# Compiling and Running the Programs

The submission folder contains 5 sub-folders, 1 pdf file, and 1 env.output file. The table below explains the usage of each sub-folder.

|  |  |
| --- | --- |
| Folders/Files | Comments |
| BlockingMPI | This folder contains all the source code, test results, Makefile, and bash scripts for the Blocking MPI program. |
| NonBlockMPI | This folder contains all the source code, test results, Makefile, and bash scripts for the Non-Blocking MPI program. |
| BalanceLoad | This folder contains all the source code, test results, Makefile, and bash scripts for the Non-Blocking MPI program with balanced load. |
| Serial | This folder contains all raw source code provided by Dr. Sherman from directory, ~ahs3/cpsc424/assignment2 |
| testEntry | This is the top level wrapper folder where user can change the block size and processor number, and submit qsub request. |
| report.pdf | The Write-up of this assignment |
| env.output | System environment output file for my develop system. |

Table 1: Top-tier directory

As stated above, each folder is named after its task, and more details in each task and folder will be provided in the upcoming sections of this report. In this secton, I will first look into the *testEntry* folder, its files, and the usage of its files for compiling and running the entire assignment.

In the *testEntry,* you will simply see two files. The table below shows the usage of each file.

|  |  |
| --- | --- |
| Files | Comments |
| build\_all.sh | This is the script to build all the program, and run the program on qsub by issuing qsub command. |
| variable.h | This is simply an input file for all the programs. It specifies the matrix size and number of processors for the programs. |

Table 2: testEntry folder

As presented in table 2, build\_all.sh is the qsub script for compiling and running the program. By default, the number of processors is set to 8 on one node, calculating matrix size of 1000. Without changing anything, to submit the qub job for the assignment, one can do:

*Bash$ qsub build\_all.sh*

The *build\_all.sh* script will first clean prior compiled executable and object files, and re-make executable file for each program. Once the qsub finishes running, an output file named *result.o123456* will be placed in this folder, and this the *STDOUT* of all three MPI programs required in this assignment. The result file will contain some progress prints for *make* and *qsub*. Further down the result file, for each MPI program, The *STDOUT* will contain very last element of each product matrix for verification purpose along with the calculation time and real time on each process. If the specified matrix size is less than 18, the *STDOUT* also contains the entire product matrix. Details on how to interrupt the *STDOUT* for each program will be presented in the next sections.

The *variable.h* is the file that all three MPI programs reference to get the matrix size and process numbers (MPI\_WORLD\_SIZE). I have placed it in this folder for easy access for Ronghui. The contents of this file is shown in the screen-shot below.



Figure 1: screen shot of variable.h

By default, the MAT\_SIZE is set to 8, and NUM\_PROCESSORS is set to 8. To change the matrix size, one can simply replace the MAT\_SIZE value in figure 1 to any number as one wish. Similarly, one can replace the 8 with any number that is the factor of the matrix size. However, since the number of processors gets changed, there is a couple changes need to be made in the *build\_all.sh,* referring to the screenshot below.









Figure 2: NUM\_PROCESSOR constnat dependent lines in build\_all.sh

As figure 2 shown, user need to change 8s in the lines specified in the screenshot above to same value as NUM\_PROCESSORS. Once done changing the constants, one should save the file and submit the qsub job (*Bash$ qsub build\_all.sh*). The newly generated results will be based on the new MAT\_SIZE and NUM\_PROCESSORS.

Though all three programs can run on any matrix sizes, **the number of processors has to be the factor of the matrix size.** The programs does not have error checking schemes for this requirement, so user needs to be aware of this problem.

# Task 1: Serial Program

The files in the *Serial* folder are copied directly from ~ahs3/cpsc424/assignment2. There is no major changes in the given source files. I only added printf() for the multiplication results in *serial.c*. However, the results may not be the same in my MPI programs since the values of the triangular matrices (A and B) are different between the serial program and my MPI programs due to different ways of initializing the matrices. The serial program initializes matrices, A and B, together to produce different elements in each matrix. My MPI programs initialize A and B separately to produce the same random elements for each matrix. One advantage of my method is that if I print out the entire product matrix, I can verify the product pattern is of the result of two triangular matrices (lower and upper).

