An application of distributed memory parallel programming techniques for the implementation of a combinatorial optimization problem, the wandering salesman problem

High Performance Technical Computing 2022-2023

Purin Tanirat s388240

Supervised by Dr. Irene Moulitsas

School of Aerospace Transport and Manufacturing, Cranfield University (MK43 0AL, UK)

Submitted on Monday 06/02/2023

Table of Contents

Table of Figures

Table of Equations

Introduction

High-performance computing (HPC) has become increasingly important in solving complex problems (see Ref. [1] p.6) in various fields such as finance, weather forecasting, and bioinformatics. The wandering salesman problem (WSP), where the shortest path to visit a given set of cities without returning to the starting city must be found, is one such problem that can benefit from HPC. The WSP is considered an NP-hard problem, and its time complexity increases exponentially with the number of cities, making it difficult to solve for large instances using traditional algorithms.

In this project, the utilization of the power of HPC to solve the WSP more efficiently is aimed to be achieved. Both serial and parallel algorithms for the WSP will be implemented and their performance will be compared. The parallel algorithms will be implemented using the MPI library, allowing for the computation to be distributed among multiple processors (see Ref [2] p. 94-103). To optimize the performance of the parallel algorithms, different communication patterns and techniques will be explored.

Methodology

WSP and Branch and Bound Solution

The Wandering Salesman Problem (WSP) is a computationally challenging NP-Hard problem that involves finding the shortest route for a traveling salesman (see Ref. [3] p.5) to visit a set of cities exactly once. A solution is to evaluate all possible routes using a tree. An example of a 4-city problem is in Figure 1.

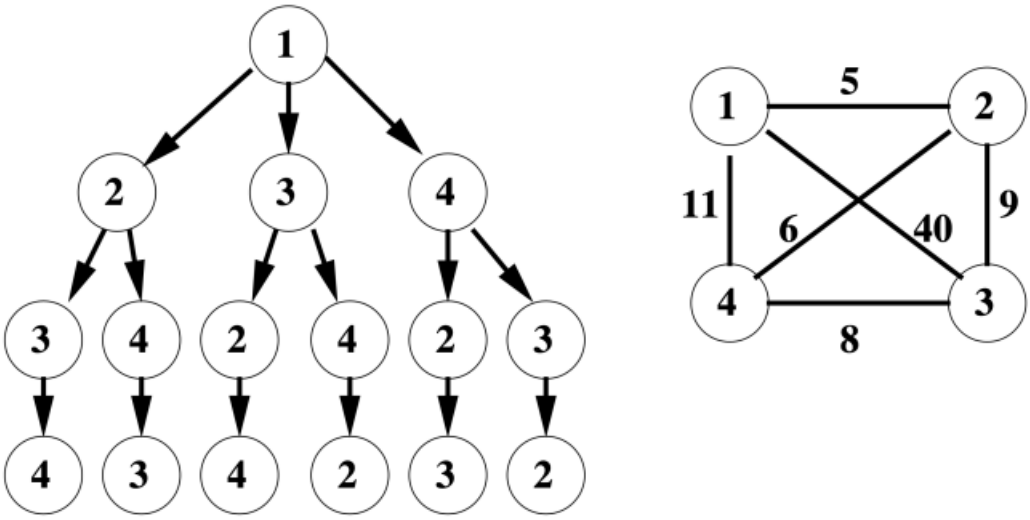

Figure 1 A Four-City Wandering Salesman Problem. (Left) A tree showing all unique paths through the four cities. (Right) A graph illustrating the distances between the cities.

A more efficient way to traverse the tree is using a recursive approach with a "branch and bound" method (see Ref. [4] p. 429-457). The approach uses a summary of the path distance (bound) to avoid repeating earlier parts of the route. Before exploring further, the current bound is compared with the previous best path, and if it's larger, exploration stops.

Program Structure and Coding Format

The program has three versions: a serial code, parallel code v1, and parallel code v2 with processor sharing. The program is written in C with MPI library for parallelism and divided into functions for better readability and maintainability. The use of global variables and dynamically allocated arrays is common across all versions of the code.

Global variables were used as the primary means of updating calculations, particularly in the ***branch_and_bound function***. This allows for easy access to the data from any function without having to pass the information as arguments.

