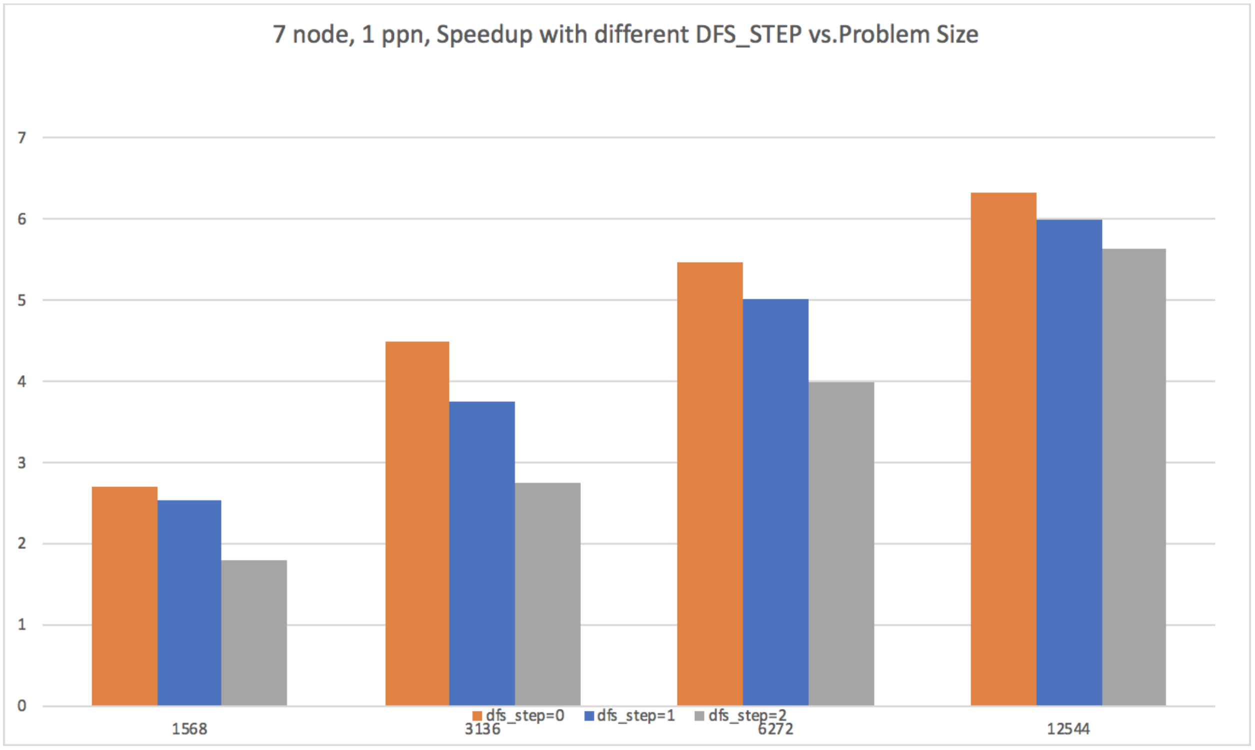


Figure 3.2 Speedup with different DFS_STEP vs. Problem Size (1 node, 7 ppn)

Figure 3.2 shows the speedup of our program running on 7 processors (1 node, 7 ppn), with DFS_STEP set to 0, 1, and 2. We verified that DFS can alleviate the memory limit issue. When the problem size is 12544x12544, the BFS solution encountered OOM problem, where setting DFS_STEP=1 or 2 did not. Due to the Latedays quota limit, this is the largest problem size (input matrix is 1.4GB) we can test. Although a DFS step reduces the use of memory, it adds additional communication overhead. The performance impact when running on 1 nodes is not significant. When we run on 7 nodes, as the communication performance has larger weight, a noticeable performance downgrade happens (Figure 3.3). Therefore, DFS steps should only be used when the memory limit is cannot hold a BFS solution. For the following performance analysis, we will focus on the performance of BFS implementation.