I ran the serial program as instructed, the screen-shot below shows the *STDOUT* from qsub.

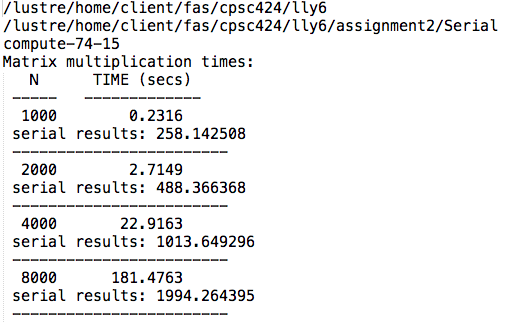


Figure 3: serial output

# Task 2: Blocking MPI

All source files of Blocking MPI are stored in BlockMPI folder. The files and their usages are presented in the table below (table 3).

|  |  |
| --- | --- |
| Files/folder | Comments |
| blockingMPI.c | The main() entry of the blocking program |
| matmul.c | The source file that contains all the matrix related initialization of calculation |
| matmul.h | The head file of matmul.c, this is included in the blockingMPI.c |
| mpiWrapper.c | Some MPI function wrappers for easier and more readable code review |
| mpiWrapper.h | Header file for mpiWrapper.h |
| Makefile | Makefile for *make* (rules for compiling the program) |
| blockMPI | Executable generated by *make* |
| runBlockingMPI.sh | qsub job file for this singular program |
| results/ | The backlog for all the jobs I have ran so far. You are welcome to have a look, but the table presented in this section is much more informative than the backlogs. |
| \*.o | Anything with .o extension is the object file generated by *make* |

Table 3: BlockingMPI directory

As shown in table 3, although build\_all.sh can *make* all programs, each program has its own *Makefile* to do separate compilation.

Table 4 and Chart figure 4 shows the computation time for the blocking program. The computation time is the time span the program spent on matrix multiplication.

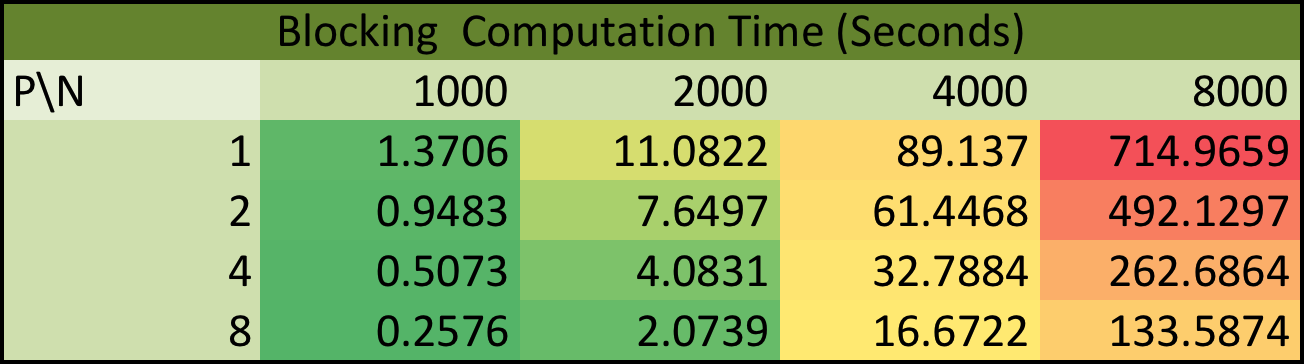


Table 4: Blocking MPI Computation Results

Figure 4, Blocking MPI Computation Chart

In table 4, I have color-coded the values for better comparison. Red means slowest, and green means fastest. In terms of raw performance, as we can see that while the program size (N) is fixed, with the increase of the processes, the computation time gradually declines. From the figure 4, I can also see that although the problem size varies, but each of problem size has the similar decline slope. Since I focus on matrix multiplication alone, and there is not a part of the matrix multiplication that cannot be parallelized, the serial fraction (f) here is virtually 0. Based on equation:



