Sparse-Matrix and Fat-Vector Multiplication

High Performance Computing Report

M.Sc. Software Engineering for Technical Computing

Aman Makkar
S321839

22nd of February 2021

Abstract

This report has been compiled based on the work carried out using the Message Passing Interface (MPI) library, for a multiplication of a Sparse Matrix and a Fat-vector. The purpose is to implement the multiplication using the C programming language coupled with MPI. Three algorithms have been designed, analysed, implemented and tested on a High-Performance Computer (HPC), alongside an External repository. The purpose is to understand the effect of communication and computation cost in parallel programming and to this end, based on the results gathered via the HPC, the performance of the three algorithms is compared with a sequential program as well as an external repository.

Contents

1. Introduction

The purpose of this report is to examine the application of distributed-memory parallel programming techniques for the implementation of sparse matrix fat-vector multiplication. Three algorithms have been designed and analysed alongside an External repository. The three algorithms are then written in the C programming language coupled with MPI. The reason for using MPI is because it is the only Message Passing Library that is a standard and it can be used across many HPC platforms (*Message Passing Interface (MPI)*, n.d.). It should be noted that MPI itself is not a programming language, it is a library that provides bindings for C, C++ and Fortran. MPI is provided by various companies such as Intel and Microsoft, it is also available via open source in the form of OpenMPI, because MPI is a standard, using any of the providers would mean that the code is still portable and efficient (Pachero, 1997).

2. Problem statement

A sparse matrix and a fat-vector have been provided, distributed parallel-programming techniques must be used for the implementation. Three algorithms need to be devised in the C programming language coupled with MPI, these algorithms then need to be tested on the "Crescent" HPC computer located at Cranfield University. The performance of all the algorithms must be measured, compared and then analysed with the serial program as well as an external repository to determine the computation and communication costs.

$$\begin{pmatrix}
\times & \times & & & \times & & & \times \\
\times & \times & \times & & & & & \\
& & & & & \times & \times & \\
& \times & \times & \times & & & & \\
& & \times & \times & & & & \\
\times & & & & \times & \times & & \times \\
\times & & & & \times & & & \times \\
& & & & \times & \times & \times & \\
& & & & & \times & \times & \\
\times & & & & \times & & & \times
\end{pmatrix} * \begin{pmatrix}
\times & \times & \times \\
\times & \times & \times \\
\times & \times & \times \\
\times & \times & \times \\
\times & \times & \times \\
\times & \times & \times \\
\times & \times & \times \\
\times & \times & \times
\end{pmatrix}$$

where M is a $m \times n$ sparse matrix and v is an $n \times k$ vector.

Fig 1: M Sparse Matrix and V Fat Vector.

3. Parallel Programming with MPI

MPI is used in a system that has distributed memory architecture where processors have their own local memory. Each processor works independently and there is no global address space (Moulitsas, 2020). MPI provides a way of sending and receiving messages explicitly between the processes and these processes are statically allocated at the beginning of program execution (Pachero, 1997). Messages can be sent synchronously or asynchronously between the processes and the design of the program is system driven. For instance, if the system has buffer capability or not and if a blocking or non-blocking communication is required. To be able to determine which of these designs would provide the best performance, i.e. the correct solution in the shortest time with the least or available computation power, some understanding of the available hardware is useful.

There are other parallel programming methods and libraries available, for instance, if the system is shared but can also be private then OpenMP library can be used. If the memory is private but it can share memory then PGAS can be used (Moulitsas, 2020).

3.1 Amdahl's law and Efficiency

In order to be able to measure the performance of parallel programs, Amdahl's law will be used to measure the speed-up and in turn the efficiency of the algorithm. Amdahl's law is as follows (Moulitsas, 2020):

$$S = \frac{T_s}{T_p}$$

Where S is the speed-up, Ts is Execution time using single system and Tp is execution time using a multiprocessor system with P processors.

In order to measure the efficiency of the algorithm, the following formula is provided (Moulitsas, 2020):

$$E = \frac{S}{P}$$

Where E is the efficiency, S is the speed-up and P is number of processors.

4. Methodology

For the purposes of this work, three separate algorithms will be designed, analysed and tested on Crescent. Alongside the three designed algorithms, an external library will also be compared and tested. The three possible methods will be algorithms, where one algorithm will use point-to-point blocking communication, the second algorithm will use point-to-point non-blocking communication and the third algorithm will use collective communication. It should be noted that the external repository has been chosen based on the code that could be comparable to the three algorithms that are designed and presented in this report.

4.1 Algorithm one

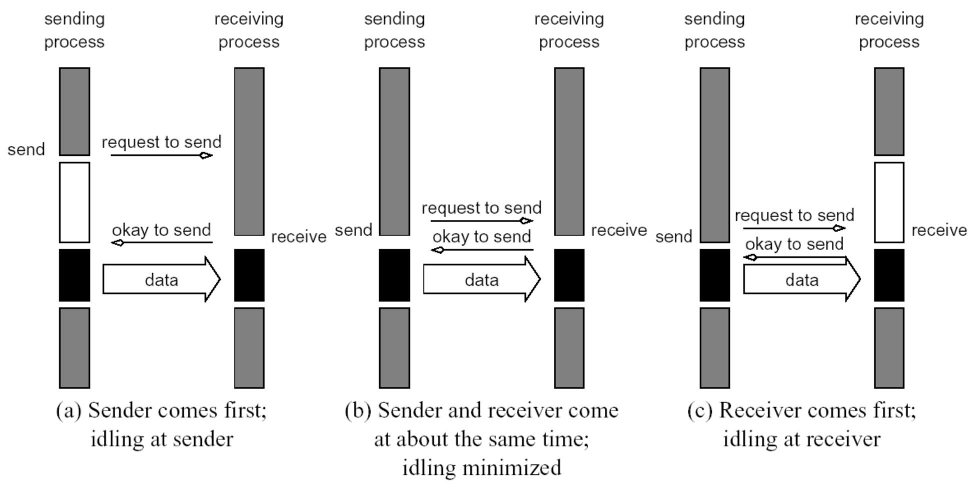
sending process
receiving process
send
request to send
okay to send
data
receive
(a) Sender comes first; idling at sender
(b) Sender and receiver come at about the same time; idling minimized
(c) Receiver comes first; idling at receiver

Figure 6.1 Handshake for a blocking non-buffered send/receive operation. It is easy to see that in cases where sender and receiver do not reach communication point at similar times, there can be considerable idling overheads.

Fig 2: Point to point blocking communication

Figure two above shows three scenarios, in scenario “a” the sending process requests the receiving process to start sending and until the receiving process send an “okay to send” message to the sending process, the sending process remains idle. In scenario “b” which would be the most ideal condition for blocking communication in non-buffered hardware, is that the send and receive processes come at the same time, reducing or eliminating the idle time, improving the performance. In scenario “c”, if the receiving process is ready to receive however, if the send process is not yet ready to send, the receive process will remain idle until the send process sends a request to send (*6. Programming Using the Message-Passing Paradigm - Introduction to Parallel Computing, Second Edition*, n.d.).

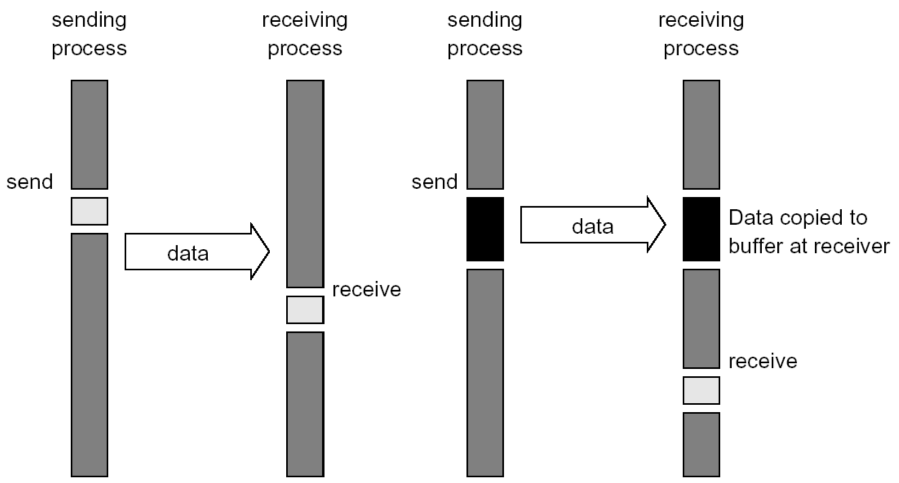

Figure 6.2 Blocking buffered transfer protocols: (a) in the presence of communication hardware with buffers at send and receive ends; and (b) in the absence of communication hardware, sender interrupts receiver and deposits data in buffer at receiver end.

Fig 3: Point to point blocking buffered communication

Figure three above shows two scenarios, scenario "a" has communication hardware and buffers at both send and receive ends, which means both processes can carry on working and only carry out the send and receive action when they are ready. In scenario "b", there is no communication hardware although there is buffer, when the sending process is ready to send data it would interrupt the work that receiving process is doing and deposit the data in the buffer.