Memory allocation is a useful tool not only for allocating memory, but also for declaring arrays in global variables. This is because the size of the array can be determined at runtime, making the program more flexible in terms of array size specifications, while allowing all functions to access the global array variable.

In parallelization, file read/write operations are done only by the ROOT processor to avoid file access issues. All versions have a function to save results in a CSV file for easy analysis and comparison in a spreadsheet.

Serial Algorithm for WSP (Serial Code)

The serial code for solving the WSP is designed to compute the solution without parallelization, using a sequential Branch-and-Bound approach. The overall flow of the algorithm is depicted in Figure 2. The algorithm starts by reading city data from a file and storing it in the ***n*** variable and ***dist*** array. Then, the program goes to the ***branch_and_bound function***.

The ***branch_and_bound function*** is a recursive function that implements the Branch-and-Bound algorithm for the WSP. The function takes 4 arguments:

1. ***path***: an array that stores the current order of visiting cities.
2. ***path_bound***: an integer that stores the distance of the current path.
3. ***visited***: an array that stores the status of each city (visited or not visited).
4. ***level***: an integer that keeps track of the current level or position in the path.

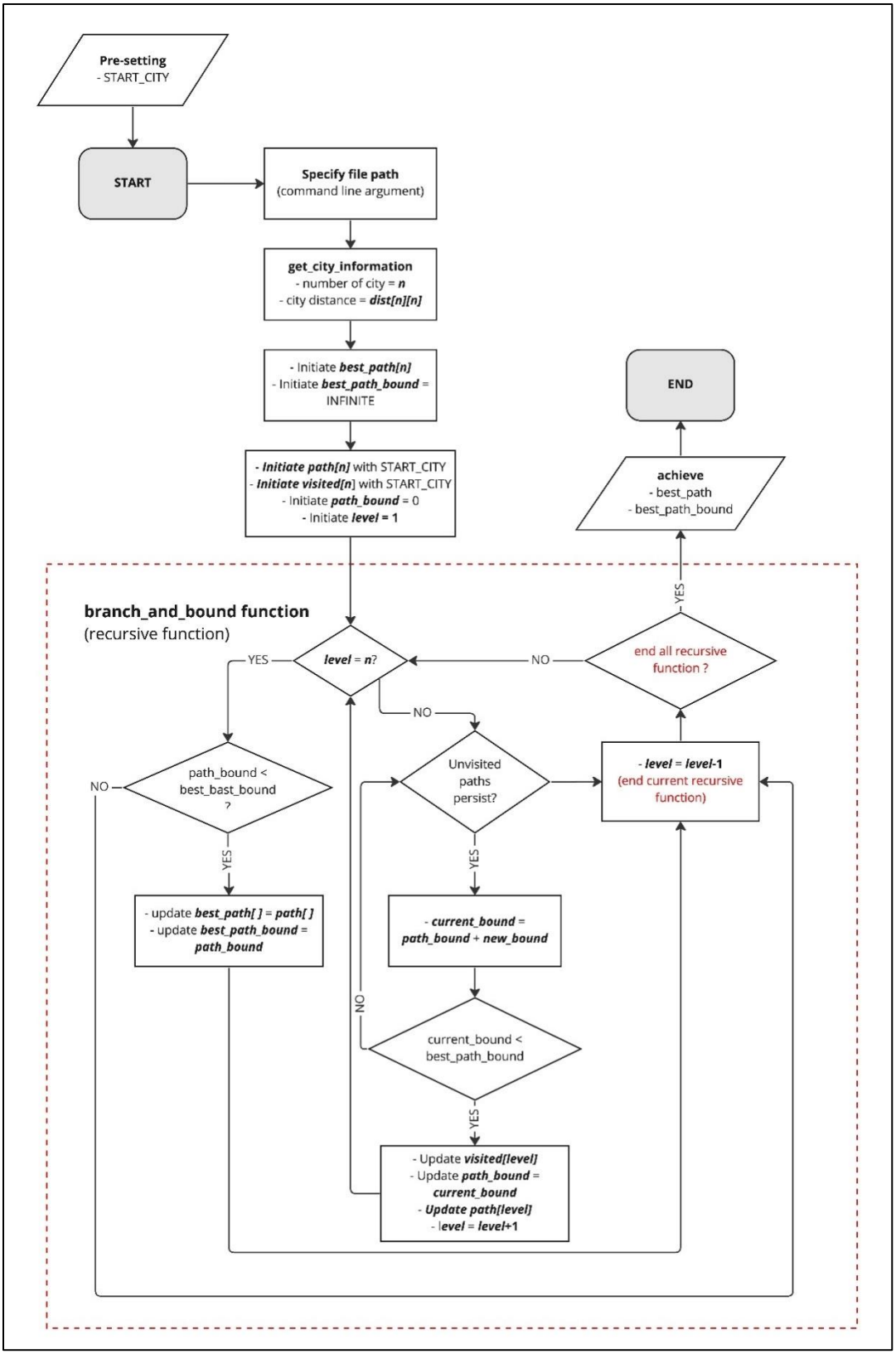

Figure 2 Serial Algorithm for WSP (Serial Code)

The function works as follows:

- If ***level*** is equal to ***n*** (number of cities), it means that all cities have been visited, so the function compares the current ***path_bound*** with the best path found so far, stored in ***best_path_bound***.
- If ***path_bound*** is smaller, it updates ***best_path_bound*** with ***path_bound*** and ***best_path*** with the current ***path***.