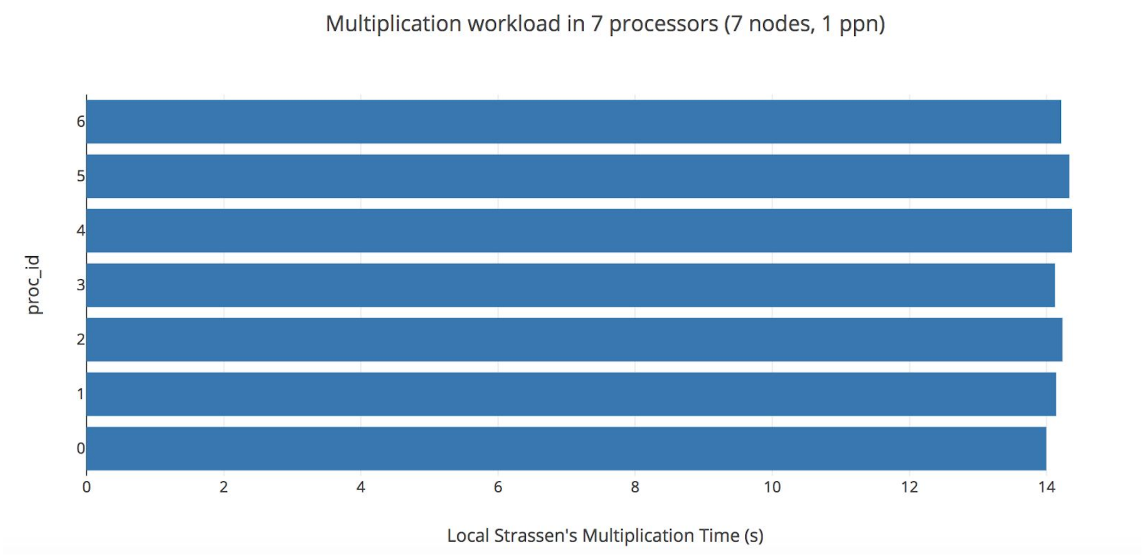


.**Figure 3.3** Speedup with different DFS_STEP vs. Problem Size (7 node, 1 ppn)

• What limited your speedup?

• Can you break execution time of your algorithm into a number of distinct components. What percentage of time is spent in each region? Where is there room to improve?

Our approach guaranteed a sufficient level of parallelism and good load balance as each processor will perform the multiplication locally of equal sized matrices.



.**Figure 3.4** Multiplication workload in 7 processors (7 node, 1 ppn) with problem size 6272x6272

The limiting factor is the communication overhead, especially when the problem size is small and the number of nodes is large. To verify our assumption, we measured the the time used for each processor to:

1. Multiply matrix locally using Strassens' Algorithm
2. Communicate
3. Add/Subtract matrices to produce T, S, C.
4. Other overhead including changing memory layout, allocation memory, copy submatrices, etc.

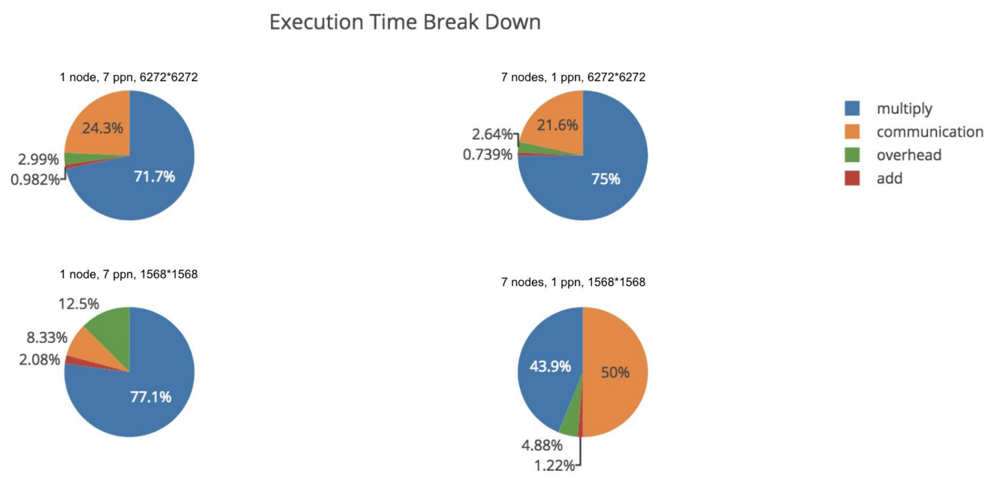


Figure 3.5 Execution Time Break Down

As shown in Figure 3.5, by comparing the breakdown of execution time in these 4 setup, communication dominates when the problem size is smaller and number of nodes is higher. The bottom right graph shows that communication take up 50% of the execution time when the problem size is 1568x1568. From Figure 3.1, this is the setup where using 7 nodes is outperformed by using 1 node. Therefore, the communication model when the problem size is small can be a look for a potential improvement.

We also noticed from Figure 3.1 that the 7 nodes, 1 ppn setup outperforms 1 node, 7 ppn setup (except when problem size is 1568x1568) despite that the communication between nodes is more expensive. We investigated and concluded that the cache size is the cause.