The first algorithm will use the point-to-point blocking communication method, this means that when one of the processes calls for a message receive and if that receive message is not yet available, the calling process will remain idle until the message has been received (Pachero, 1997). For the purposes of this design, process 0 will send the matrix and vector to all the worker processes and these processes will remain idle until process 0 completes the send messages. The worker processes will then receive the matrix and vector, carry out the multiplication and save the product in a resultant vector, which is then sent as a message to process 0. Process 0 will remain idle until the worker processes complete the send message, the message is then received and printed by process 0. This design is expected to have high idling overheads as the processors numbers increase due to the amount of time all processes will be idle and the amount of waiting process 0 will have to do while waiting for worker processes to complete sending, the idling time for the processes would also increase (Wilkinson & Allen, 2005).

4.2 Algorithm 2

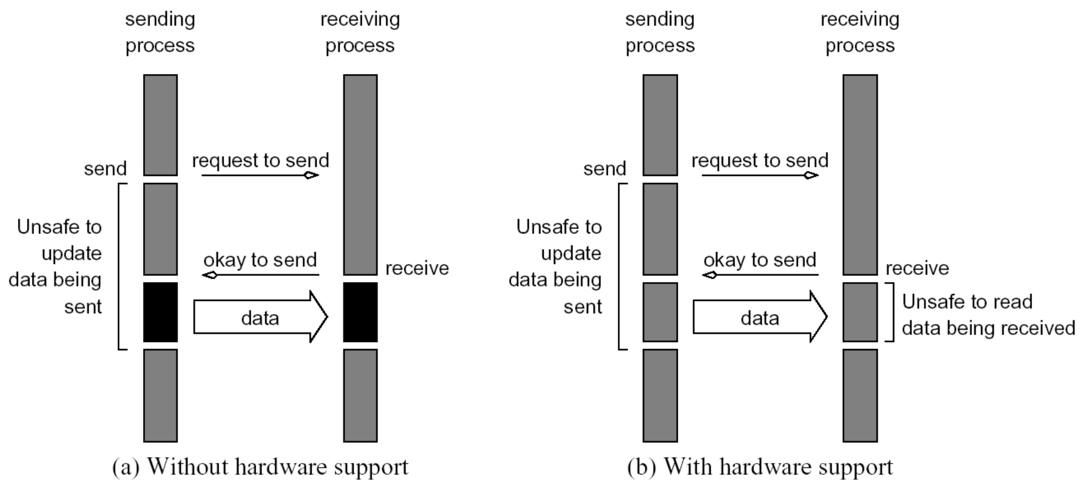
sending process
receiving process
send
request to send
Unsafe to update data being sent
okay to send
receive
data
(a) Without hardware support
sending process
receiving process
send
request to send
Unsafe to update data being sent
okay to send
receive
data
Unsafe to read data being received
(b) With hardware support

Figure 6.4 Non-blocking non-buffered send and receive operations (a) in absence of communication hardware; (b) in presence of communication hardware.

Fig 4: Non-Blocking Non-Buffered Communication

Figure four above shows two scenarios, scenario “a” shows how non-blocking communication works if there is no communication hardware software. The sending process sends a request to the receiving process however, it carries on doing its own work without waiting for the receive process to send a “okay to send” message. However, once the request to send has been sent by the sending process, the data that is to be sent must not be updated. Once the receiving process is ready to receive, the sending process will stop the work it is doing and send the data to the receiving process and the receiving process will also stop to receive the data. In scenario “b”, there is communication hardware provided which means the sending process will send a request to send data and will carry on doing its own work. Neither process will stop their own work while the data is being sent or received however, it is unsafe to read the data in the receiving process while the data is being received and it is unsafe to change the data while it is being sent (*Chapter 3. Distributed-Memory Programming with MPI - An Introduction to Parallel Programming*, n.d.).

The second algorithm will use point-to-point non-blocking communication method, this could have large improvements on the performance of the program. For example, if a node has the capability of carrying out computation work simultaneously with the communication, the computation that does not depend on any other communication (Pachero, 1997). For the purposes of this design, process 0 will send messages to all the worker nodes however, as soon as a process has received the information it needs to carry out its work, it will start that work rather than waiting for all the information to complete the sending process. MPI_Isend and MPI_Irecv functions will be used for this design, these functions are specific for non-blocking communication. As this algorithm will not have high idling times, it is expected to perform better than algorithm 1 (*6. Programming Using the Message-Passing Paradigm - Introduction to Parallel Computing, Second Edition*, n.d.).

4.4 Algorithm three

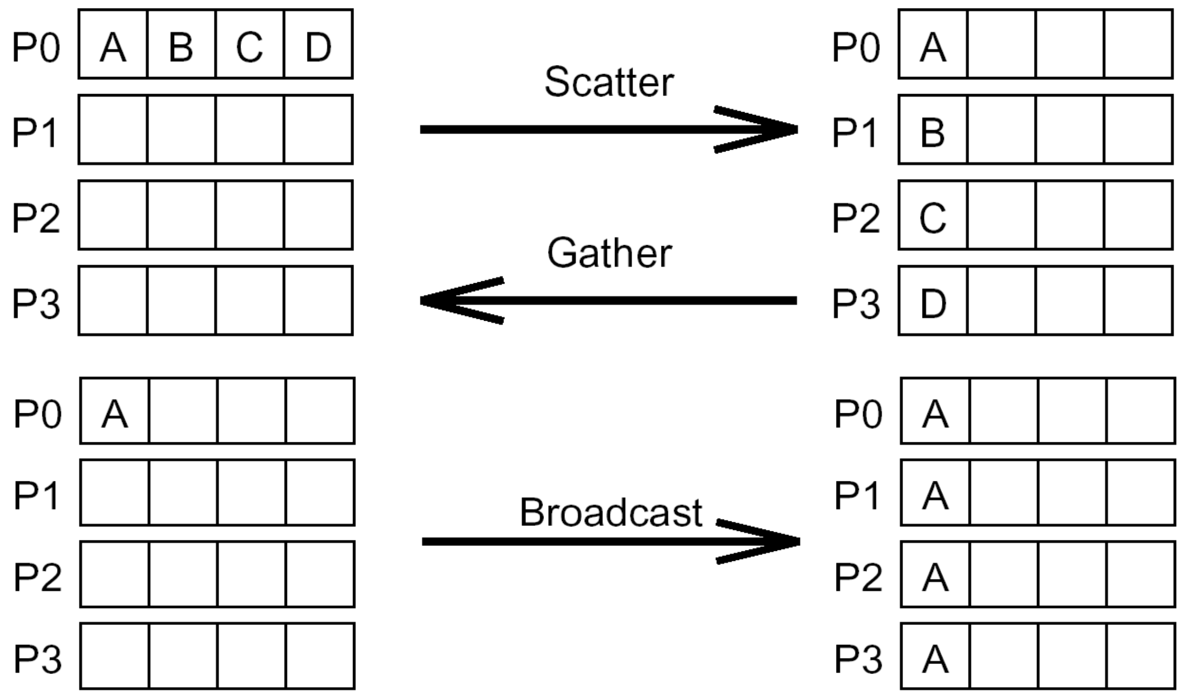

Fig 5: Scatter, Gather and Broadcast for Collective Communications

Fig 5 above shows three separate functions in MPI. The MPI_Scatter function takes all the data from process 0 and distributes it among all the processes including process 0 itself. Once all the processes have completed their tasks, MPI_Gather collects all the data from all the processes and sends it all to process 0. MPI_Bcast function sends the same information to all the processes, including process 0 (*6. Programming Using the Message-Passing Paradigm - Introduction to Parallel Computing, Second Edition*, n.d.).

The third algorithm will use collective communication method, this involves a group of processes that are all called within a communicator (Wilkinson & Allen, 2005). One of the advantages of using collective communication is that it can be easier to understand for the programmer. For the purposes of this design, first MPI_Scatter sends equal amount of data from the Matrix A (rows) to all the processes, then MPI_Bcast sends the whole of vector B to all the processes. Multiplication of matrix A with vector B is then carried out (Rows of matrix A * vector B) and saved. MPI_Gather then gathers this information and saves it in a resultant vector, I.e vector C. This algorithm is expected to perform better than algorithm 1 as rather than sending vector B to all the process individually and blocking until that sending is complete, MPI_Bcast will broadcast the whole vector B to all the processes simultaneously however, due to the broadcast process idling until it has been completed, this algorithm is not expected to perform better than algorithm 2 because algorithm 2 will carry on doing its work while waiting to send messages, and the process will start work instantaneously (Wilkinson & Allen, 2005).

5 Discussion and Results

The solution to the given problem statement was carried out, and this included obtaining and comparing results between the sequential algorithm with the 3 parallel algorithms as well as an external repository. The solution to the matrix with vector multiplication is checked for validity to ensure that all 4 algorithms provide the correct results. Moreover, the performance of all the algorithms are compared, for comparison reasons the sequential algorithm is run on the local laptop where all the code was developed as well as on the Cranfield University Supercomputer, Crescent. Parallel algorithms are all compiled and executed on Crescent using various processor sizes starting from 16 processors up to 64 processors, the results are recorded and compared, these performance results are shown and discussed below.

