6004CEM Parallel and Distributed Programming

Portfolio

Chan Khai Shen

Table of Contents

Task 1

1-Parallelism

Open multi-programming (OpenMP) is a tool to support parallel programming. OpenMP is regulated by the OpenMP Architecture Review Board and is supported by several hardware and software vendors (Womack, Wang, Flynn, Yi, & Yan, 2022). OpenMP is a cross compiler tool because mainstream compilers provide good support to OpenMP functionalities and are using the same set of standards (Womack, Wang, Flynn, Yi, & Yan, 2022). OpenMP is used for shared memory systems with multiple core central processing unit (CPU) or multiple CPU (Womack, Wang, Flynn, Yi, & Yan, 2022). OpenMP is available in C/C++ and Fortran languages.

Compute Unified Device Architecture (CUDA) is a tool to support parallel programming using graphical processing unit (GPU). CUDA was first launched by Ian Buck of Nvidia in 2006, for doing general purpose computing on Nvidia's GPU (Heller, 2022). CUDA is capable of speeding up parallelizable compute intensive tasks by harnessing the computation power of GPU (Heller, 2022). CUDA has been adopted in areas that need high performance computation and is used by deep learning frameworks as their GPU support (Heller, 2022).

Both OpenMP and CUDA are frameworks that provide support for parallel programming. Both frameworks use the Same Instruction, Multiple Data (SIMP) program model, which the parallel region runs the same code, but each thread or process has its own data (Womack, Wang, Flynn, Yi, & Yan, 2022). Both frameworks work on a shared memory system.

OpenMP is segregate tasks to different cores of a multiple core CPU or multiple CPU to achieve parallel processing, whereas CUDA uses the high number of cores in GPU to perform parallel processing tasks. OpenMP is supported by mainstream compilers by using the same set of standards regulated by the OpenMP Architecture Review Board, whereas CUDA is specific to Nvidia's GPU.

CUDA is more suitable than OpenMP if the task need is highly computation intensive and need to use GPU. OpenMP might be suitable if the task needs to parallelize the for loop which has high number of iterations or needs to increase speed of recursive summation using reduction.

Currently, CUDA is widely used in areas that need high performance computing, such as data science and deep learning (Heller, 2022). In future, since there might be more applications of machine learning and thus the demand in high performance computing, CUDA might be more adopted in machine learning fields. Similarly, OpenMP might also be more widely adopted in fields that need intensive computation because of its ability to implement parallel processing.

2-Distribution

Open multi-programming (OpenMP) is a tool to support parallel programming in a shared memory system. The standards of OpenMP are regulated by the OpenMP Architecture Review Board backed by several hardware and software vendors and mainstream compilers are providing good support to OpenMP functionalities (Womack, Wang, Flynn, Yi, & Yan, 2022). OpenMP is used for shared memory systems with multiple core central processing unit (CPU) or multiple CPU (Womack, Wang, Flynn, Yi, & Yan, 2022). OpenMP is available in C/C++ and Fortran languages.

Message Passing Interface (MPI) is a tool to support parallel programming in a distributed memory system or a cluster of nodes. Although not officially endorsed, MPI is generally adopted by the industry as the standardized means of controlling and synchronizing processes and exchanging data between processes across a distributed memory system (Gillis & Bigelow, 2022). MPI is used for distributed memory systems in a cluster of multiple computers or within a computer with multiple cores (Gillis & Bigelow, 2022). MPI is available in C/C++ and Fortran languages.

Both OpenMP and MPI are frameworks that support parallel programming. Both frameworks use the Same Instruction, Multiple Data (SIMP) program model, which the parallel region runs the same code, but each thread or process has its own data (Womack, Wang, Flynn, Yi, & Yan, 2022). Both frameworks are able to use multiple cores on a computer to achieve parallel processing. In contrast to vendor-specific CUDA, both OpenMP and MPI are cross platform and are the common standard in the industry.

OpenMP works based on a shared memory system which variables can be specified as “shared” or “private”, whereas MPI works based on a distributed memory system which data is passed between processes.

MPI is more suitable than OpenMP if the task is run on a cluster of computers because it OpenMP does not support distributed memory. OpenMP might be suitable if the task is run on a single computer and needs to parallelize for-loop with very high number of iterations or increase speed of recursive summation using reduction.

In future, MPI might be more widely adopted in fields that require large volume of memory and high performance computation, such as data science and big data programming, which cluster of computers would need to be used. OpenMP might be more widely used in computation intensive tasks, such as machine learning, which parallelizing part of the task can increase the speed.

Task 2

Parallel Programming Portfolio Element

Part A

“ParallelPartA_c.c” is answering the first part of the question, which is printing hello world with 10 threads, whereas “ParallelPartA_b.c” is answering the second part of the question by not specifying the number of threads, but leaving it to be decided by altering environment variable “OMP_NUM_THREADS”. Figure 1. and Figure 2. are the screenshots of “ParallelPartA_a.c” and “ParallelPartA_b.c” running.

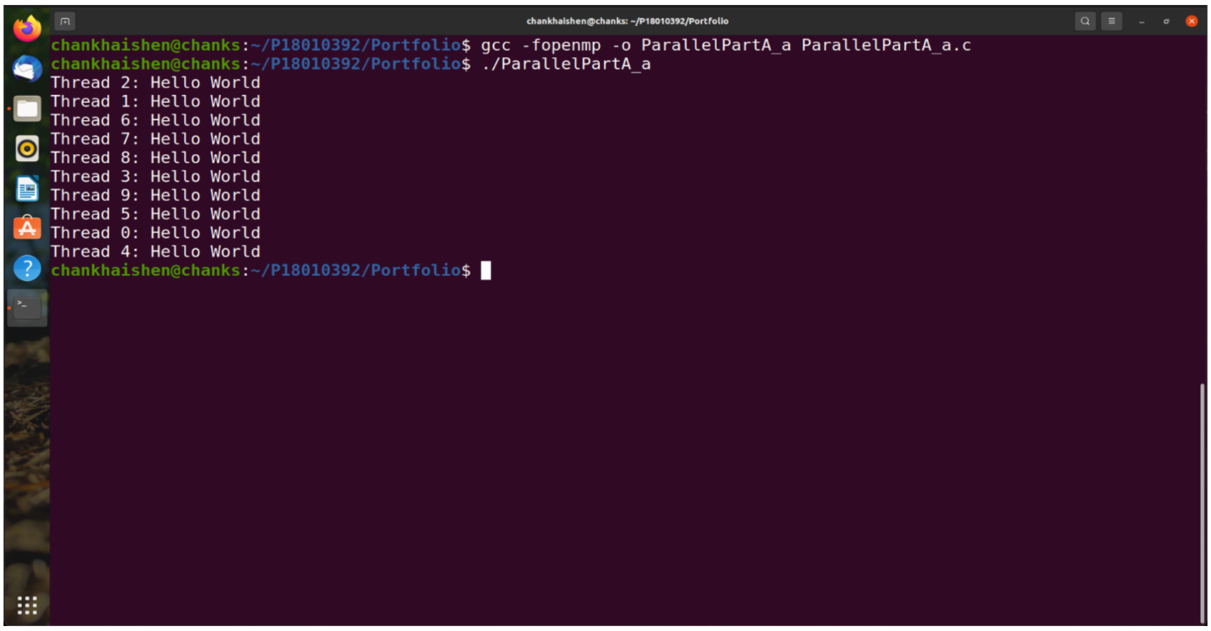
ParallelPartA_a.c

```
#include <stdio.h>

#include <omp.h>


int main() {

        #pragma omp parallel num_threads(10)

        {

                printf("Thread %d: Hello World\n", omp_get_thread_num());

        }


        return 0;

}
```