	1 node, 7 ppn	7 nodes, 1 ppn
Cache Miss Rate	9.79%	6.45%

Figure 3.6 Average Cache Miss Rate (per processor), problem size is 6272x6272

According to documentation [3], the processors of one CPU on a node share a 15 MB cache, which cannot hold the working set for 6 subproblems. Therefore, the 7 nodes, 1 ppn setup has a lower cache miss rate and therefore can execute the local multiplication faster.

• Was your choice of machine target sound? (If you chose a GPU, would a CPU have been a better choice? Or vice versa.)

We chose the CPU cluster, and we think it’s a sound choice. GPU could work but when it comes to huge matrix, the limited resources, especially memory resource, within one GPU may not able to handle the work. Things are different if we have a GPU cluster. It might have better performance than our current CPU cluster. This is because GPU has more computation unit to do the simple arithmetic operations, which could augment the speedup the local matrix multiplication phase that occupies a big percent of the whole computation time. However, the communication is still a big issue and our final performance depends on how the GPU hardware is connected with each other within this cluster.

References

[1] Grey Ballard, James Demmel, Olga Holtz, Benjamin Lipshitz, and Oded Schwartz. 2012. Communication-optimal parallel algorithm for strassen's matrix multiplication. In Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures (SPAA '12). ACM, New York, NY, USA, 193-204. DOI: https://doi.org/10.1145/2312005.2312044

[2] The Two-dimensional Block-Cyclic Distribution. https://www.netlib.org/scalapack/slug/node75.html

[3] Xeon e5-2620 v3 processor. https://ark.intel.com/products/83352/Intel-Xeon-Processor-E5-2620-v3-15M-Cache-2_40-GHz

Work Distribution

Yuhao Lei: 50%. Implemented Block Cyclic Layout and BFS.

Yunke Cao: 50%. Implemented serialized version and DFS.

Appendix

Execution Time:

Matrix Size 12544 * 12544

nodes/ppn/dfs_steps	1/1/0	1/7/0	7/1/0	7/7/0	1/7/1	7/1/1	7/7/1	1/7/2	7/1/2	7/7/2
Total Computation:	760	OOM	120	38	166	127	53	181	135	74
Communication:	0		19	20	35	25	35	32	30	54
Local Strassen:	760		99	16	122	98	15	132	97	15
Addition:	0		0.5	0.3	0.5	0.21	0.06	0.44	0.2	0.05

Matrix Size: 6272 * 6272

nodes/ppn/dfs_steps	1/1/0	1/7/0	7/1/0	7/7/0	1/7/1	7/1/1	7/7/1	1/7/2	7/1/2	7/7/2
Total Computation:	103	23.2	18.8	8.2	19.53	20.5	11.0	24.7	25.8	27.3

Communication:	0	5.7	4.1	5.7	1.2	5.8	8.2	6.0	10.7	24.4
Local Strassen:	103	16.8	14.2	2.2	16.7	13.6	2.2	15.4	12.7	1.8
Addition:	0	0.23	0.14	0.09	0.10	0.05	0.02	0.09	0.05	0.02

Matrix Size: 3136 * 3136

nodes/ppn/dfs_steps	1/1/0	1/7/0	7/1/0	7/7/0	1/7/1	7/1/1	7/7/1	1/7/2	7/1/2	7/7/2
Total Computation:	14.0	4.0	3.12	1.83	3.61	3.73	4.55	4.55	5.09	27.3
Communication:	0	1.5	1.07	1.46	0.26	1.57	4.13	1.67	2.86	24.2
Local Strassen:	14.0	2.4	1.93	0.30	2.81	1.84	0.26	1.93	1.54	1.87
Addition:	0	0.05	0.03	0.02	0.03	0.02	0.01	0.03	0.02	0.02

Matrix Size: 1568 * 1568

nodes/ppn/dfs_steps	1/1/0	1/7/0	7/1/0	7/7/0	1/7/1	7/1/1	7/7/1	1/7/2	7/1/2	7/7/2
Total Computation:	2.19	0.47	0.81	1.16	0.51	0.86	2.78	0.44	1.22	15.6
Communication:	0	0.04	0.41	1.09	0.06	0.44	2.71	0.04	0.79	15.56
Local Strassen:	2.19	0.37	0.36	0.04	0.32	0.30	0.03	0.20	0.22	0.03
Addition:	0	0.01	0.01	0.01	0.01	0.004	0.0003	0.006	0.004	0.001