5.1 Algorithm 1

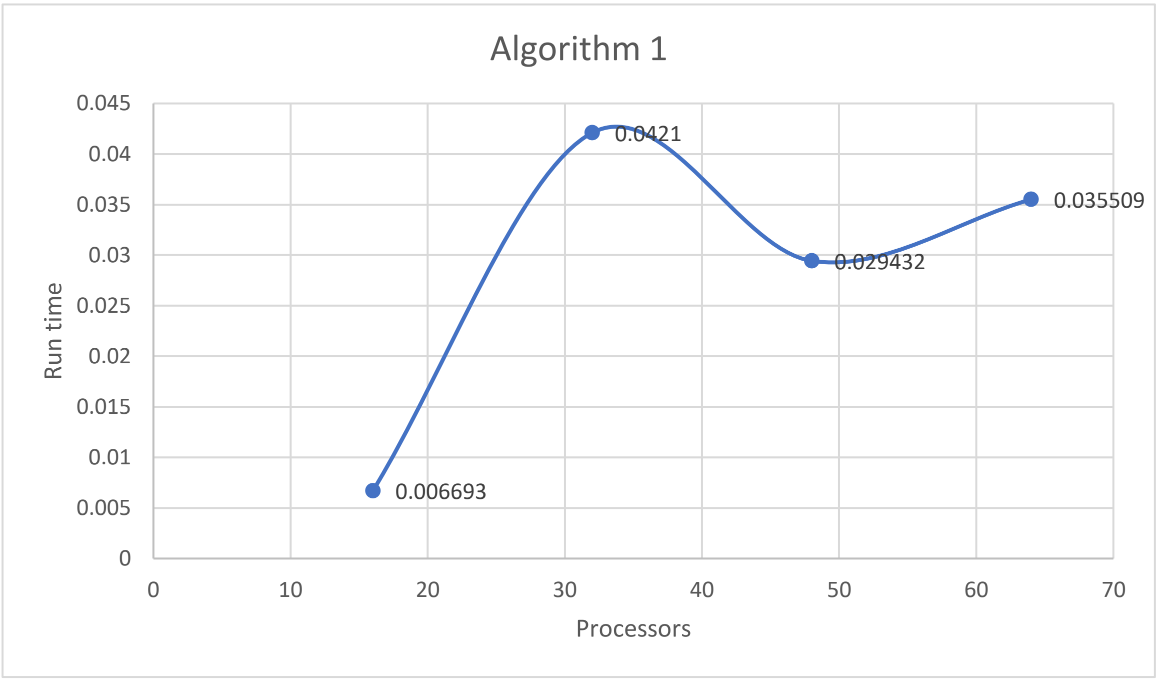

Fig 6: Algorithm 1 performance with varying numbers of processors

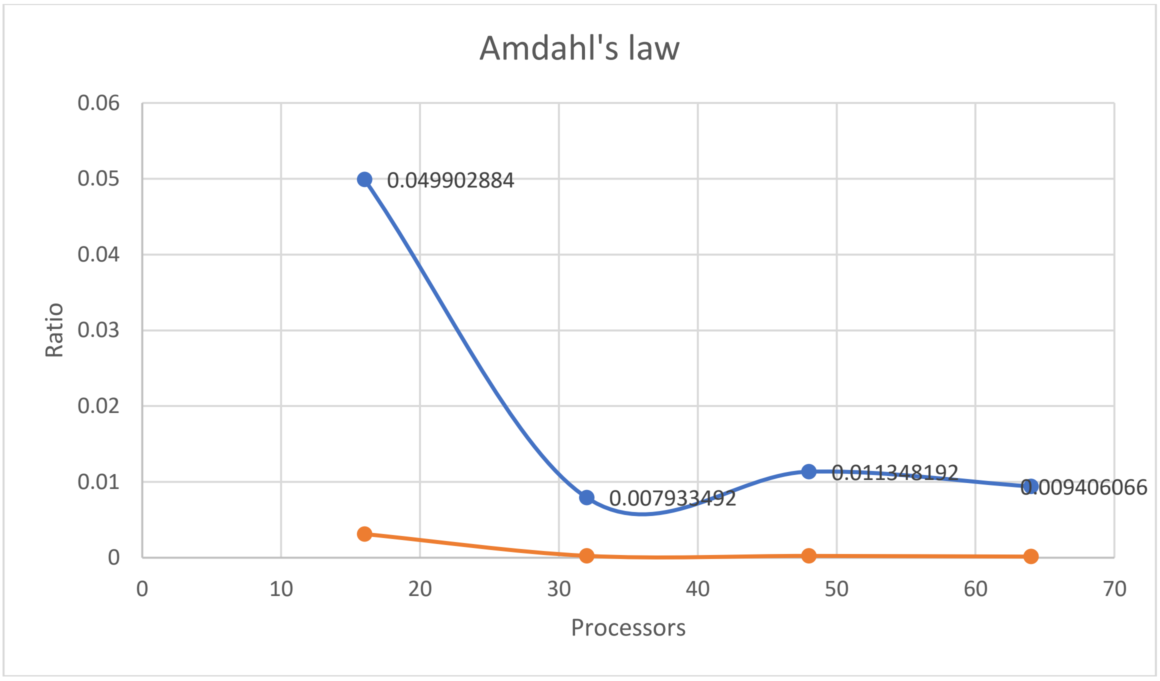

Fig 7: Amdahl's law and Efficiency for Algorithm one (Blue = Speed-up, Orange = Efficiency)

Figure 6 above shows algorithm 1 being tested across varying number of processors, the first point is tested on the local machine, the second point is 16 processors, third point 32 processors, fourth point 48 processors and fifth point 64 processors and these are tested on Crescent.

As can be seen from figure 6 and figure 7, that the fastest performance for this algorithm is with 16 processors. There are two reasons for this, one being that the size of the problem is not large enough to warrant a usage of a supercomputer and the second reason is as explained in the theory above, that as the number of processes have increased, the amount of idling time for the processes have also increased which has led to lower performance. Figure 7 shows the Amdahl's law for speed up and the efficiency, and as can be seen the performance is in constant decline, showing that at no point it is more beneficial to use a supercomputer for a problem this size, at least not with this blocking algorithm.