ParallelPartA_b.c

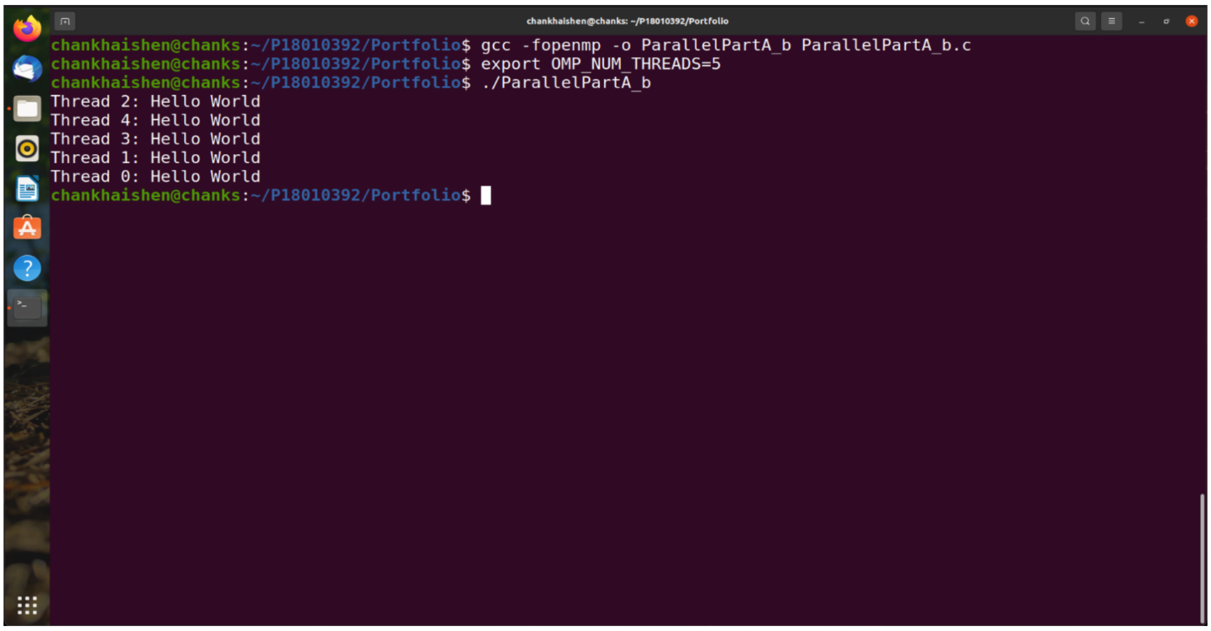
```
#include <stdio.h>

#include <omp.h>
```

```
int main() {

        #pragma omp parallel

        {

                printf("Thread %d: Hello World\n", omp_get_thread_num());

        }


        return 0;

}
```



```
chankhaishen@chanks:~/P18010392/Portfolio$ gcc -fopenmp -o ParallelPartA_a ParallelPartA_a.c
chankhaishen@chanks:~/P18010392/Portfolio$ ./ParallelPartA_a
Thread 2: Hello World
Thread 1: Hello World
Thread 6: Hello World
Thread 7: Hello World
Thread 8: Hello World
Thread 3: Hello World
Thread 9: Hello World
Thread 5: Hello World
Thread 0: Hello World
Thread 4: Hello World
chankhaishen@chanks:~/P18010392/Portfolio$
```

Figure 1 Screenshot of ParallelPartA_a.c running



```
chankhaishen@chanks:~/P18010392/Portfolio$ gcc -fopenmp -o ParallelPartA_b ParallelPartA_b.c
chankhaishen@chanks:~/P18010392/Portfolio$ export OMP_NUM_THREADS=5
chankhaishen@chanks:~/P18010392/Portfolio$ ./ParallelPartA_b
Thread 2: Hello World
Thread 4: Hello World
Thread 3: Hello World
Thread 1: Hello World
Thread 0: Hello World
chankhaishen@chanks:~/P18010392/Portfolio$ 
```

Figure 2 Screenshot of ParallelPartA_b.c running

Part B

"ParallelPartB_a.c" is answering the first part of the question, which is adding two vectors of length 2000 using parallel for construct with static schedule and the default chunk size, whereas "ParallelPartB_b.c" is answering the second part, which is using dynamic schedule and chunk size of 100. Figure 3. and Figure 4. are the screenshots of "ParallelPartB_a.c" and "ParallelPartB_b.c" running.

ParallelPartB_a.c

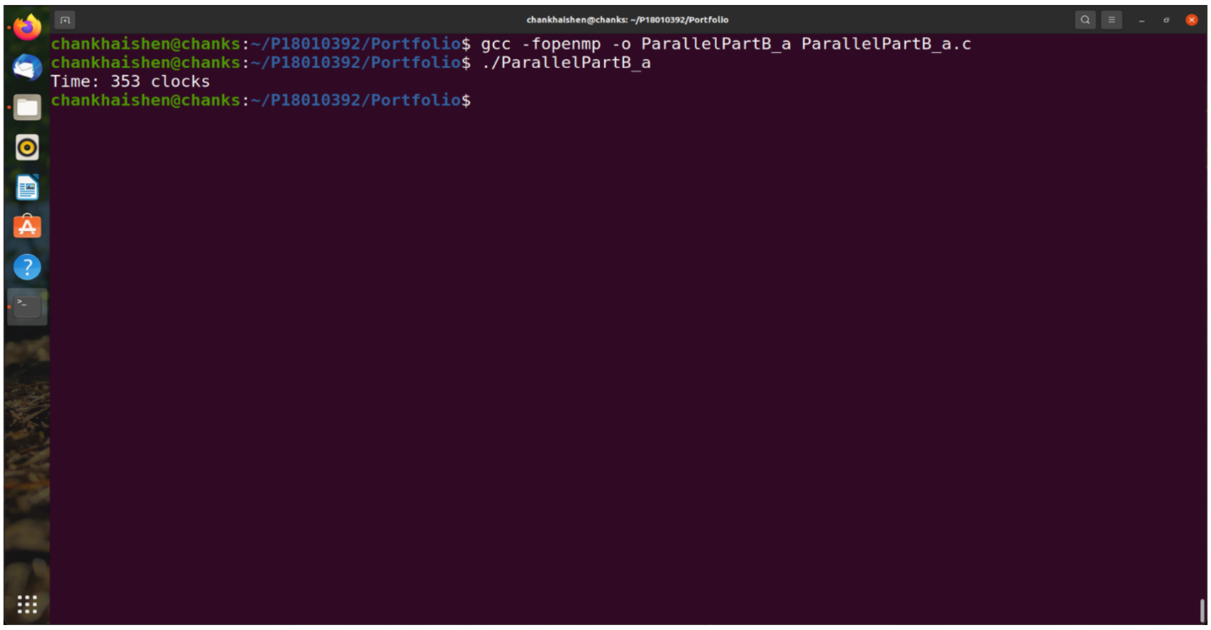
```
#include <stdio.h>

#include <omp.h>

#include <time.h>



int main() {

        int VECTOR_SIZE = 2000;

        int vector1[VECTOR_SIZE], vector2[VECTOR_SIZE], vectorSum[VECTOR_SIZE];
        for (int i = 0; i < VECTOR_SIZE; i++) {

                vector1[i] = (i+1)*100;

        }



        for (int i = 0; i < VECTOR_SIZE; i++) {

                vector2[i] = i+3;

        }



        clock_t start = clock();

        #pragma omp parallel for schedule(static) num_threads(4)

        for (int i = 0; i < VECTOR_SIZE; i++) {

                vectorSum[i] = vector1[i] + vector2[i];

        }

        clock_t end = clock();

        printf("Time: %ld clocks\n", end-start);



        return 0;

}
```