- If ***level*** is less than ***n***, the function loops through all cities.
- If a city has not been visited, it marks it as ***visited***, adds its distance to ***path_bound*** and calls itself recursively with the updated values of ***path***, ***path_bound***, ***visited***, and **level + 1**.
- If the new ***path_bound*** is greater than or equal to ***best_path_bound***, the function returns without further exploration of this path.
- After the recursion ends, the function returns to the caller with the final value of ***best_path*** and ***best_path_bound***.

Parallel Algorithm for WSP (Parallel Code)

The parallel algorithm for solving the wandering salesman problem (WSP) consists of two parallel codes, v1 and v2.

Parallel code v1 divides the WSP into smaller sub-problems and assigns each to a separate processor. Each processor computes the shortest path for their sub-problem, and the results are combined and compared once all processors finish. The algorithm flow is shown in Figure 3.

The algorithm starts by reading city data into ***n*** variable and ***dist*** array, with the ROOT processor responsible for reading and distributing the data through Sending communication. The ***initiation_path function*** is then performed in parallel by all processors, which is similar to the ***branch_and_bound function*** in concept. This function creates ***init_path*** array and ***init_path_bound*** arrays. Then, the ***branch_and_bound function*** is performed in parallel by each processor. However, the arrays ***best_path*** and ***best_path_bound*** have one extra dimension compared to the serial algorithm, as the parallel algorithm stores the results of each processor individually. After that, the Gathering communication is performed to let the ROOT collect all results.