5.2 Algorithm 2

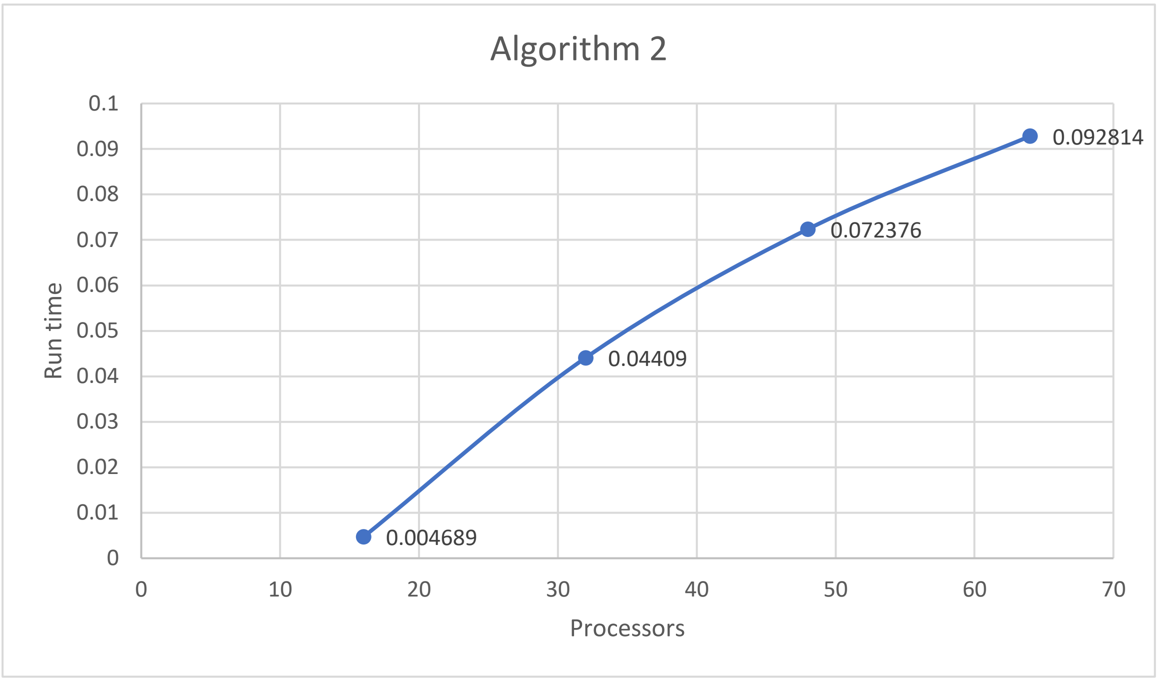

Fig 8: Algorithm 2 performance with varying numbers of processors

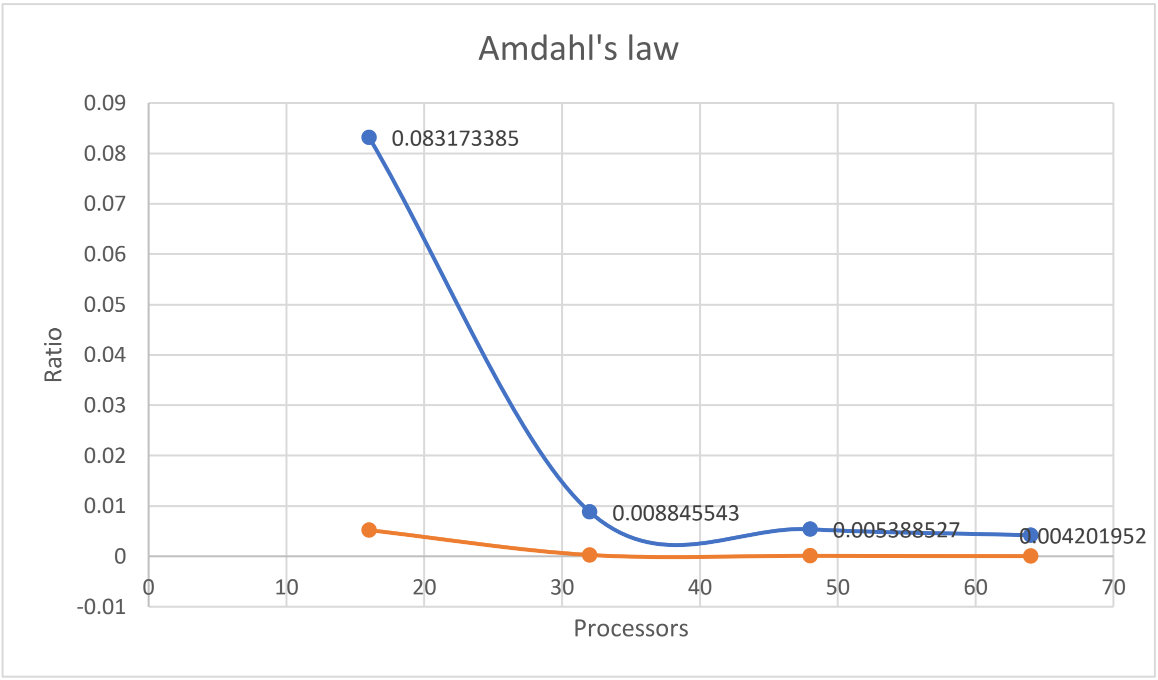

Fig 9: Amdahl's law and Efficiency for Algorithm two (Blue = Speed-up, Orange = Efficiency)

As expected, based on the theory above, algorithm two performed better than algorithm one. However, as can be seen from figure 8 and figure 9 that even in this case with non-blocking communication and a much better performance compared to algorithm one, the reason being that due to the size of this problem, the calculation can be carried out much faster on the local machine due to no extra communication required across multiple processes, even if they are sent to a buffer.

5.3 Algorithm 3

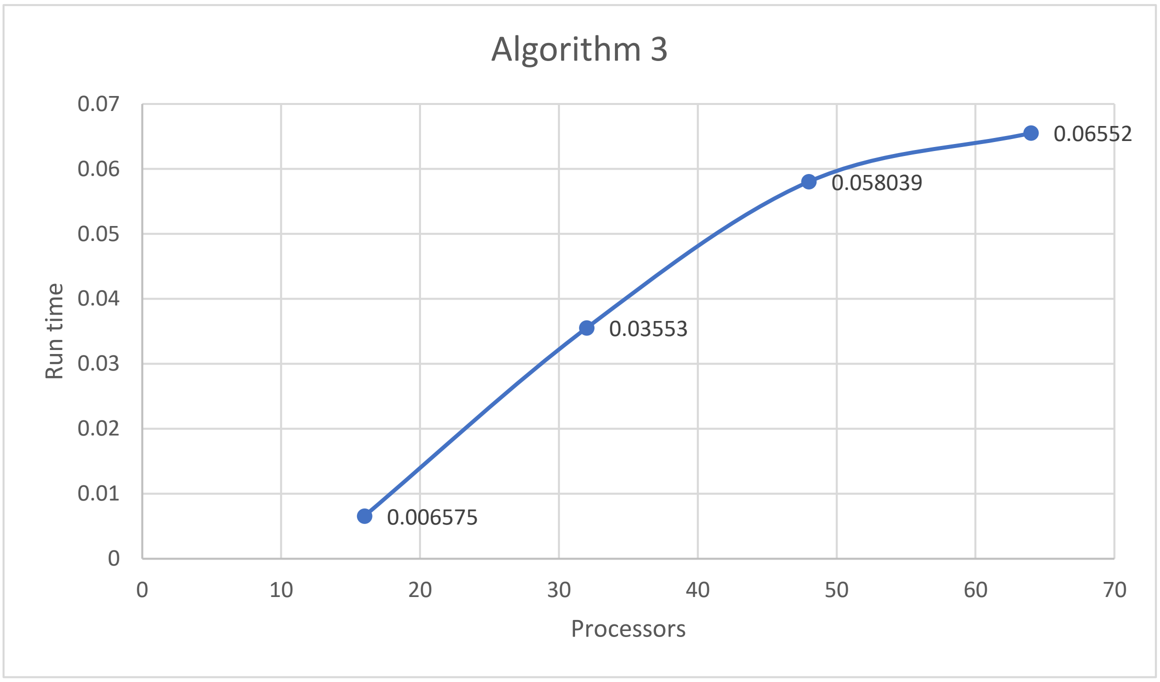

Fig 10: Algorithm 3 performance with varying numbers of processors

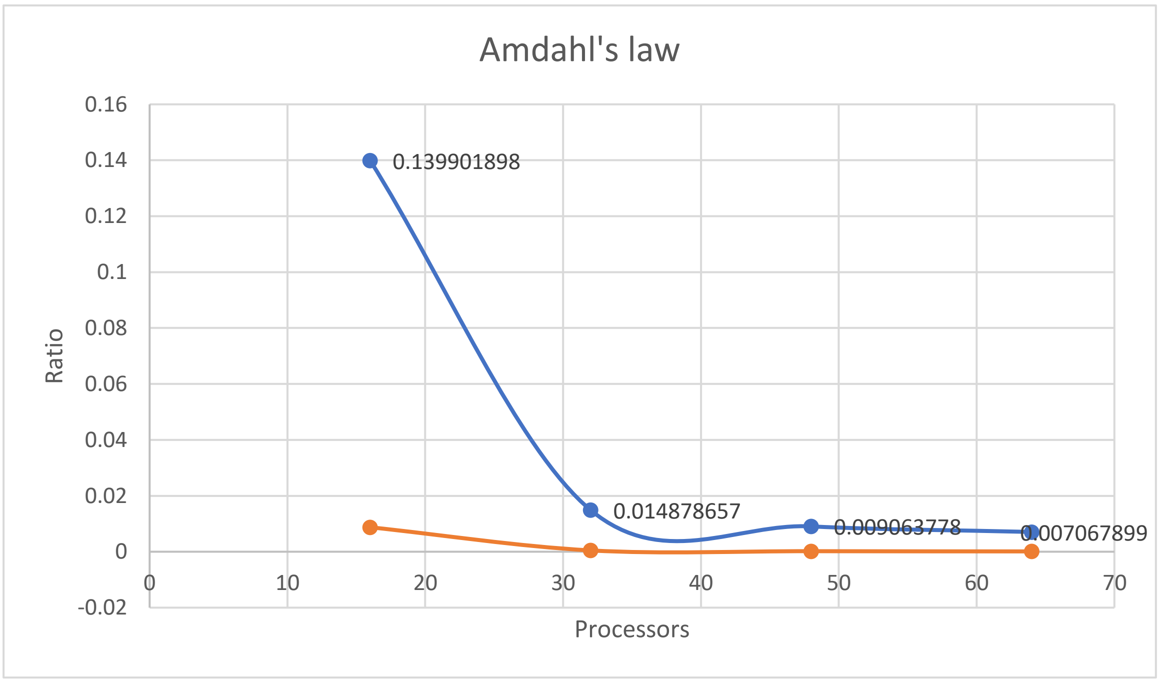

Fig 11: Amdahl's law and Efficiency for Algorithm three (Blue = Speed-up, Orange = Efficiency)

As described in the theory above, this algorithm performs better than algorithm one but not better than algorithm two. This is because when MPI_Bcast send vector B to all the process, the processes remain idle until MPI_Bcast has sent the vector to all the processes, making algorithm three faster than algorithm one but not algorithm two. Amdahl’s law above shows that there is no positive speed-up when using this algorithm and using this problem with this algorithm on a supercomputer doesn’t have any benefits in terms of performance.