ParallelPartB_b.c

```
#include <stdio.h>

#include <omp.h>
```

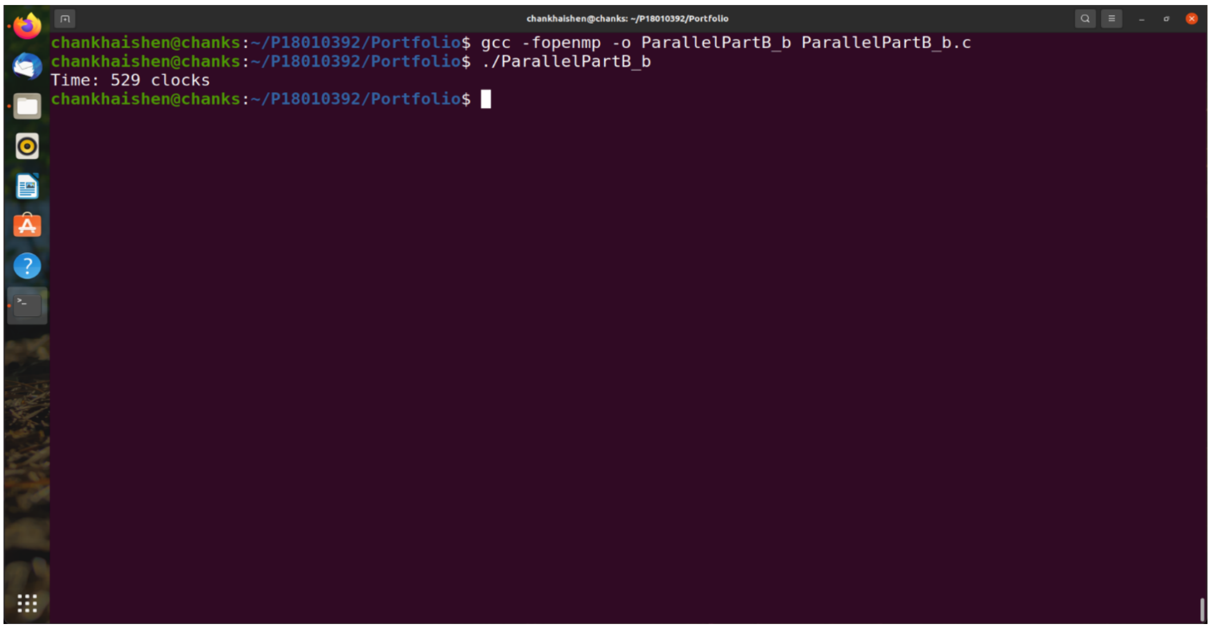
```
#include <time.h>


int main() {

        int VECTOR_SIZE = 2000;

        int vector1[VECTOR_SIZE], vector2[VECTOR_SIZE], vectorSum[VECTOR_SIZE];



        for (int i = 0; i < VECTOR_SIZE; i++) {

                vector1[i] = (i+1)*100;

        }



        for (int i = 0; i < VECTOR_SIZE; i++) {

                vector2[i] = i+3;

        }



        clock_t start = clock();

        #pragma omp parallel for schedule(dynamic, 100) num_threads(4)

        for (int i = 0; i < VECTOR_SIZE; i++) {

                vectorSum[i] = vector1[i] + vector2[i];

        }

        clock_t end = clock();

        printf("Time: %ld clocks\n", end-start);



        return 0;
```

}



```
chankhaishen@chanks:~/P18010392/Portfolio$ gcc -fopenmp -o ParallelPartB_a ParallelPartB_a.c
chankhaishen@chanks:~/P18010392/Portfolio$ ./ParallelPartB_a
Time: 353 clocks
chankhaishen@chanks:~/P18010392/Portfolio$
```

Figure 3 Screenshot of ParallelPartB_a.c running



```
chankhaishen@chanks:~/P18010392/Portfolio$ gcc -fopenmp -o ParallelPartB_b ParallelPartB_b.c
chankhaishen@chanks:~/P18010392/Portfolio$ ./ParallelPartB_b
Time: 529 clocks
chankhaishen@chanks:~/P18010392/Portfolio$
```

Figure 4 Screenshot of ParallelPartB_b.c running

My conclusion is the execution time when using static scheduling is shorter than that of dynamic scheduling. This might be because the amount of work of each iteration of vector addition is almost equal, so there is no slow down when using static scheduling. Due to dynamic scheduling uses more resources, so it is slower.

Part C

"ParallelPartC_a.cpp" does matrix multiplication in serial. "ParallelPartC_b.cpp" does matrix multiplication by parallelizing the outermost loop, whereas "ParallelPartC_c.cpp" does matrix multiplication by parallelizing the innermost loop. Since each iteration does the same set of steps and there are no if-else branches in the steps, so the amount of work of each iteration is likely to be equal. Thus, the scheduling of the parallel section is set as "static" and the chunk size is the default chunk size. Figure 5., Figure 6. And Figure 7. are the screenshots of "ParallelPartC_a.cpp", "ParallelPartC_b.cpp" and "ParallelPartC_c.cpp" running.

ParallelPartC_a.cpp

```
#include <iostream>

#include <omp.h>

#include <vector>

#include <chrono>


using namespace std;


int main() {

        int N;

        cout << "Size of matrix: ";

        cin >> N;


        vector<vector<double>> matrixA(N, vector<double>(N));
```

```
vector<vector<double>> matrixB(N, vector<double>(N));

vector<vector<double>> matrixResult(N, vector<double>(N));


// Initialization

for (int i = 0; i < N; i++) {

        for (int j = 0; j < N; j++) {

                matrixA[i][j] = 386.58745*i + 12.847381*j;

                matrixB[i][j] = 4788.4572*i + 18373.712*j;

        }

}


auto start = chrono::high_resolution_clock::now();

// 10 test runs

for (int run = 0; run < 10; run++) {

        // Matrix multiplication

        for (int i = 0; i < N; i++) {

                for (int j = 0; j < N; j++) {

                        int sum = 0;

                        for (int k = 0; k < N; k++) {

                                sum += matrixA[i][k] * matrixB[k][j];

                        }

                        matrixResult[i][j] = sum;

                }
        }
    }
    auto end = chrono::high_resolution_clock::now();
    auto duration = chrono::duration_cast<chrono::microseconds>(end - start);

    cout << "Average time used: " << duration.count()/10 << " microseconds" << endl;

    return 0;
}
```

ParallelPartC_b.cpp

```
#include <iostream>
#include <omp.h>
#include <vector>
#include <chrono>

using namespace std;

int main() {
    int N;
    cout << "Size of matrix: ";
    cin >> N;
```

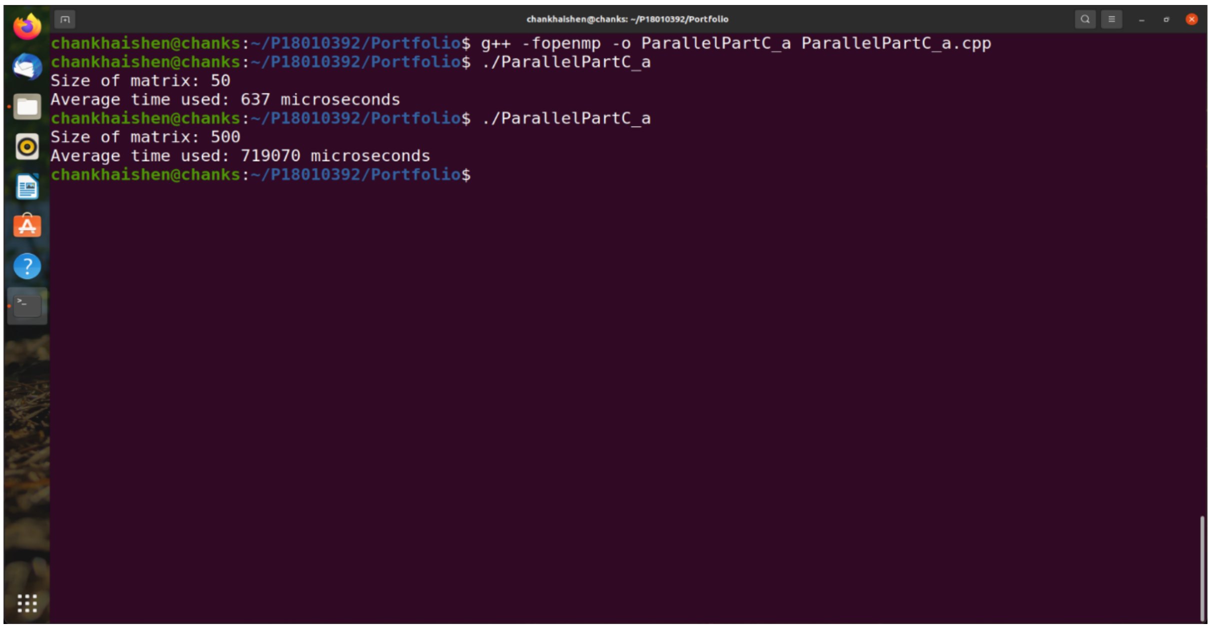
```
	vector<vector<double>> matrixA(N, vector<double>(N));

	vector<vector<double>> matrixB(N, vector<double>(N));

	vector<vector<double>> matrixResult(N, vector<double>(N));



	// Initialization

	for (int i = 0; i < N; i++) {

		for (int j = 0; j < N; j++) {

			matrixA[i][j] = 386.58745*i + 12.847381*j;

			matrixB[i][j] = 4788.4572*i + 18373.712*j;

		}

	}



	// 10 test runs

	auto start = chrono::high_resolution_clock::now();

	for (int run = 0; run < 10; run++) {

		#pragma omp parallel for schedule(static) shared(matrixA, matrixB,
matrixResult)

		// Matrix multiplication

		for (int i = 0; i < N; i++) {

			for (int j = 0; j < N; j++) {

				int sum = 0;

				for (int k = 0; k < N; k++) {

					sum += matrixA[i][k] * matrixB[k][j];
```

```
                        }
                        matrixResult[i][j] = sum;
                    }
                }
            }
            auto end = chrono::high_resolution_clock::now();
            auto duration = chrono::duration_cast<chrono::microseconds>(end - start);

            cout << "Time used: " << duration.count()/10 << " microseconds" << endl;

            return 0;
}
```