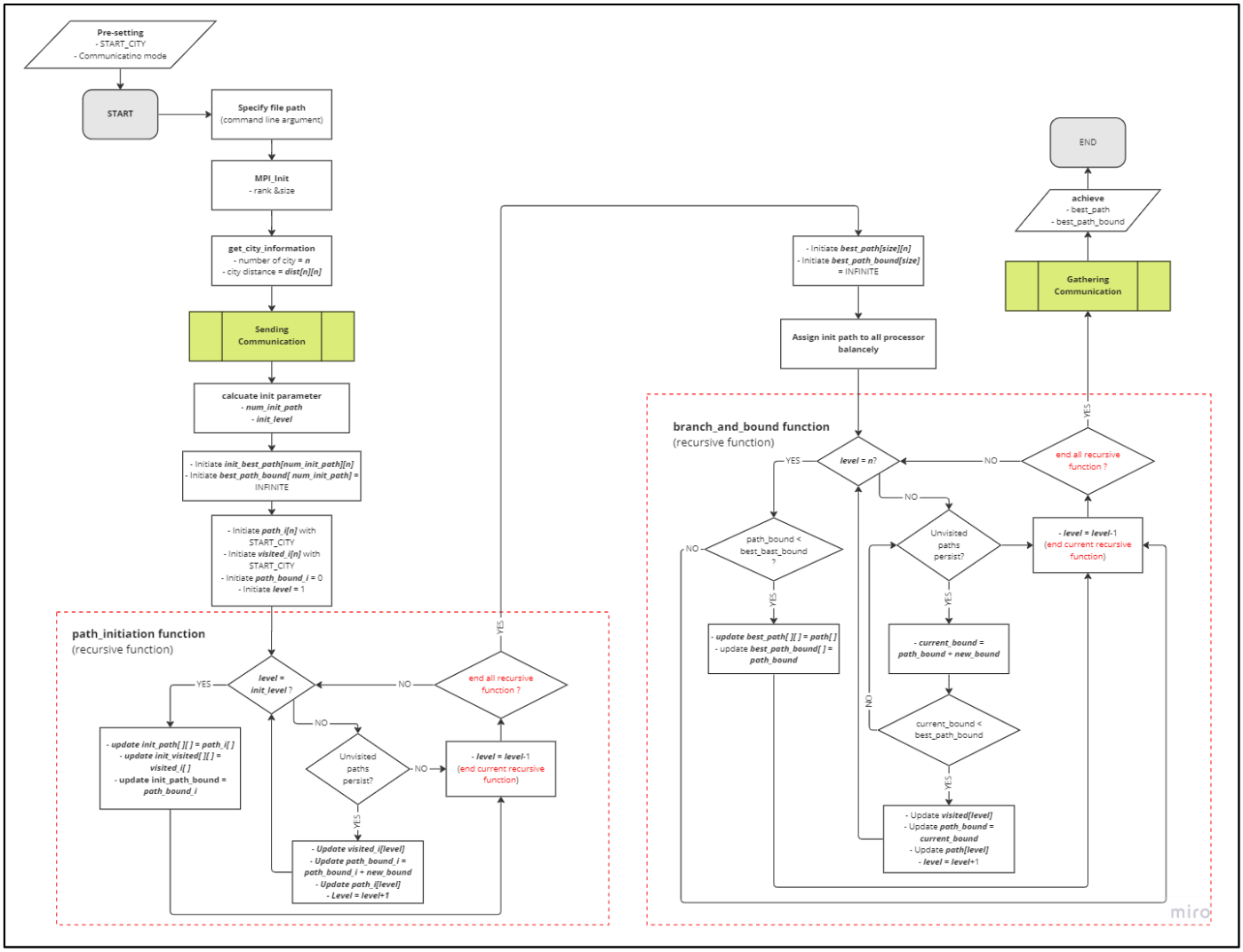

Figure 3 Parallel code v1 for WSP

Parallel code v2 is similar to parallel code v1 but includes a mechanism for sharing the latest best results among processors to avoid unnecessary evaluations. The flow of the algorithm is shown in Figure 4.