5.4 External Library

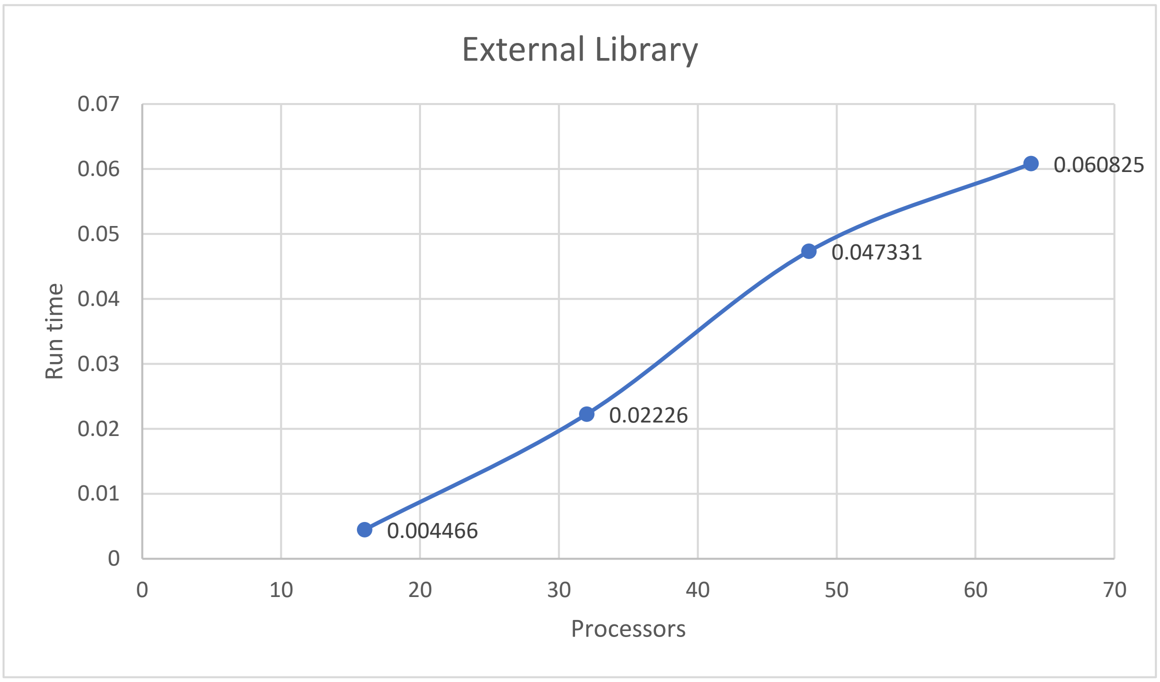

Fig 12: External Algorithm performance with varying numbers of processors

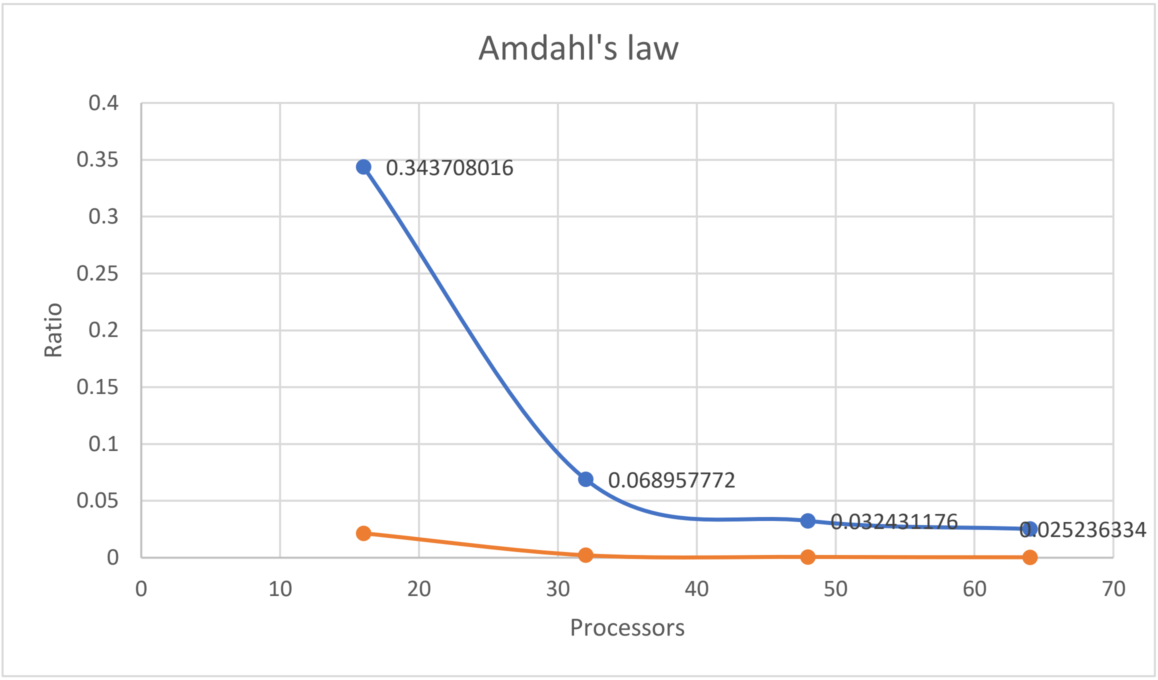

Fig 13: Amdahl's law and Efficiency for External Repository (Blue = Speed-up, Orange = Efficiency)

The external repository uses a mix of functions that are used in algorithm two and three. It uses non-blocking point-to-point communication as well collective communication. Comparing the external library to all three algorithms, it is slowest out of all four on the local computer as can be seen in fig 12 and 13. In terms of performance, it is very similar to that of algorithm two as it uses MPI_Isend for non-blocking communication and MPI_Bcast to send the vector to all the processes. However, even with this external it is clear that for the problem of the size that is used for the purposes of this assignment, there are no benefits of using a supercomputer as the fastest results are recoded on the local computer.

5.5 Sequential code

The sequential code as all the other algorithms are provided in the appendix below. In order to compare performance, the sequential code for this problem was written and tested. Below is a comparison of all the algorithms, including the external library and the sequential code when tested on the local computer.

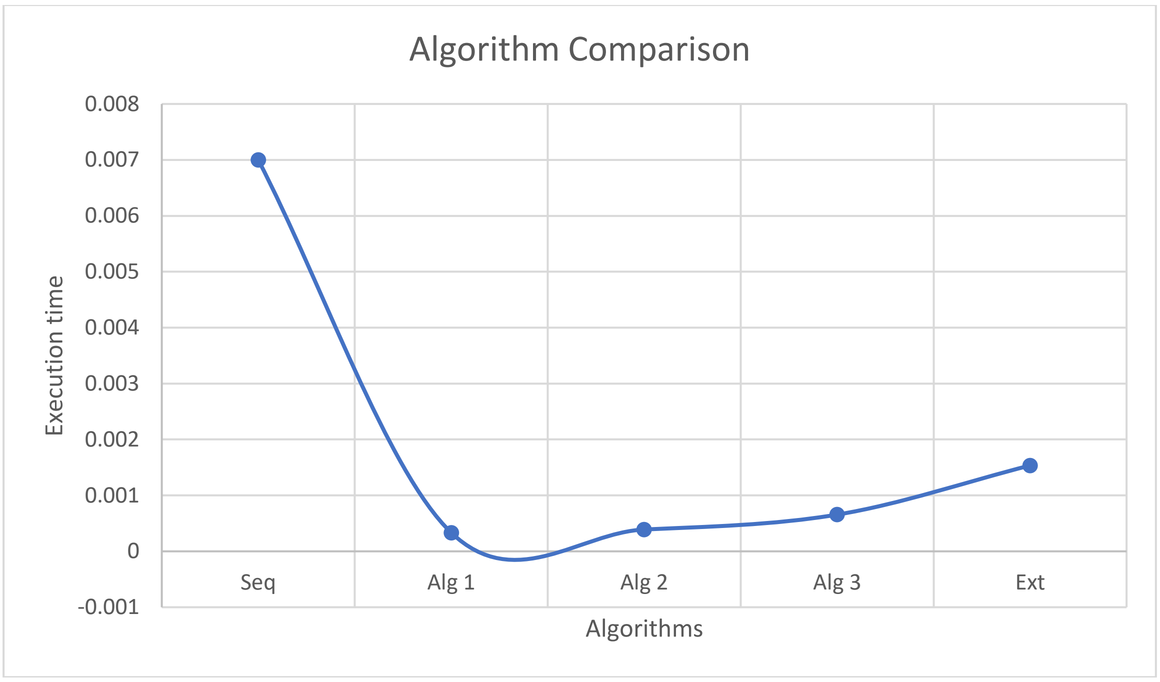

Fig 14: Algorithms comparison with the Sequential Code and External library

As can be seen from figure 14 that when tested on the local computer that has four cores, sequential code was the slowest, which would suggest that although it is somewhat useful to run the code using MPI due to the increased performance however, those gains are very small and due to the size of the problem the benefits are not large enough that would warrant a use of a supercomputer.