I can see that when f = 0, the parallel efficiency is 1. Therefore, by looking at the computation time alone, I can see that the efficiency is fixed with the increase number of processes with a fixed problem size (N). The program is strongly scalable. However, the table below shows the program run-time (real time mpiexec produces). The real-time take into account of the communication time between different processes in addition to the computation time. Based on the color-coding, I can see that when we keep the N fixed, the improvement is not as gradual as before. This is probably because the communication time is the serial fraction of the problem, and it dominants the total run time. If I follow the E(p) equation, we can clearly see that the E(p) is decreasing when f is fixed, and p increases. In an extreme case where p tends to be infinite, based on the equation:



I can see that the efficiency eventually becomes 0. Therefore, when looking at the program as a whole, the program is weakly scalable.

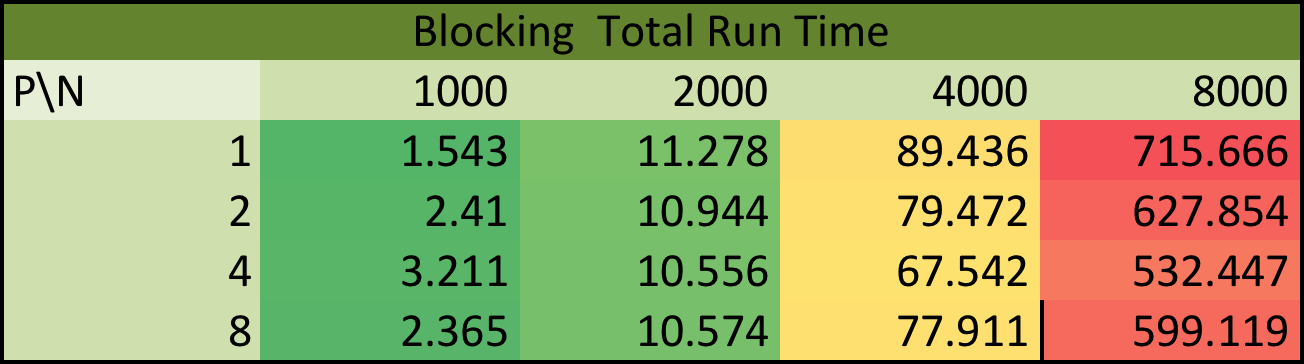


Table 5: Blocking MPI Real Time

I have discussed the problem running on one node. The table 6 below shows how the total run time behaves when it is deployed on multiple nodes.

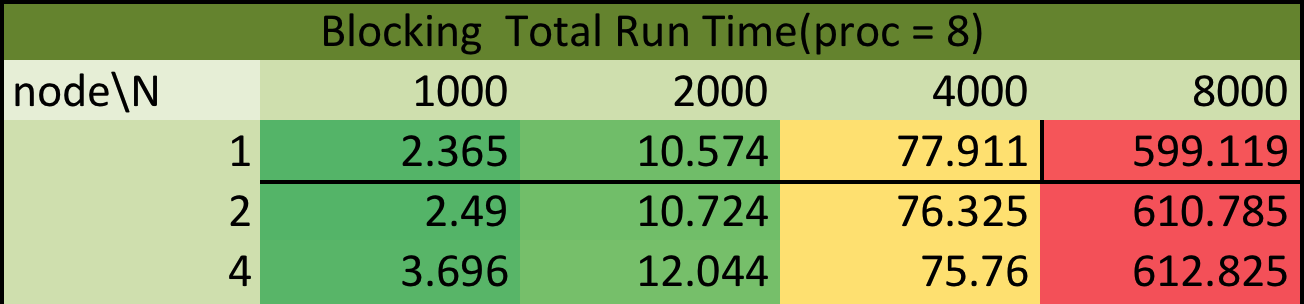


Table 6: Blocking MPI on multiple nodes

Similar to the problem running on one node, the cross-node run on blocking MPI does not change the scalability of the program (weakly scalable). The computation time remains the same when it is deployed on different nodes.

The current implementation has unbalanced load, the multiplication approach assigns low load to higher ranked processes (0 being highest rank), and the lowest ranked processes has the highest load.