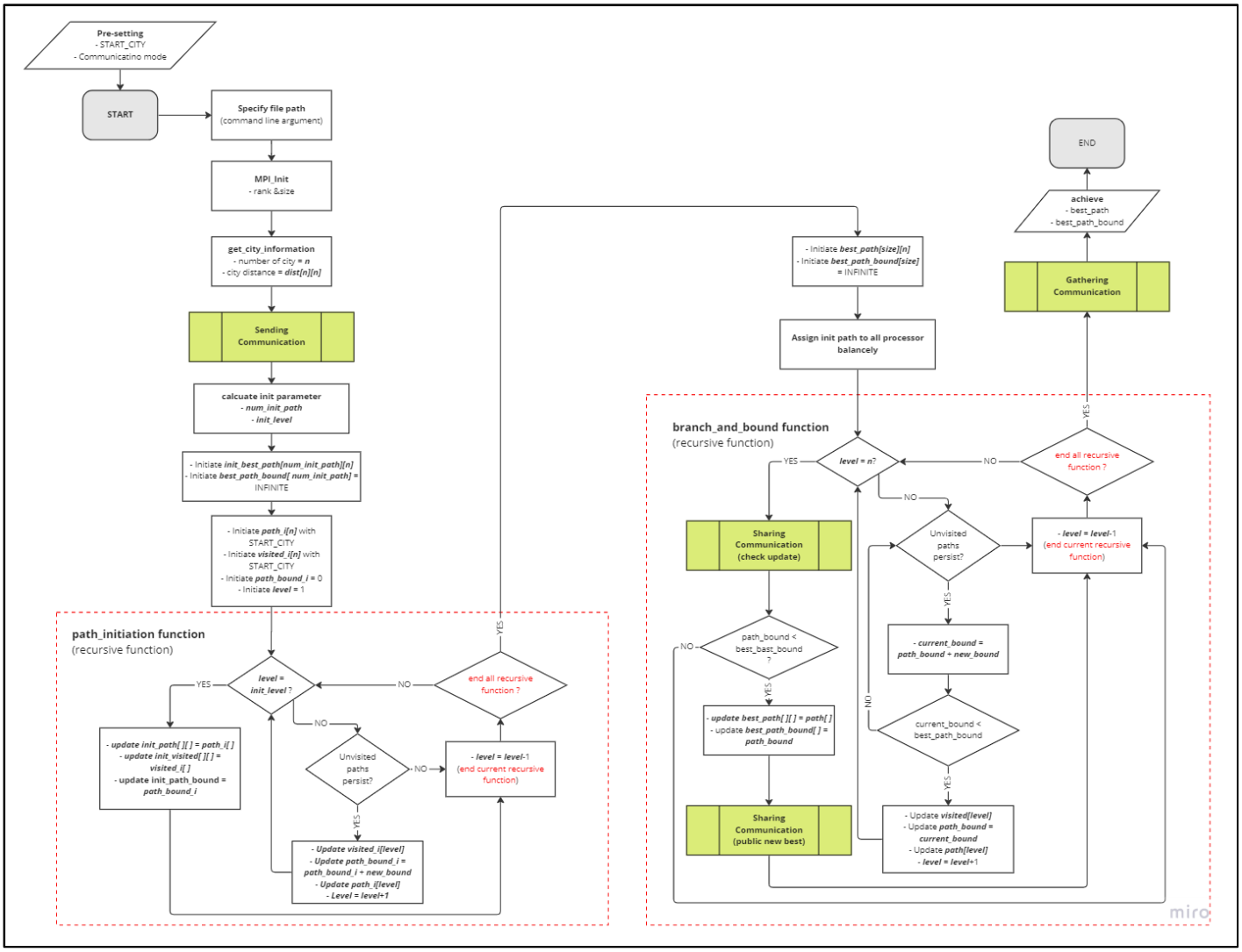

Figure 4 Parallel code v2 for WSP

This program operates similarly to parallel code v1. The difference is that, when a processor finds a new best result (***best_path_bound***), it shares it with other processors to update the best result for the entire system with Sharing communication. The sharing occurs within the ***branch_and_bound function***.

Parallel communication

The parallel code implements communication through the MPI library. The communication in the code is divided into three main types:

1. Sending communications: This type of communication is used to send city information, such as the distance between each city, from the ROOT processor to other processors. Both parallel code v1 and v2 implement this type of communication. There are four types of communications tested in this project:

    a. ***MPI_Bcast***: This communication is broadcasted to all processors.
    b. ***MPI_Ibcast***: This is a non-blocking communication that is broadcasted to all processors. Non-ROOT processors use ***MPI_Test*** to ensure they receive the information before moving on to the next step.
    c. ***MPI_Send*** and ***MPI_Recv***: ROOT processor uses ***MPI_Send*** while other processors use ***MPI_Recv***.
    d. ***MPI_Isend*** and ***MPI_Irecv***: ROOT processor uses ***MPI_Isend*** while other processors use ***MPI_Irecv***. This is a non-blocking communication and non-ROOT processors use ***MPI_Wait*** to ensure they receive the information before moving on to the next step.

2. Gathering communications: This communication is used by the ROOT processor to receive results from other processors. Both parallel code v1 and v2 implement this communication. To ensure that the ROOT processor receives all the data before proceeding, only blocking communications are used in this step. Two types of communication were tested in this project:

    a. ***MPI_Allgather***: This communication is implemented by all processors.
    b. ***MPI_Send*** and ***MPI_Recv***: ***MPI_Send*** is used by the ROOT processor, while ***MPI_Recv*** is used by other processors.

3. Sharing Communications: This communication is only implemented in parallel code v2 to share the best result (***best_path_bound***) while performing ***branch_and_bound function*** in each processor. ***MPI_Isend*** and ***MPI_Irecv*** were used in all processors as a non-blocking communication. This was selected because it allows the program to continue with other steps even when there is no data from other processors and each processor can communicate individually, without performing collective communication. However, ***MPI_Test*** was also used to check the status of the request message during the step of updating the new ***best_bound*** from other processors. To reduce the number of messages

sent, ***MPI_Cancel*** was called before sharing a new best bound. This ensures that other processors always receive the latest update every time they check the message from other processors.