6 Future work

A thorough comparison has been made in this report between the three MPI parallel algorithms as well as the External repository that uses MPI and the sequential code. For the purposes of future work, a bigger problem size could be used to make full use of parallel programming and a supercomputer. Furthermore, perhaps instead of using integer matrices, floating point matrices could be used for the algorithms as that would require more computation power and would perhaps be able to make better use of the supercomputer. Moreover, the comparison can also be made using the Lapack math library to determine the performance difference using the algorithms written for this report and with the algorithms using a Lapack alongside MPI.

It has been noted that there was some confusion by the author in terms of a difference between algorithms and the implementation of those algorithms. It was assumed that using different communication systems would be different algorithms however, after further clarification it has been understood that the communication systems are merely the implementation of those algorithms, and not algorithms in themselves. Therefore, this piece of work only has 2 algorithms that has 3 different implementations between them. For the purposes of future work, another algorithm will be designed and tested to meet all the requirements of this assignment.

7 Conclusion

After comparing the performances above, for this problem size using a supercomputer may not be necessary. Although, there are improvements in the performance when using a supercomputer, considering how costly a time on a supercomputer can be, it does not warrant a use for a problem this size. Algorithm 1 provided the least amount of improvement due to using the blocking communication, as the idling time of processes started to increase due to the extra waiting time required per process. Algorithm 2 and the External library provided similar performance results however, the external library also used collective communication alongside the non-blocking communication. Algorithm 3, although performed better than Algorithm 1 and sequential code, it was slower than Algorithm 2 and the External library. This is due to Algorithm 3 only using collective communication, which as discussed in the theory is expected to be slower than non-blocking communication for this kind of problem.

8 Bibliography

6. Programming Using the Message-Passing Paradigm - Introduction to Parallel Computing, Second Edition. (n.d.). Retrieved January 18, 2021, from https://learning.oreilly.com/library/view/introduction-to-parallel/0201648652/ch06.html

Chapter 3. Distributed-Memory Programming with MPI - An Introduction to Parallel Programming. (n.d.). Retrieved January 18, 2021, from https://learning.oreilly.com/library/view/an-introduction-to/9780123742605/B9780123742605000038.xhtml#B978-0-12-374260-5.00003-8

matrix-vector-multiplication-using-mpi/main.c at master · topninja/matrix-vector-multiplication-using-mpi · GitHub. (n.d.). Retrieved January 18, 2021, from https://github.com/topninja/matrix-vector-multiplication-using-mpi/blob/master/main.c

Moulitsas, D. I. (2020). *High Performance Computing*.

Pachero, P. S. (1997). *Parallel programming with MPI*. Morgan Kaufman.

Wilkinson, B., & Allen, M. (2005). *PARALLEL PROGRAMMING TECHNIQUES AND APPLICATIONS USING NETWORKED WORKSTATIONS AND PARALLEL COMPUTERS 2nd Edition*.

Appendix A

Algorithm 1 – source Code

main.c

```
#include "algorithm.h"

int main(int argc, char *argv[])
{
   algorithm();

   return 0;
}
```

Algorithm.h

```
#ifndef ALG1_ALGORITHM_H
#define ALG1_ALGORITHM_H
void algorithm(void);
#endif //ALG1_ALGORITHM_H
```

Algorithm.c

```
#include <stdio.h>
#include "mpi.h"
#include "print_matrix.h"
#include "product_pp.h"
#include "matrix_declaration.h"

void algorithm(void)
{
   int comm_sz,       //number of processes
   my_rank,           //My process rank
   i, j, k,           //for looping purposes
   source,            // to be able to receive information from all the worker nodes.
   worker_nodes,      // the number of working nodes the work will be distributed to.
   dest;              //to be able to send work to worker nodes.
   double start, finish, global, duration;   //Variables to measure execution times.

   //Initialise MPI
   MPI_Init(NULL, NULL);
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   MPI_Status status;
```

```
    //to set the number of worker nodes and keep 1 node free as a master node
    worker_nodes = comm_sz - 1;

    //Send messages to the worker nodes.
    //Alg1
    start = MPI_Wtime();

    if (my_rank == 0) {
        //Send messages to worker nodes to carry out the calculations.
        for (dest = 1; dest <= worker_nodes; dest++) {
            MPI_Send(&a, R1 * C1, MPI_INT, dest, 1, MPI_COMM_WORLD);
            MPI_Send(&b, R2 * C2, MPI_INT, dest, 1, MPI_COMM_WORLD);
        }

        //Receive the completed multiplication from the worker nodes
        for (i = 1; i <= worker_nodes; i++) {
            source = i;
            MPI_Recv(&c, R1 * C2, MPI_INT, i, 2, MPI_COMM_WORLD, &status);
        }
        //print the resultant matrix
        print_matrix(c);
    } else {
        //Receive messages from the master node to carry out the multiplication.
        MPI_Recv(&a, R1 * C1, MPI_INT, 0, 1, MPI_COMM_WORLD, &status);
        MPI_Recv(&b, R2 * C2, MPI_INT, 0, 1, MPI_COMM_WORLD, &status);

        //Carry out the multiplication.
        product_pp(a, b, c);

        //Send the multiplication to the master node.
        MPI_Send(&c, R1 * C2, MPI_INT, 0, 2, MPI_COMM_WORLD);
    }
    //To measure run time of the algorithm.
    MPI_Barrier(MPI_COMM_WORLD);
    finish = MPI_Wtime();
    duration = finish - start;
    MPI_Reduce(&duration, &global, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
    if (my_rank == 0)
        printf("The run time is: %f\n", global);

    MPI_Finalize();
}
```

matrix_declaration.h

```
#ifndef ALG1_MATRIX_DECLARATION_H
#define ALG1_MATRIX_DECLARATION_H
//Matrix A, Vector B and resultant Matrix C declarations.

int a[R1][C1] = {
        {1, 2, 0, 0, 3, 0, 0, 4},
        {5, 6, 7, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 8, 9, 0},
        {0, 10, 11, 12, 0, 0, 0, 0},
        {0, 0, 13, 14, 0, 0, 0, 0},
        {15, 0, 0, 0, 16, 17, 0, 18},
        {19, 0, 0, 0, 20, 0, 0, 21},
        {0, 0, 0, 0, 22, 23, 24, 0},
        {0, 0, 0, 0, 0, 25, 26, 0},
        {27, 0, 0, 0, 28, 0, 0, 29}
};

int b[R2][C2] = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9},
        {10, 11, 12},
        {13, 14, 15},
        {16, 17, 18},
        {19, 20, 21},
        {22, 23, 24}
};

int c[R1][C2];          //Resultant array
#endif //ALG1_MATRIX_DECLARATION_H
```

print_matrix.c

```
#include <stdio.h>

#define R1 10
#define C2 3

void print_matrix(int c[R1][C2])
{
   int i, j;

   printf("The resultant matrix is:\n");
   for(i = 0; i < R1; i++)
   {
        for(j = 0; j < C2; j++)
        {
            printf("%4d   ", c[i][j]);
        }
        printf("\n");
    }
}
```

print_matrix.h

```
#ifndef ALG1_PRINT_MATRIX_H
#define ALG1_PRINT_MATRIX_H
//Prototype for print_matrix function.

#define R1 10
#define C2 3

void print_matrix(int c[R1][C2]);
#endif //ALG1_PRINT_MATRIX_H
```

product_pp.c

```
#define R1 10
#define C1 8
#define R2 8
#define C2 3

void product_pp(int a[R1][C1], int b[R2][C2], int c[R1][C2])
{
    int i, j, k;

    for (i = 0; i < 10; i++) {
        for (k = 0; k < 3; k++) {
            c[i][k] = 0;
            for (j = 0; j < 8; j++) {
                c[i][k] = c[i][k] + a[i][j] * b[j][k];
            }
        }
    }
}
```

product_pp.h

```
#ifndef ALG1_PRODUCT_PP_H
#define ALG1_PRODUCT_PP_H
//Prototype for product_pp function.
#define R1 10
```

```
#define C1 8
#define R2 8
#define C2 3

void product_pp(int a[R1][C1], int b[R2][C2], int c[R1][C2]);
#endif //ALG1_PRODUCT_PP_H
```