ParallelPartC_c.cpp

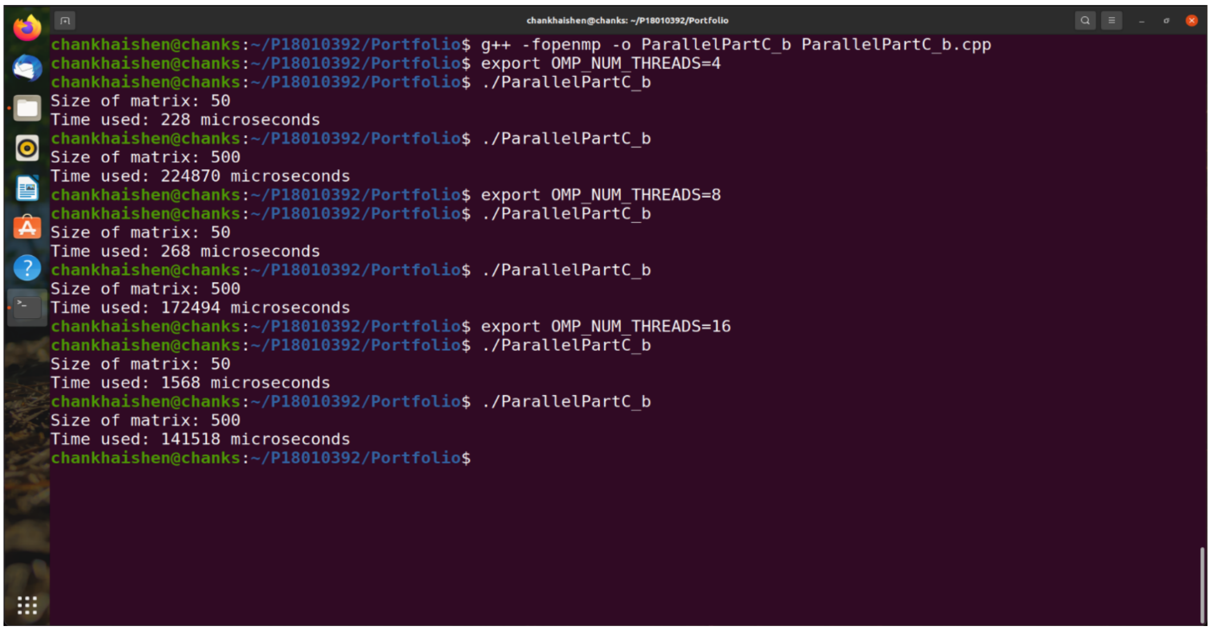
```
#include <iostream>
#include <omp.h>
#include <vector>
#include <chrono>

using namespace std;

int main() {
            int N;
```

```
cout << "Size of matrix: ";

cin >> N;


vector<vector<double>> matrixA(N, vector<double>(N));

vector<vector<double>> matrixB(N, vector<double>(N));

vector<vector<double>> matrixResult(N, vector<double>(N));


// Initialization

for (int i = 0; i < N; i++) {

        for (int j = 0; j < N; j++) {

                matrixA[i][j] = 386.58745*i + 12.847381*j;

                matrixB[i][j] = 4788.4572*i + 18373.712*j;

        }

}


// 10 test runs

auto start = chrono::high_resolution_clock::now();

for (int run = 0; run < 10; run++) {\

        // Matrix multiplication

        for (int i = 0; i < N; i++) {

                for (int j = 0; j < N; j++) {

                        int sum = 0;
                        #pragma omp parallel for schedule(static) firstprivate(i, j)
shared(matrixA, matrixB, matrixResult) reduction(+: sum)

                        for (int k = 0; k < N; k++) {

                            sum += matrixA[i][k] * matrixB[k][j];

                        }

                        matrixResult[i][j] = sum;

                    }

                }

            }

    auto end = chrono::high_resolution_clock::now();

    auto duration = chrono::duration_cast<chrono::microseconds>(end - start);



    cout << "Time used: " << duration.count()/10 << " microseconds" << endl;



    return 0;

}
```



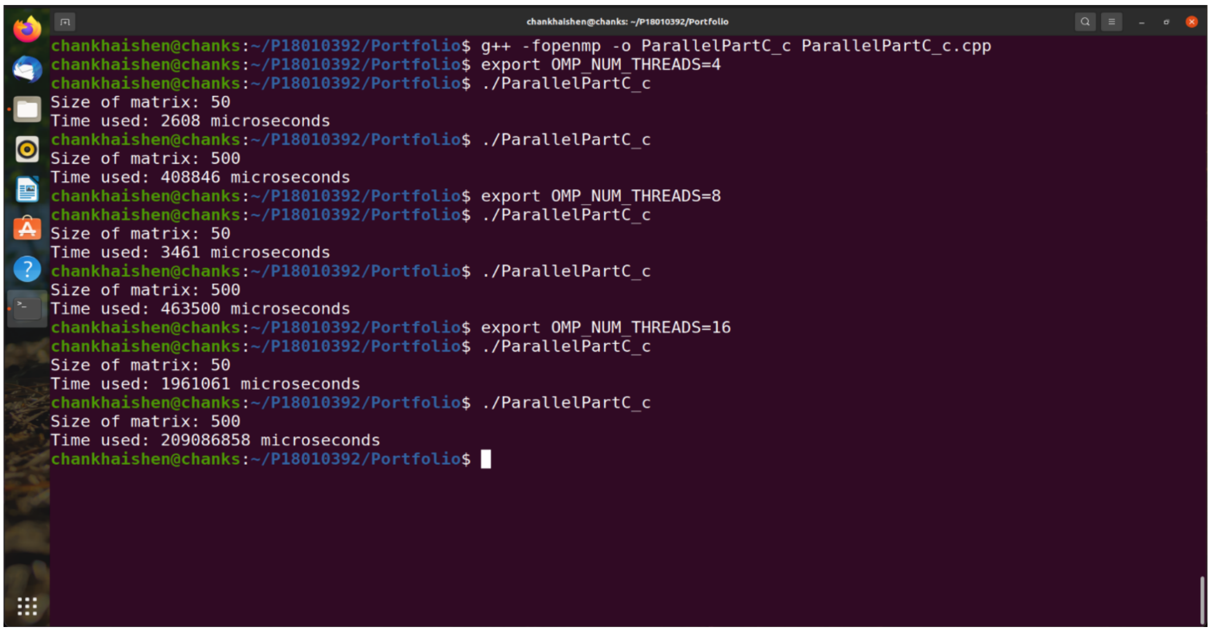
```
chankhaishen@chanks:~/P18010392/Portfolio$ g++ -fopenmp -o ParallelPartC_a ParallelPartC_a.cpp
chankhaishen@chanks:~/P18010392/Portfolio$ ./ParallelPartC_a
Size of matrix: 50
Average time used: 637 microseconds
chankhaishen@chanks:~/P18010392/Portfolio$ ./ParallelPartC_a
Size of matrix: 500
Average time used: 719070 microseconds
chankhaishen@chanks:~/P18010392/Portfolio$
```

Figure 5 Screenshot of ParallelPartC_a.cpp running



```
chankhaishen@chanks:~/P18010392/Portfolio$ g++ -fopenmp -o ParallelPartC_b ParallelPartC_b.cpp
chankhaishen@chanks:~/P18010392/Portfolio$ export OMP_NUM_THREADS=4
chankhaishen@chanks:~/P18010392/Portfolio$ ./ParallelPartC_b
Size of matrix: 50
Time used: 228 microseconds
chankhaishen@chanks:~/P18010392/Portfolio$ ./ParallelPartC_b
Size of matrix: 500
Time used: 224870 microseconds
chankhaishen@chanks:~/P18010392/Portfolio$ export OMP_NUM_THREADS=8
chankhaishen@chanks:~/P18010392/Portfolio$ ./ParallelPartC_b
Size of matrix: 50
Time used: 268 microseconds
chankhaishen@chanks:~/P18010392/Portfolio$ ./ParallelPartC_b
Size of matrix: 500
Time used: 172494 microseconds
chankhaishen@chanks:~/P18010392/Portfolio$ export OMP_NUM_THREADS=16
chankhaishen@chanks:~/P18010392/Portfolio$ ./ParallelPartC_b
Size of matrix: 50
Time used: 1568 microseconds
chankhaishen@chanks:~/P18010392/Portfolio$ ./ParallelPartC_b
Size of matrix: 500
Time used: 141518 microseconds
chankhaishen@chanks:~/P18010392/Portfolio$
```