One suggestion for the program is to unevenly divide the raw matrix such that the lower ranked processes with high load do not bottleneck the computation time. Another way to improve scalability is to make a non-blocking MPI program so that we can largely reduce the fraction of communication and thus reduce serial fraction.

# Task 3: Non-Blocking MPI

All source files of non-blocking MPI are stored in NonBlockMPI folder. The files and their usages are presented in the table below (table 7).

|  |  |
| --- | --- |
| Files/folder | Comments |
| nonBlockingMPI.c | The main() entry of the non blocking program |
| matmul.c | The source file that contains all the matrix related initialization of calculation |
| matmul.h | The head file of matmul.c, this is included in the nonBlockingMPI.c |
| mpiWrapper.c | Some MPI function wrappers for easier and more readable code review |
| mpiWrapper.h | Header file for mpiWrapper.h |
| Makefile | Makefile for *make* (rules for compiling the program) |
| nonBlockMPI | Executable generated by *make* |
| runNonBlockingMPI.sh | qsub job file for this singular program |
| results/ | The backlog for all the jobs I have ran so far. You are welcome to have a look, but the table presented in this section is much more informative than the backlogs. |
| \*.o | Anything with .o extension is the object file generated by *make* |

Table 7: NonBlockingMPI directory

Table 8 shows the computation time comparison between non-blocking and blocking program. The computation time is the time span the program spent on matrix multiplication.