Algorithm 2 – source code

main.c

```
#include "algorithm.h"

int main(int argc, char *argv[]) {

   algorithm();

   return 0;
}
```

algorithm.c

```
#include <stdio.h>
#include "mpi.h"
#include "print_matrix.h"
#include "product_pp.h"
#include "matrix_declaration.h"

void algorithm(void)
{
   int comm_sz,       //number of processes
   my_rank,           //My process rank
   i, j, k,           //for looping purposes
   source,            // to be able to receive information from all the worker nodes.
   worker_nodes,      // the number of working nodes the work will be distributed to.
   dest;              //to be able to send work to worker nodes.
   double start, finish, global, duration;   //Variables to measure execution times.

   //Initialise MPI
   MPI_Init(NULL, NULL);
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   MPI_Status status;
   //For MPI wait, needed to have the next process wait until the previous one is completed.
   MPI_Request request[6];
```

```
//to set the number of worker nodes and keep 1 node free as a master node
worker_nodes = comm_sz - 1;

start = MPI_Wtime();

if (my_rank == 0) {
   //Send messages to worker nodes to carry out the calculations.
   for (dest = 1; dest <= worker_nodes; dest++) {
      MPI_Isend(&a, R1 * C1, MPI_INT, dest, 1, MPI_COMM_WORLD, &request[0]);
      MPI_Isend(&b, R2 * C2, MPI_INT, dest, 2, MPI_COMM_WORLD, &request[1]);

      //Wait for the messages to be completed.
      MPI_Wait(&request[0], &status);
      MPI_Wait(&request[1], &status);
   }
   //Receive the completed multiplication from worker nodes.
   for (i = 1; i <= worker_nodes; i++) {
      MPI_Irecv(&c, R1 * C2, MPI_INT, i, 3, MPI_COMM_WORLD, &request[5]);
      //Wait for the messages to be completed.
      MPI_Wait(&request[5], &status);
   }
   //Print the resultant matrix.
   print_matrix(c);
}

if (my_rank > 0) {
   //Receive messages from the master node to carry out the multiplication.
   MPI_Irecv(&a, R1 * C1, MPI_INT, 0, 1, MPI_COMM_WORLD, &request[2]);
   MPI_Irecv(&b, R2 * C2, MPI_INT, 0, 2, MPI_COMM_WORLD, &request[3]);

   //Wait for the messages to be completed.
   MPI_Wait(&request[2], &status);
   MPI_Wait(&request[3], &status);

   //Carry out the multiplication.
   product_pp(a, b, c);

   //Send the multiplication to master node.
   MPI_Isend(&c, R1 * C2, MPI_INT, 0, 3, MPI_COMM_WORLD, &request[4]);

   //Wait for the messages to be completed.
   MPI_Wait(&request[4], &status);
}
//To measure run time of the algorithm.
MPI_Barrier(MPI_COMM_WORLD);
finish = MPI_Wtime();
duration = finish - start;
```

```
   MPI_Reduce(&duration, &global, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
   if (my_rank == 0)
      printf("The run time is: %f\n", global);

   MPI_Finalize();
}
```

algorithm.h

```
#ifndef ALG2_ALGORITHM_H
#define ALG2_ALGORITHM_H
void algorithm(void);
#endif //ALG2_ALGORITHM_H
```

matrix_declaration.h

```
#ifndef ALG2_MATRIX_DECLARATION_H
#define ALG2_MATRIX_DECLARATION_H
//Matrix A, Vector B and resultant Matrix C declarations.

int a[R1][C1] = {
      {1, 2, 0, 0, 3, 0, 0, 4},
      {5, 6, 7, 0, 0, 0, 0, 0},
      {0, 0, 0, 0, 0, 8, 9, 0},
      {0, 10, 11, 12, 0, 0, 0, 0},
      {0, 0, 13, 14, 0, 0, 0, 0},
      {15, 0, 0, 0, 16, 17, 0, 18},
      {19, 0, 0, 0, 20, 0, 0, 21},
      {0, 0, 0, 0, 22, 23, 24, 0},
      {0, 0, 0, 0, 0, 25, 26, 0},
      {27, 0, 0, 0, 28, 0, 0, 29}
};

int b[R2][C2] = {
      {1, 2, 3},
      {4, 5, 6},
      {7, 8, 9},
      {10, 11, 12},
      {13, 14, 15},
      {16, 17, 18},
      {19, 20, 21},
      {22, 23, 24}
};

int c[R1][C2];         //Resultant array
#endif //ALG2_MATRIX_DECLARATION_H
```

print_matrix.c

```
#include <stdio.h>

#define R1 10
#define C2 3

void print_matrix(int c[R1][C2])
{
   int i, j;

   printf("The resultant matrix is:\n");
   for(i = 0; i < R1; i++)
   {
      for(j = 0; j < C2; j++)
      {
         printf("%4d   ", c[i][j]);
      }
      printf("\n");
   }
}
```

print_matrix.h

```
#ifndef ALG2_PRINT_MATRIX_H
#define ALG2_PRINT_MATRIX_H
//Prototype for print_matrix function.

#define R1 10
#define C2 3

void print_matrix(int c[R1][C2]);
#endif //ALG2_PRINT_MATRIX_H
```

product_pp.c

```
#define R1 10
#define C1 8
#define R2 8
#define C2 3

void product_pp(int a[R1][C1], int b[R2][C2], int c[R1][C2])
{
   int i, j, k;

   for (i = 0; i < 10; i++) {
      for (k = 0; k < 3; k++) {
         c[i][k] = 0;
         for (j = 0; j < 8; j++) {
            c[i][k] = c[i][k] + a[i][j] * b[j][k];
         }
      }
   }
}
```

product_pp.h

```
#ifndef ALG2_PRODUCT_PP_H
#define ALG2_PRODUCT_PP_H
//Prototype for product_pp function.
#define R1 10
#define C1 8
#define R2 8
#define C2 3

void product_pp(int a[R1][C1], int b[R2][C2], int c[R1][C2]);
#endif //ALG2_PRODUCT_PP_H
```

Algorithm 3 – source code

main.c

```
#include "algorithm.h"

int main(int argc, char *argv[])
{
   algorithm();

   return 0;
}
```

algorithm.c

```
#include <stdio.h>
#include "mpi.h"
#include "print_matrix.h"
#include "product_cc.h"
#include "matrix_declaration.h"

void algorithm(void)
{
   int comm_sz,       //number of processes
   my_rank,           //My process rank
   i, j, k,           //for looping purposes
   source,            // to be able to receive information from all the worker nodes.
   worker_nodes,      // the number of working nodes the work will be distributed to.
   dest;              //to be able to send work to worker nodes.
   double start, finish, global, duration;   //Variables to measure execution times.

   //Initialise MPI
   MPI_Init(NULL, NULL);
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   MPI_Status status;

   //to set the number of worker nodes and keep 1 node free as a master node
   worker_nodes = comm_sz-1;

   start = MPI_Wtime();

   int aRows[R1];     //To save the rows from matrix A.
   int cRows[R1];     //To save the resultant matrix.

   //Scatter rows from Matrix A.
   MPI_Scatter(a, R1/worker_nodes, MPI_INT, aRows, R1/worker_nodes, MPI_INT, 0,
MPI_COMM_WORLD);

   //Broadcast matrix B to all the processes.
   MPI_Bcast(b, R2 * C2, MPI_INT, 0, MPI_COMM_WORLD);

   //Carry out the multiplication.
   product_cc(a, b, c, cRows);

   //Gather data from all the processes and save it in matrix C.
   MPI_Gather(cRows, R1/worker_nodes, MPI_INT, c, R1/worker_nodes, MPI_INT, 0,
MPI_COMM_WORLD);
```

```
    //Print the resultant matrix.
    if(my_rank == 1) {
        print_matrix(c);
    }
    //To measure run time of the algorithm.
    MPI_Barrier(MPI_COMM_WORLD);
    finish = MPI_Wtime();
    duration = finish - start;
    MPI_Reduce(&duration, &global, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
    if(my_rank == 0)
        printf("The run time is: %f\n", global);

    MPI_Finalize();
}
```

algorithm.h

```
#ifndef ALG3_ALGORITHM_H
#define ALG3_ALGORITHM_H
void algorithm(void);
#endif //ALG3_ALGORITHM_H
```

matrix_declaration.h

```
#ifndef ALG3_MATRIX_DECLARATION_H
#define ALG3_MATRIX_DECLARATION_H
//Matrix A, Vector B and resultant Matrix C declarations.

int a[R1][C1] = {
        {1, 2, 0, 0, 3, 0, 0, 4},
        {5, 6, 7, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 8, 9, 0},
        {0, 10, 11, 12, 0, 0, 0, 0},
        {0, 0, 13, 14, 0, 0, 0, 0},
        {15, 0, 0, 0, 16, 17, 0, 18},
        {19, 0, 0, 0, 20, 0, 0, 21},
        {0, 0, 0, 0, 22, 23, 24, 0},
        {0, 0, 0, 0, 0, 25, 26, 0},
        {27, 0, 0, 0, 28, 0, 0, 29}
};

int b[R2][C2] = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9},
        {10, 11, 12},
        {13, 14, 15},
        {16, 17, 18},
        {19, 20, 21},
        {22, 23, 24}
};

int c[R1][C2];          //Resultant array
#endif //ALG3_MATRIX_DECLARATION_H
```

print_matrix.c

```
#include <stdio.h>

#define R1 10
#define C2 3

void print_matrix(int c[R1][C2])
{
    int i, j;

    printf("The resultant matrix is:\n");
    for(i = 0; i < R1; i++)
    {
        for(j = 0; j < C2; j++)
        {
            printf("%4d   ", c[i][j]);
        }
        printf("\n");
    }
}
```

print_matrix.h

```
#ifndef ALG3_PRINT_MATRIX_H
#define ALG3_PRINT_MATRIX_H
//Prototype for print_matrix function.

#define R1 10
#define C2 3

void print_matrix(int c[R1][C2]);
#endif //ALG3_PRINT_MATRIX_H
```

product_cc.c

```
#define R1 10
#define C1 8
#define R2 8
```

```
#define C2 3

void product_cc(int a[R1][C1], int b[R2][C2], int c[R1][C2], int cRows[R1])
{
   int i, j, k;

   for (i = 0; i < 10; i++) {
      for (k = 0; k < 3; k++) {
         c[i][k] = 0;
         for (j = 0; j < 8; j++) {
            c[i][k] = c[i][k] + a[i][j] * b[j][k];
         }
         cRows[i] = c[i][k];
      }
   }
}
```

product_cc.h

```
#ifndef ALG3_PRODUCT_CC_H
#define ALG3_PRODUCT_CC_H
//Prototype for product_cc function.

#define R1 10
#define C1 8
#define R2 8
#define C2 3

void product_cc(int a[R1][C1], int b[R2][C2], int c[R1][C2], int cRows[R1]);
#endif //ALG3_PRODUCT_CC_H
```