Figure 6 Screenshot of ParallelPartC_b.cpp running



```
chankhaishen@chanks:~/P18010392/Portfolio$ g++ -fopenmp -o ParallelPartC_c ParallelPartC_c.cpp
chankhaishen@chanks:~/P18010392/Portfolio$ export OMP_NUM_THREADS=4
chankhaishen@chanks:~/P18010392/Portfolio$ ./ParallelPartC_c
Size of matrix: 50
Time used: 2608 microseconds
chankhaishen@chanks:~/P18010392/Portfolio$ ./ParallelPartC_c
Size of matrix: 500
Time used: 408846 microseconds
chankhaishen@chanks:~/P18010392/Portfolio$ export OMP_NUM_THREADS=8
chankhaishen@chanks:~/P18010392/Portfolio$ ./ParallelPartC_c
Size of matrix: 50
Time used: 3461 microseconds
chankhaishen@chanks:~/P18010392/Portfolio$ ./ParallelPartC_c
Size of matrix: 500
Time used: 463500 microseconds
chankhaishen@chanks:~/P18010392/Portfolio$ export OMP_NUM_THREADS=16
chankhaishen@chanks:~/P18010392/Portfolio$ ./ParallelPartC_c
Size of matrix: 50
Time used: 1961061 microseconds
chankhaishen@chanks:~/P18010392/Portfolio$ ./ParallelPartC_c
Size of matrix: 500
Time used: 209086858 microseconds
chankhaishen@chanks:~/P18010392/Portfolio$
```

Figure 7 Screenshot of ParallelPartC_c.cpp running

Table 1. records the average time of 50*50 and 500*500 matrix multiplication with serial code and with outer loop parallelization and inner loop parallelization using 4, 8 and 16 threads. Charts in Figure 8. and Figure 9. visualise the trends of average time used for 50*50 and 500*500 matrix multiplication respectively. In both charts, the average time of matrix multiplication with inner loop parallelization using 16 threads is excluded because the number is nearly five hundred time larger than the longest time among other cases, including it will cover up the trend in other cases.

Table 1 Average time used for matrix multiplication

Size of matrix	**Type of parallelization**	**Number of threads**	**Average time used per test run (microsecond)**
50*50	No parallelization (serial)	1	637
	Outer loop parallelization	4	228
		8	268
		16	1568
	Inner loop parallelization	4	2608

		8	3461
		16	1961061
500*500	No parallelization (serial)	1	719070
	Outer loop parallelization	4	224870
		8	172494
		16	141518
	Inner loop parallelization	4	408846
		8	463500
		16	209086858

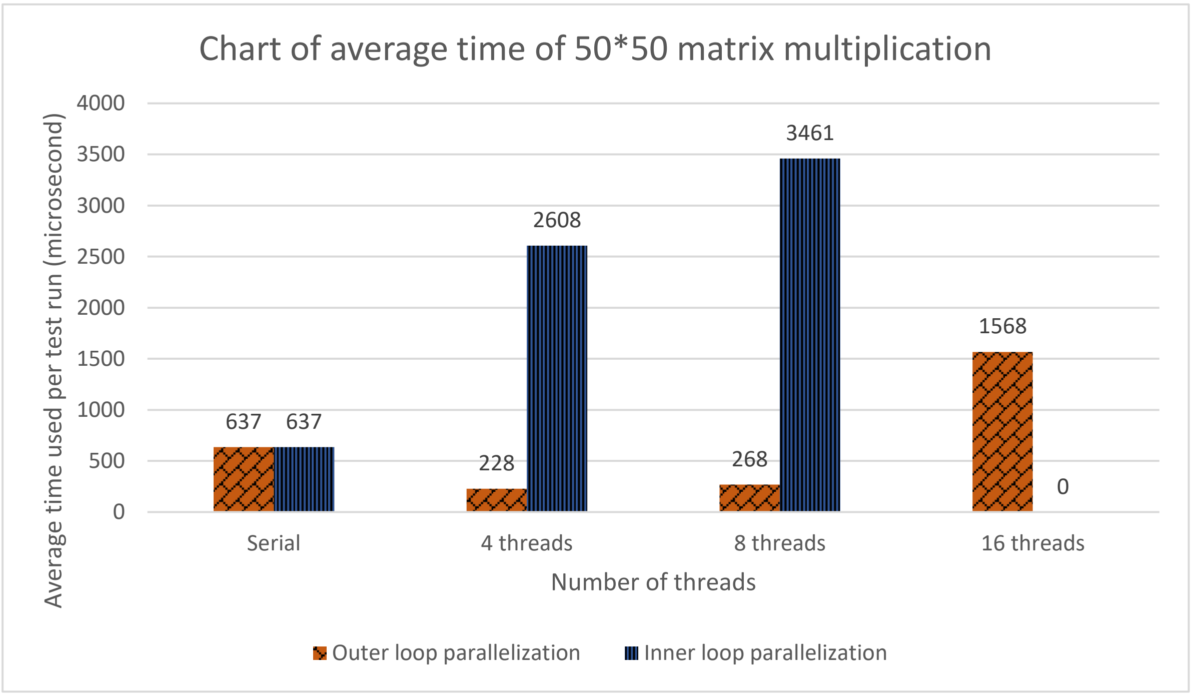


Figure 8 Chart of average time of 50*50 matrix multiplication

Based on Figure 8., the time used for 50*50 matrix multiplication using outer loop parallelization reduces when the number of threads increases, and reaches the shortest time at 4 threads, then increases after that, whereas the time used when using inner loop parallelization increases when the number of threads increases. Comparing serial and 4 threads, outer loop parallelization uses shorter time then serial, whereas inner loop parallelization uses longer time than serial.

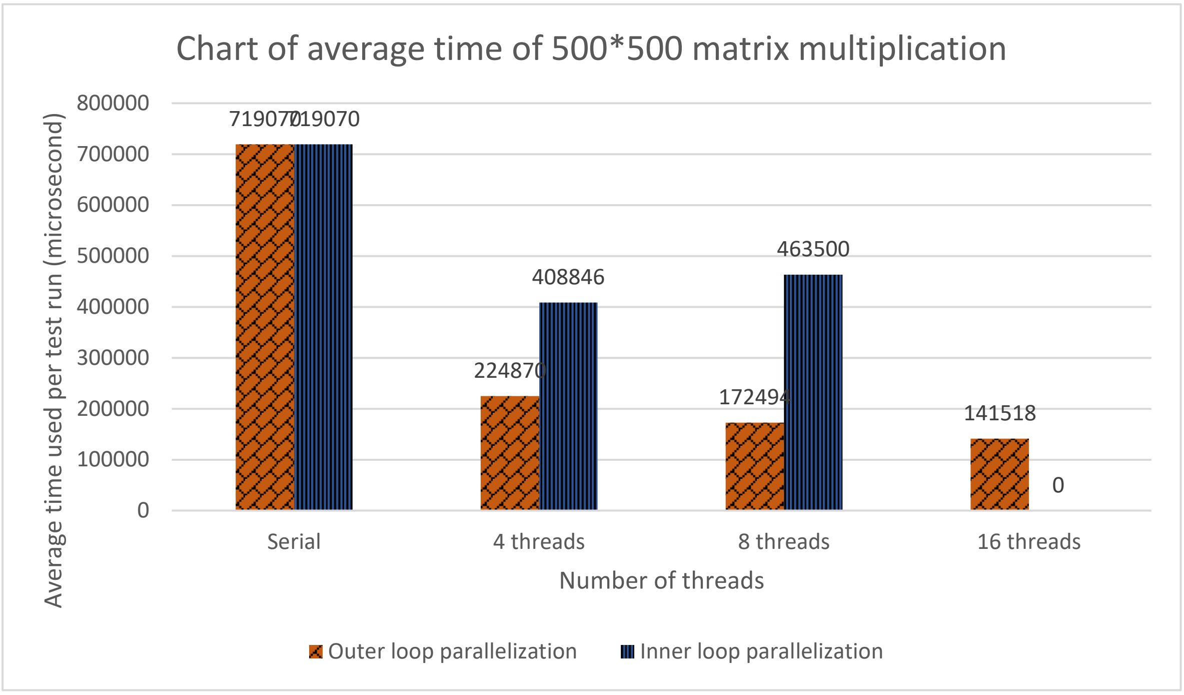


Figure 9 Chart of average time of 500*500 matrix multiplication

Based on Figure 9., the time used for 500*500 matrix multiplication using outer loop parallelization reduces when the number of threads increases, whereas the time used when using inner loop parallelization reduces and reaches the shortest time at 4 threads, then increases after that (note than time used for 16 threads is much more than the rest). It can be logically estimated that the time used for outer loop parallelization can still be reduced if the number of threads increases.