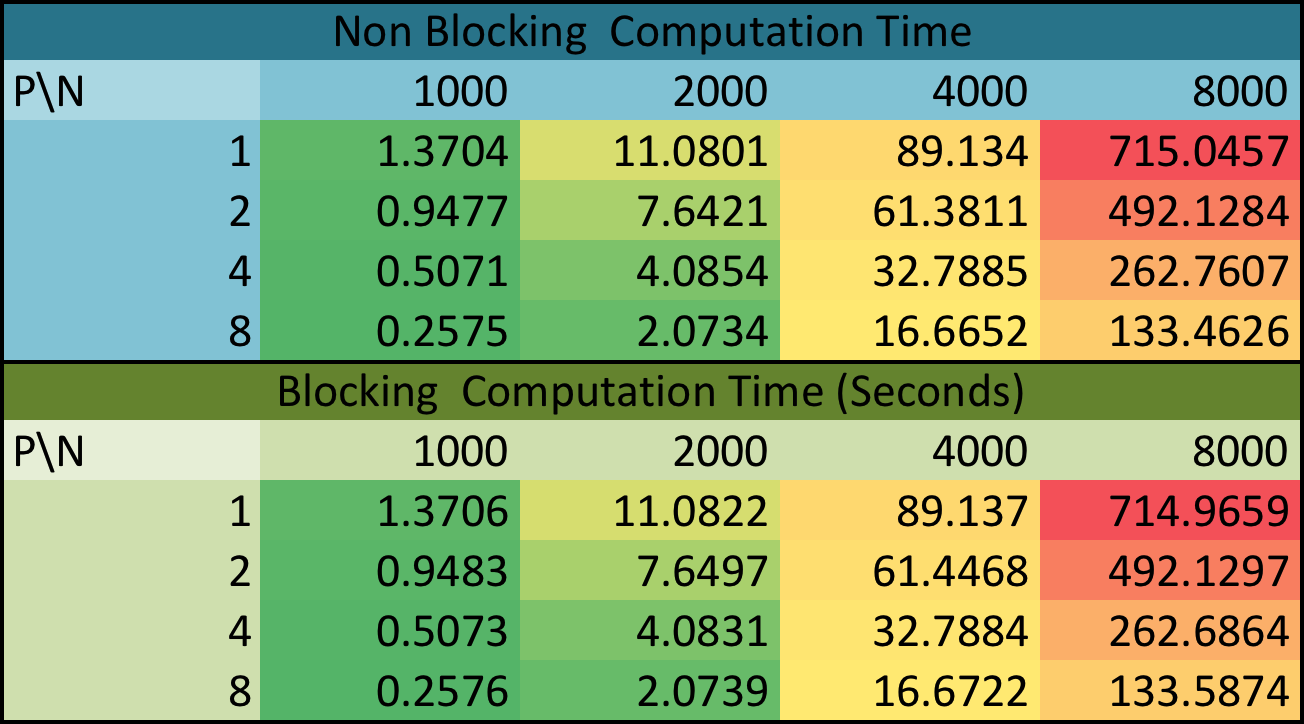


Table 8: Comparison between blocking and Non-blocking Computation Time

As I can see from table 8, the overlapping computation and communication does not necessarily improve the computation performance as the algorithm of computing the matrix remains the same. Therefore, in non-blocking program, the computation alone is strongly scalable.

Table 9 shows the total run time comparison between the two programs.

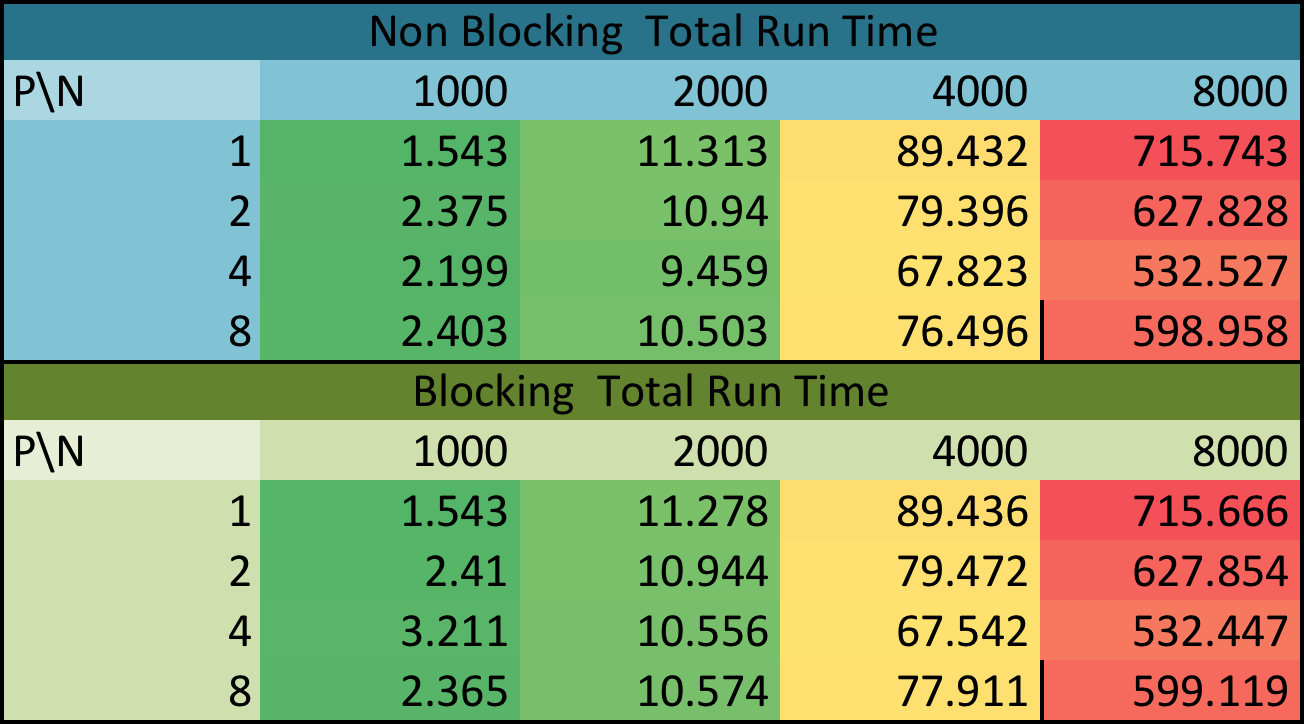


Table 9: Comparison between blocking and Non-blocking total run time

Interestingly, from table 9, I do not see any significant differences between the two in terms of total run time for one-node deployment. I want to move on to the multiple-node deployment and come back to discuss this interesting finding.