External Library – source code

(*Matrix-Vector-Multiplication-Using-Mpi/Main.c at Master · Topninja/Matrix-Vector-Multiplication-Using-Mpi · GitHub*, n.d.)

main.c

```
#include<stdio.h>
#include<mpi.h>
#define MATRIX_ROWS 10
#define MATRIX_COLUMNS 8
#define VECTOR_ROWS 8
#define VECTOR_COLUMNS 3
#define MASTER_TO_SLAVE_TAG 1
#define SLAVE_TO_MASTER_TAG 4
int matrix[MATRIX_ROWS][MATRIX_COLUMNS] = {
        {1, 2, 0, 0, 3, 0, 0, 4},
        {5, 6, 7, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 8, 9, 0},
        {0, 10, 11, 12, 0, 0, 0, 0},
        {0, 0, 13, 14, 0, 0, 0, 0},
        {15, 0, 0, 0, 16, 17, 0, 18},
        {19, 0, 0, 0, 20, 0, 0, 21},
        {0, 0, 0, 0, 22, 23, 24, 0},
        {0, 0, 0, 0, 0, 25, 26, 0},
        {27, 0, 0, 0, 28, 0, 0, 29}
};

int vector[VECTOR_ROWS][VECTOR_COLUMNS] = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9},
        {10, 11, 12},
        {13, 14, 15},
        {16, 17, 18},
        {19, 20, 21},
        {22, 23, 24}
};

int result[MATRIX_ROWS][VECTOR_COLUMNS];        //Resultant array
int rank,size,i,j,k,low_bound,upper_bound,portion;
double start_time, end_time;
MPI_Status status;
MPI_Request request;
int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (rank == 0) {
        start_time = MPI_Wtime();
        for (i = 1; i < size; i++) {
            portion = (MATRIX_ROWS / (size - 1));
            low_bound = (i - 1) * portion;
            if (((i + 1) == size) && ((MATRIX_ROWS % (size - 1)) != 0)) {
                upper_bound = MATRIX_ROWS;
            } else {
                upper_bound = low_bound + portion;
            }
            MPI_Isend(&low_bound, 1, MPI_INT, i, MASTER_TO_SLAVE_TAG,
MPI_COMM_WORLD, &request);
            MPI_Isend(&upper_bound, 1, MPI_INT, i, MASTER_TO_SLAVE_TAG + 1,
MPI_COMM_WORLD, &request);
```

```
         MPI_Isend(&matrix[low_bound][0], (upper_bound - low_bound) * MATRIX_ROWS,
MPI_INT, i, MASTER_TO_SLAVE_TAG + 2, MPI_COMM_WORLD, &request);
      }
      if(size==1){
         for (i = 0; i < MATRIX_ROWS; i++) {
            for (j = 0; j < VECTOR_COLUMNS; j++) {
               result[i][j]=0;
               for (k = 0; k < VECTOR_ROWS; k++) {
                  result[i][j] += (matrix[i][k] * vector[k][j]);
               }
            }
         }
      }
   }
   MPI_Bcast(&vector, VECTOR_ROWS*VECTOR_COLUMNS, MPI_INT, 0,
MPI_COMM_WORLD);
   if (rank > 0) {
      MPI_Recv(&low_bound, 1, MPI_INT, 0, MASTER_TO_SLAVE_TAG, MPI_COMM_WORLD,
&status);
      MPI_Recv(&upper_bound, 1, MPI_INT, 0, MASTER_TO_SLAVE_TAG + 1,
MPI_COMM_WORLD, &status);
      MPI_Recv(&matrix[low_bound][0], (upper_bound - low_bound) * MATRIX_ROWS,
MPI_INT, 0, MASTER_TO_SLAVE_TAG + 2, MPI_COMM_WORLD, &status);
      for (i = low_bound; i < upper_bound; i++) {
         for (j = 0; j < VECTOR_COLUMNS; j++) {
            result[i][j]=0;
            for (k = 0; k < VECTOR_ROWS; k++) {
               result[i][j] += (matrix[i][k] * vector[k][j]);
            }
         }
      }
      MPI_Isend(&low_bound, 1, MPI_INT, 0, SLAVE_TO_MASTER_TAG,
MPI_COMM_WORLD, &request);
      MPI_Isend(&upper_bound, 1, MPI_INT, 0, SLAVE_TO_MASTER_TAG + 1,
MPI_COMM_WORLD, &request);
      MPI_Isend(&result[low_bound][0], (upper_bound - low_bound) * VECTOR_COLUMNS,
MPI_INT, 0, SLAVE_TO_MASTER_TAG + 2, MPI_COMM_WORLD, &request);
   }
   if (rank == 0) {
      for (i = 1; i < size; i++) {
         MPI_Recv(&low_bound, 1, MPI_INT, i, SLAVE_TO_MASTER_TAG,
MPI_COMM_WORLD, &status);
         MPI_Recv(&result[low_bound][0], (upper_bound - low_bound) * VECTOR_COLUMNS,
MPI_INT, i, SLAVE_TO_MASTER_TAG + 2, MPI_COMM_WORLD, &status);
      }
      end_time = MPI_Wtime();
      printf("\nRunning Time = %f\n\n", end_time - start_time);
```

```
        for(i=0;i<MATRIX_ROWS;i++){
            for(j=0; j < VECTOR_COLUMNS; j++)
                printf("%4d ",result[i][j]);

            printf("\n");
        }
    }
    MPI_Finalize();
    return 0;
}
```

MPI sub file

```
#!/bin/bash
##
## MPI submission script for PBS on CRESCENT
## ----------------------------------------
##
## Follow the 6 steps below to configure your job
##
## STEP 1:
##
## Enter a job name after the -N on the line below:
##
#PBS -N main_4
##
## STEP 2:
##
## Select the number of cpus/cores required by modifying the #PBS -l select line below
##
## Normally you select cpus in chunks of 16 cpus
## The Maximum value for ncpus is 16 and mpiprocs MUST be the same value as ncpus.
##
## If more than 16 cpus are required then select multiple chunks of 16
## e.g. 16 CPUs: select=1:ncpus=16:mpiprocs=16
##      32 CPUs: select=2:ncpus=16:mpiprocs=16
##      48 CPUs: select=3:ncpus=16:mpiprocs=16
##      ..etc..
##
#PBS -l select=1:ncpus=16:mpiprocs=16
##
## STEP 3:
##
## Select the correct queue by modifying the #PBS -q line below
##
## half_hour   -  30 minutes
## one_hour    -   1 hour
## three_hour   -   3 hours
## six_hour     -   6 hours
## half_day     -  12 hours
## one_day      -  24 hours
## two_day      -  48 hours
## five_day     - 120 hours
## ten_day      - 240 hours (by special arrangement)
##
#PBS -q half_hour
##
## STEP 4:
##
## Replace the hpc@cranfield.ac.uk email address
## with your Cranfield email address on the #PBS -M line below:
## Your email address is NOT your username
##
#PBS -m abe
#PBS -M aman.makkar@cranfield.ac.uk
##
## ====================================
## DO NOT CHANGE THE LINES BETWEEN HERE
## ====================================
#PBS -j oe
#PBS -W sandbox=PRIVATE
#PBS -k n
ln -s $PWD $PBS_O_WORKDIR/$PBS_JOBID
## Change to working directory
cd $PBS_O_WORKDIR
## Calculate number of CPUs
export cpus=`cat $PBS_NODEFILE | wc -l`
## ========
## AND HERE
## ========
##
## STEP 5:
##
##  Load the default application environment
##  For a specific version add the version number, e.g.
##  module load intel/2016b
##
module load intel
##
## STEP 6:
##
## Run MPI code
##
## The main parameter to modify is your mpi program name
## - change YOUR_EXECUTABLE to your own filename
##

mpirun -machinefile $PBS_NODEFILE -np ${cpus} ./a.out

## Tidy up the log directory
## DO NOT CHANGE THE LINE BELOW
## ============================
rm $PBS_O_WORKDIR/$PBS_JOBID
#
```