General speaking, both type of parallelization reduces the time of processing, but outer loop parallelization reduces the time of parallelization more compared to inner loop parallelization. This might be because the resources used to manage the parallelization for inner loop parallelization is more than outer loop parallelization since inner loop parallelization has larger number of tasks (2500 times more for 50*50 matrix and 250000 times more for 500*500 matrix) than outer loop parallelization.

Another observation is that when the heavier the work load, the more efficient parallelization is. When the number of threads increases, not necessarily the time used will decrease. But general speaking, if the work load is lighter, the smaller number of threads should be used. The resources used for maintaining the threads is larger when the number of threads is larger. So, if the work load is small, then parallelization using a large number of threads is not efficient because the resources used offsets the time saved. This can be observed in Figure 8. and Figure 9. where for outer loop parallelization, when the matrix size is 50*50, the time used is minimum when the number of threads is 4, whereas when the matrix size is 500*500, the time used is minimum when the number of threads is 16.

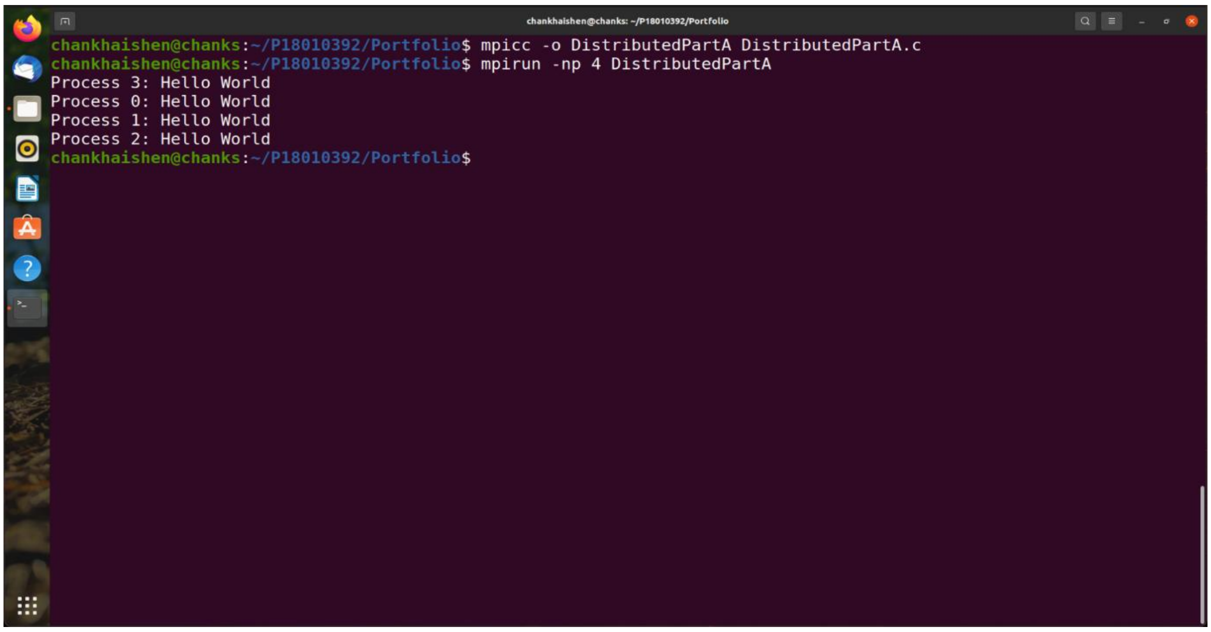
Distributed Programming Portfolio Element

Part A

“DistributedPartA.c” prints hello world from each process. Figure 10. is the screenshot of “DistributedPartA.c” running.

DistributedPartA.c

```
#include <stdio.h>

#include <mpi.h>



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

        MPI_Init(&argc, &argv);



        int rank;

        MPI_Comm_rank(MPI_COMM_WORLD, &rank);



        printf("Process %d: Hello World\n", rank);



        MPI_Finalize();



        return 0;

}
```



```
chankhaishen@chanks:~/P18010392/Portfolio$ mpicc -o DistributedPartA DistributedPartA.c
chankhaishen@chanks:~/P18010392/Portfolio$ mpirun -np 4 DistributedPartA
Process 3: Hello World
Process 0: Hello World
Process 1: Hello World
Process 2: Hello World
chankhaishen@chanks:~/P18010392/Portfolio$
```

Figure 10 Screenshot of DistributedPartA.c running

Part B

“DistributedPartB_a.c” is answering a), which all slave processes send message to the master process to be printed. “DistributedPartB_b.c” is answering b), which each slave process sends a different message. Figure 11. and Figure 12. are screenshots of “DistributedPartB_a.c” and “DistributedPartB_b.c” running.

DistributedPartB_a.c

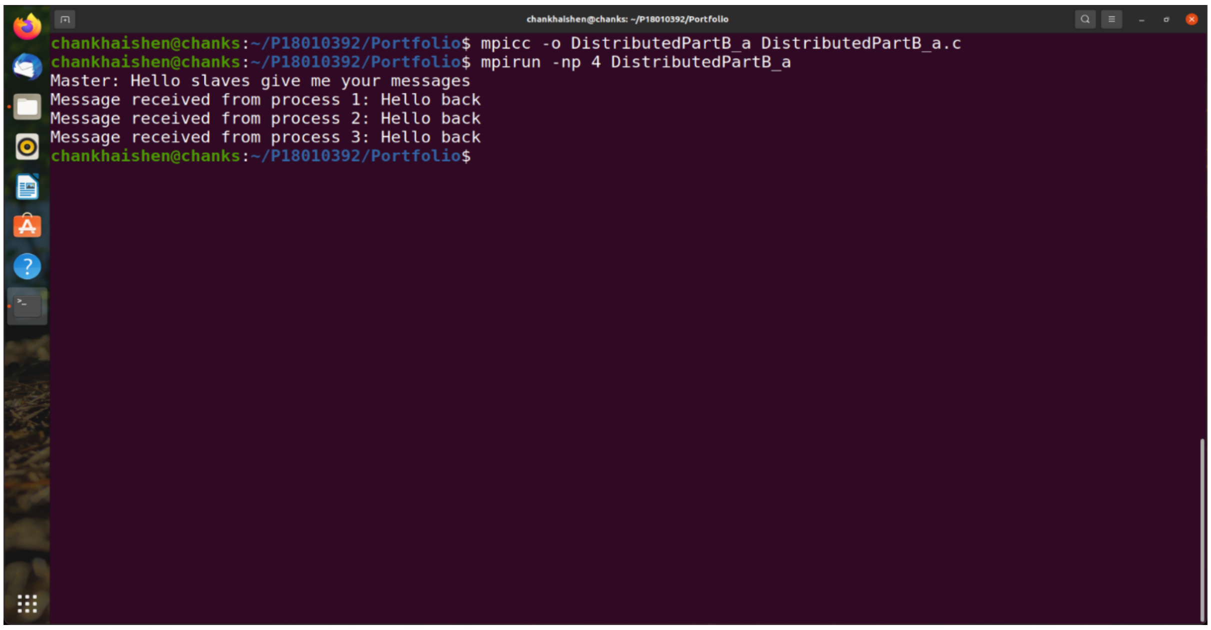
```
#include <stdio.h>

#include <mpi.h>



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

        MPI_Init(&argc, &argv);



        int rank;
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);


        int MESSAGE_LENGTH = 20;


        if (rank == 0) {
                printf("Master: Hello slaves give me your messages\n");


                MPI_Request requests[3];
                MPI_Status statuses[3];
                char message1[MESSAGE_LENGTH], message2[MESSAGE_LENGTH],
message3[MESSAGE_LENGTH];


                MPI_Irecv(message1, MESSAGE_LENGTH, MPI_CHAR, 1, 99,
MPI_COMM_WORLD, &requests[0]);
                MPI_Irecv(message2, MESSAGE_LENGTH, MPI_CHAR, 2, 99,
MPI_COMM_WORLD, &requests[1]);
                MPI_Irecv(message3, MESSAGE_LENGTH, MPI_CHAR, 3, 99,
MPI_COMM_WORLD, &requests[2]);


                MPI_Waitall(3, requests, statuses);


                printf("Message received from process 1: %s\n", message1);
                printf("Message received from process 2: %s\n", message2);
                printf("Message received from process 3: %s\n", message3);
```

```
	}
	else {
		MPI_Request request;
		char message[] = "Hello back";
		MPI_Isend(message, MESSAGE_LENGTH, MPI_CHAR, 0, 99,
MPI_COMM_WORLD, &request);
	}

	MPI_Finalize();

	return 0;
}
```