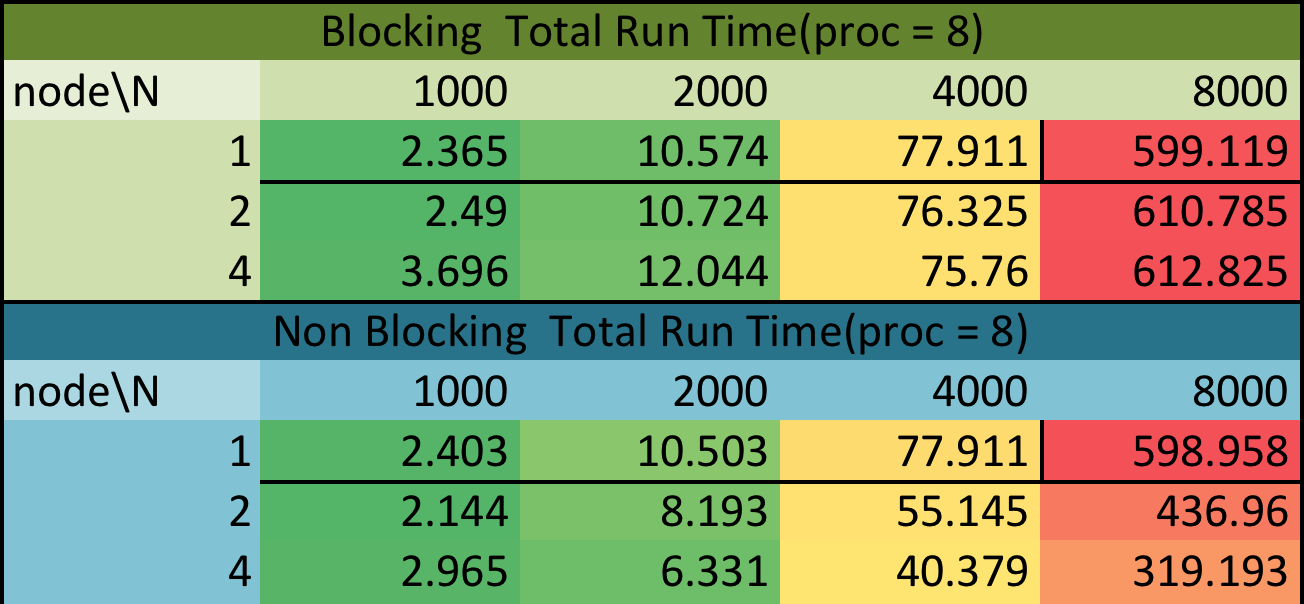


Table 10: Non-blocking vs Blocking Run Time on multiple nodes

Figure 5: non blocking run time on multiple nodes, figure

From both the figure 5 and table 10, I can see that there are significant performance gains when we increase the number of nodes. That being said, overlapping communication with computation does increase scalability and performance on multiple nodes. However, neither the performance nor the scalability saw any visible changes in one-node deployment. One possible reason is that intra-node routing has more overhead than inter-node routing, and there it cannot take full advantage of the overlapping. Another possible reason is that since I am only overlapping the sending with computation, there are still stop and wait time frames on the receiving side.

In conclusion, even in non-blocking program, the communication is still the dominant fraction of the program. However, when it is deployed on multiple nodes, the inter-node communication reduces the dominance of the communication overhead, resulting the performance and scalability gain on multi-node case.

# Task 4: Load Balance

In terms of load balancing case, I do not have enough time to fully explore the load balance implementation. Given that my existing program does see scalability and performance gain on multi-node case, it mildly offsets the unbalanced load.

Another thing I have done in the load balance program is instead of propagating the blocks from master node, each node creates its own chunks for A and B and therefore reduces some of the serial fraction of the program. Since the algorithm for multiplication remains the same, the computation time largely remains the same as task 2 or 3.

One drawback of this implementation is that since each node initialize its own A and B chunks, the multiplication results will be different on randomized elements.

# Extra Credit

The program is built with flexibility in mind. There are two parameters one can find in the variables.h file in testEntry folder. One can try to change both parameters for testing purpose.