DistributedPartB_b.c

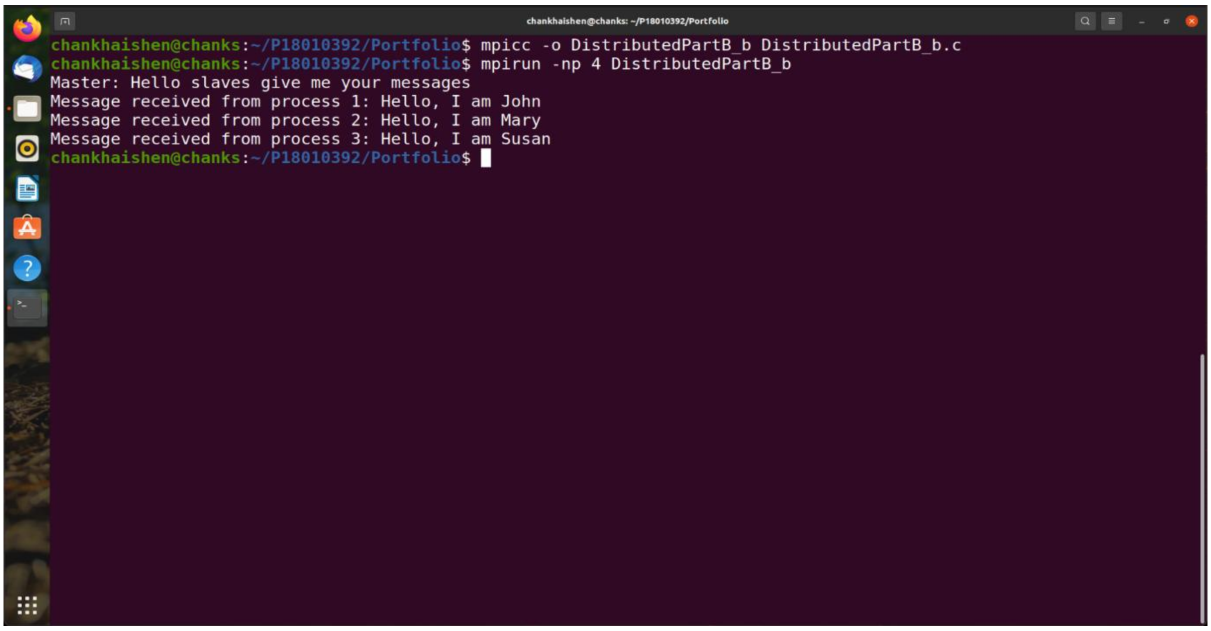
```
#include <stdio.h>
#include <mpi.h>

int main (int argc, char **argv) {
	MPI_Init(&argc, &argv);

	int rank;
	MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
        int MESSAGE_LENGTH = 20;

        if (rank == 0) {
                printf("Master: Hello slaves give me your messages\n");

                MPI_Request requests[3];
                MPI_Status statuses[3];
                char message1[MESSAGE_LENGTH], message2[MESSAGE_LENGTH],
message3[MESSAGE_LENGTH];

                MPI_Irecv(message1, MESSAGE_LENGTH, MPI_CHAR, 1, 99,
MPI_COMM_WORLD, &requests[0]);
                MPI_Irecv(message2, MESSAGE_LENGTH, MPI_CHAR, 2, 99,
MPI_COMM_WORLD, &requests[1]);
                MPI_Irecv(message3, MESSAGE_LENGTH, MPI_CHAR, 3, 99,
MPI_COMM_WORLD, &requests[2]);

                MPI_Waitall(3, requests, statuses);

                printf("Message received from process 1: %s\n", message1);
                printf("Message received from process 2: %s\n", message2);
                printf("Message received from process 3: %s\n", message3);
        }
        else if (rank == 1) {
                MPI_Request request;

                char message[] = "Hello, I am John";

                MPI_Isend(message, MESSAGE_LENGTH, MPI_CHAR, 0, 99,
MPI_COMM_WORLD, &request);

        }

        else if (rank == 2) {

                MPI_Request request;

                char message[] = "Hello, I am Mary";

                MPI_Isend(message, MESSAGE_LENGTH, MPI_CHAR, 0, 99,
MPI_COMM_WORLD, &request);

        }

        else {

                MPI_Request request;

                char message[] = "Hello, I am Susan";

                MPI_Isend(message, MESSAGE_LENGTH, MPI_CHAR, 0, 99,
MPI_COMM_WORLD, &request);

        }


        MPI_Finalize();


        return 0;

}
```



```
chankhaishen@chanks:~/P18010392/Portfolio$ mpicc -o DistributedPartB_a DistributedPartB_a.c
chankhaishen@chanks:~/P18010392/Portfolio$ mpirun -np 4 DistributedPartB_a
Master: Hello slaves give me your messages
Message received from process 1: Hello back
Message received from process 2: Hello back
Message received from process 3: Hello back
chankhaishen@chanks:~/P18010392/Portfolio$
```

Figure 11 Screenshot of DistributedPartB_a.c running



```
chankhaishen@chanks:~/P18010392/Portfolio$ mpicc -o DistributedPartB_b DistributedPartB_b.c
chankhaishen@chanks:~/P18010392/Portfolio$ mpirun -np 4 DistributedPartB_b
Master: Hello slaves give me your messages
Message received from process 1: Hello, I am John
Message received from process 2: Hello, I am Mary
Message received from process 3: Hello, I am Susan
chankhaishen@chanks:~/P18010392/Portfolio$
```

Figure 12 Screenshot of DistributedPartB_b.c running

Part C

“DistributedPartC_a.c” is answering the first part of the question, which the master process sends message to each process with tag 100. “DistributedPartC_b.c” is answering the second part of the question, which the slave processes wait for message with tag 101. Figure 13. is the screenshot of “DistributedPartC_a.c” running. Figure 14. is the screenshot of “DistributedPartC_b.c” hanging.

DistributedPartC_a.c

```
#include <stdio.h>

#include <mpi.h>



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

        MPI_Init(&argc, &argv);



        int rank;

        MPI_Comm_rank(MPI_COMM_WORLD, &rank);



        int MESSAGE_LENGTH = 20;



        if (rank == 0) {

                printf("Master: Hello slaves receive my message\n");



                MPI_Request requests[3];

                char message[] = "Hello";



                MPI_Isend(message, MESSAGE_LENGTH, MPI_CHAR, 1, 100,
MPI_COMM_WORLD, &requests[0]);

                MPI_Isend(message, MESSAGE_LENGTH, MPI_CHAR, 2, 100,
MPI_COMM_WORLD, &requests[1]);
```

```
                    MPI_Isend(message, MESSAGE_LENGTH, MPI_CHAR, 3, 100,
MPI_COMM_WORLD, &requests[2]);

          }

          else {

                    MPI_Request request;

                    MPI_Status status;

                    char message[MESSAGE_LENGTH];



                    MPI_Irecv(message, MESSAGE_LENGTH, MPI_CHAR, 0, 100,
MPI_COMM_WORLD, &request);

                    MPI_Wait(&request, &status);

                    printf("Message received by process %d: %s\n", rank, message);

          }



          MPI_Finalize();



          return 0;

}
```

DistributedPartC_b.c

```
#include <stdio.h>

#include <mpi.h>
```

```
int main (int argc, char **argv) {

        MPI_Init(&argc, &argv);


        int rank;

        MPI_Comm_rank(MPI_COMM_WORLD, &rank);


        int MESSAGE_LENGTH = 20;


        if (rank == 0) {

                printf("Master: Hello slaves receive my message\n");


                MPI_Request requests[3];

                char message[] = "Hello";


                MPI_Isend(message, MESSAGE_LENGTH, MPI_CHAR, 1, 100,
MPI_COMM_WORLD, &requests[0]);

                MPI_Isend(message, MESSAGE_LENGTH, MPI_CHAR, 2, 100,
MPI_COMM_WORLD, &requests[1]);

                MPI_Isend(message, MESSAGE_LENGTH, MPI_CHAR, 3, 100,
MPI_COMM_WORLD, &requests[2]);

        }

        else {

                MPI_Request request;

                MPI_Status status;
```

char message[MESSAGE_LENGTH];

MPI_Irecv(message, MESSAGE_LENGTH, MPI_CHAR, 0, 101, MPI_COMM_WORLD, &request);

MPI_Wait(&request, &status);

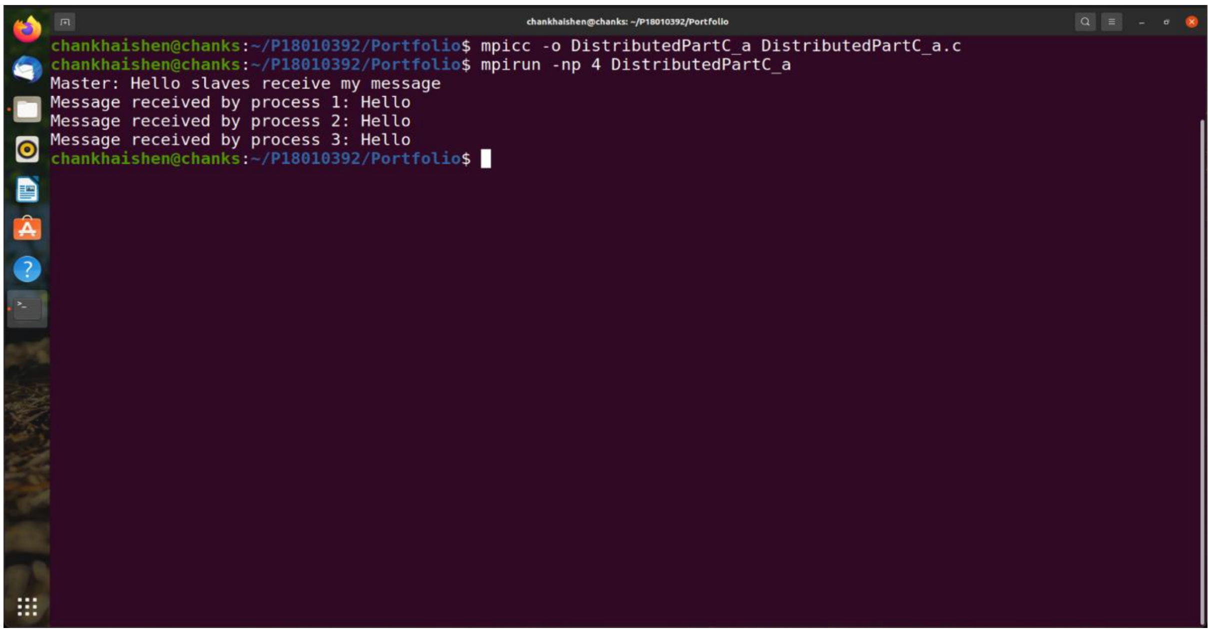
printf("Message received by process %d: %s\n", rank, message);

}

MPI_Finalize();

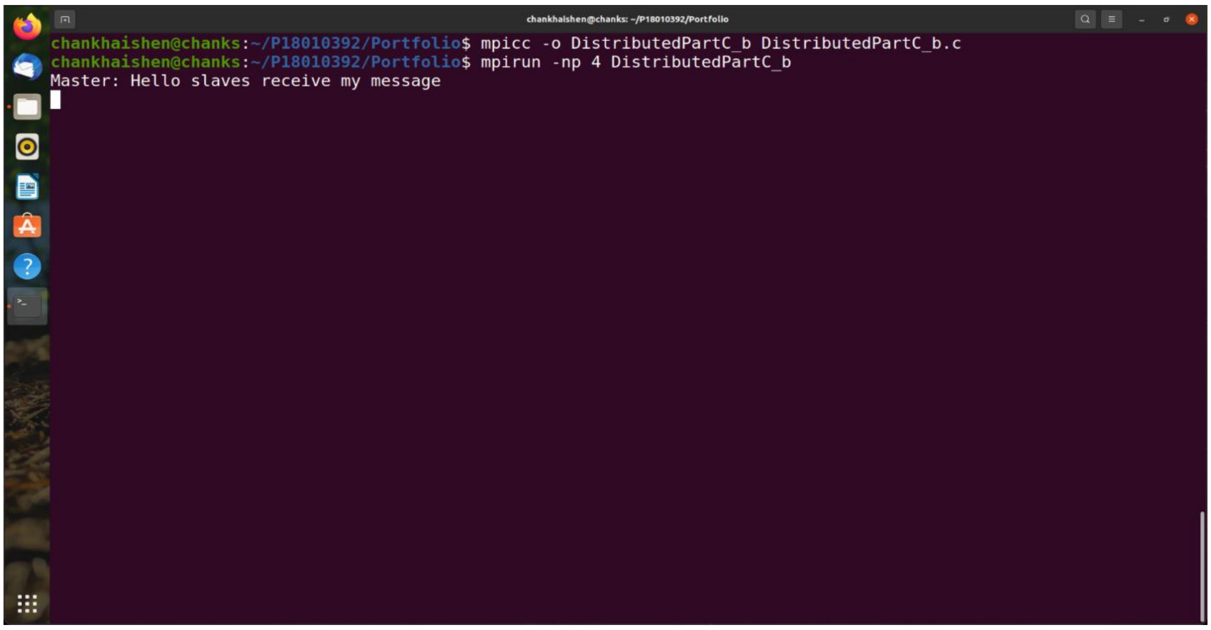
return 0;

}



```
chankhaishen@chanks:~/P18010392/Portfolio$ mpicc -o DistributedPartC_a DistributedPartC_a.c
chankhaishen@chanks:~/P18010392/Portfolio$ mpirun -np 4 DistributedPartC_a
Master: Hello slaves receive my message
Message received by process 1: Hello
Message received by process 2: Hello
Message received by process 3: Hello
chankhaishen@chanks:~/P18010392/Portfolio$
```

Figure 13 Screenshot of DistributedPartC_a.c running



```
chankhaishen@chanks:~/P18010392/Portfolio$ mpicc -o DistributedPartC_b DistributedPartC_b.c
chankhaishen@chanks:~/P18010392/Portfolio$ mpirun -np 4 DistributedPartC_b
Master: Hello slaves receive my message
```

Figure 14 Screenshot of DistributedPartC_b.c hanging

I think that when the slave process are waiting for tag"101" it hangs because MPI tag is a reference for processes to identify the message to be received, so when there is no message of tag "101" is sent out from the master process, the slave process waits for it forever until a message of tag "101" reaches.

References

avsadityavardhan. (n.d.). *time.h header file in C with Examples*. Retrieved from Geeks for geeks: https://www.geeksforgeeks.org/time-h-header-file-in-c-with-examples/

(Used in recording execution time in Task 2 Parallel PartB)

Gillis, A. S., & Bigelow, S. J. (July, 2022). *message passing interface (MPI)*. Retrieved from https://www.techtarget.com/searchenterprisedesktop/definition/message-passing-interface-MPI

Heller, M. (16 September, 2022). *What is CUDA? Parallel programming for GPUs*. Retrieved from Info World: https://www.infoworld.com/article/3299703/what-is-cuda-parallel-programming-for-gpus.amp.html

Varman, L. (22 February, 2023). *Parallel Programming With OpenMP In C++ - Matrix Multiplication Example*. Retrieved from C# Corner: https://www.c-sharpcorner.com/article/parallel-programming-with-openmp-in-cpp-matrix-multiplication-example/

(Used in matrix multiplication in Task 2 Parallel Part C)

(Womack, M., Wang, A., Flynn, P., Yi, X., & Yan, Y. (2022). *Introduction of OpenMP*. Retrieved from https://passlab.github.io/OpenMPProgrammingBook/openmp_c/1_IntroductionOfOpenMP.html