Program analysis

Communication performance

The communication performance will be evaluated by measuring the time consumed by each communication pattern. The ***MPI_Wtime*** function will be used to measure the time before and after executing the communication function. To accurately evaluate the communication performance, synchronization is crucial. Thus, ***MPI_Barrier*** is used before measuring the time.

Verification of Correctness

To ensure the accuracy of the parallel solution, the results of the serial code were compared with the parallel solution. Additionally, the locations of the cities were also obtained to visualize the result path.

Scalability and Parallel Speed-up

Parallel speed-up of the program are evaluated by determining the ratio of execution time of the parallel algorithm on a single processor ($Time_{single}$) to the execution time of the parallel algorithm on multiple processors ($Time_{multiple}$). This is known as the $SpeedUp$ and can be calculated using Equation 1.

$$SpeedUp = \frac{Time_{single}}{Time_{multiple}}$$ *Equation 1 Parallel-Speedup*

The scalability of the program will be assessed by increasing the number of processor cores and observing its effect on the parallel $SpeedUp$. In addition, the parallel efficiency ($ParallelEfficiency$) will be calculated using Equation 2.

$$ParallelEfficiency = \frac{SpeedUp}{P}$$ *Equation 2 Parallel-Efficiency*

Where P represents the number of processors

Problem size

The size of the problem, represented by the number of cities, is crucial in this NP-Hard problem. The execution time will be compared between the serial code and parallel code for different sizes of the number of cities.

Cresent

Cresent (Cranfield Educational Scientific Computing Environment for Teaching), also known as Cranfield HPC, is a high-performance computing system that can be used to parallelize and optimize algorithms. In the context of this project, Cresent was used to execute all of the project program.

Result

The results of the communication cost evaluation are shown in Figure 5 and Figure 6. The number of communication patterns for sending distance data to all processors and gathering results from all processors were compared. The communication time was measured in the ROOT processor using 4 different groups of processors: 4 processors, 16 processors, 32 processors, and 64 processors. To ensure accuracy, the average of 10 tests for sending communication and 20 tests for gathering communication were analyzed, rather than just one test. Each communication pattern was synchronized before the performance test to specifically evaluate the communication performance.

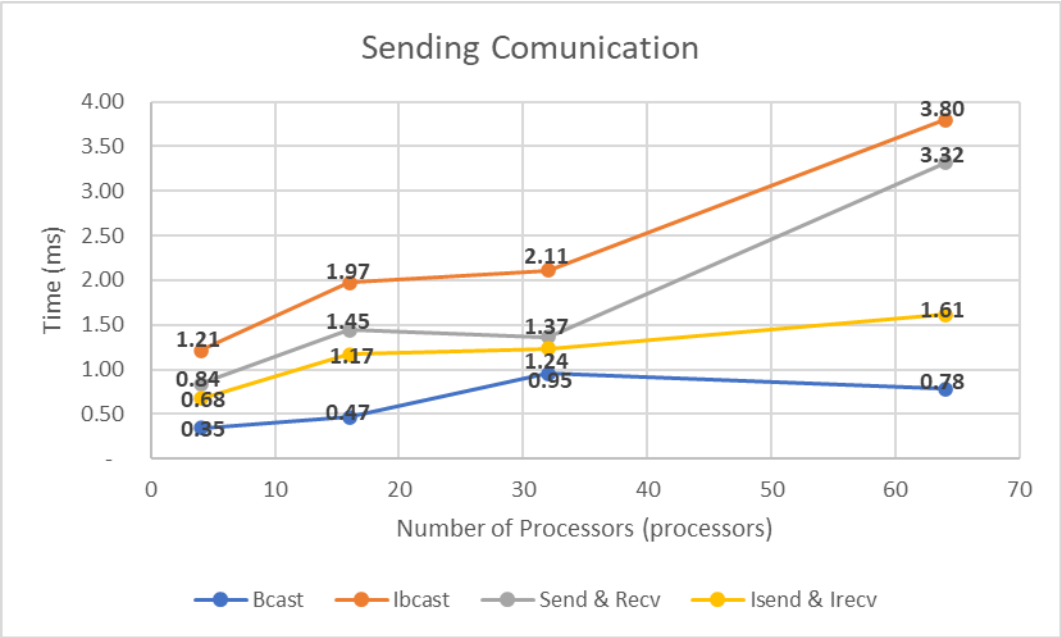

Figure 5 The sending communication cost for 4 different communication patterns

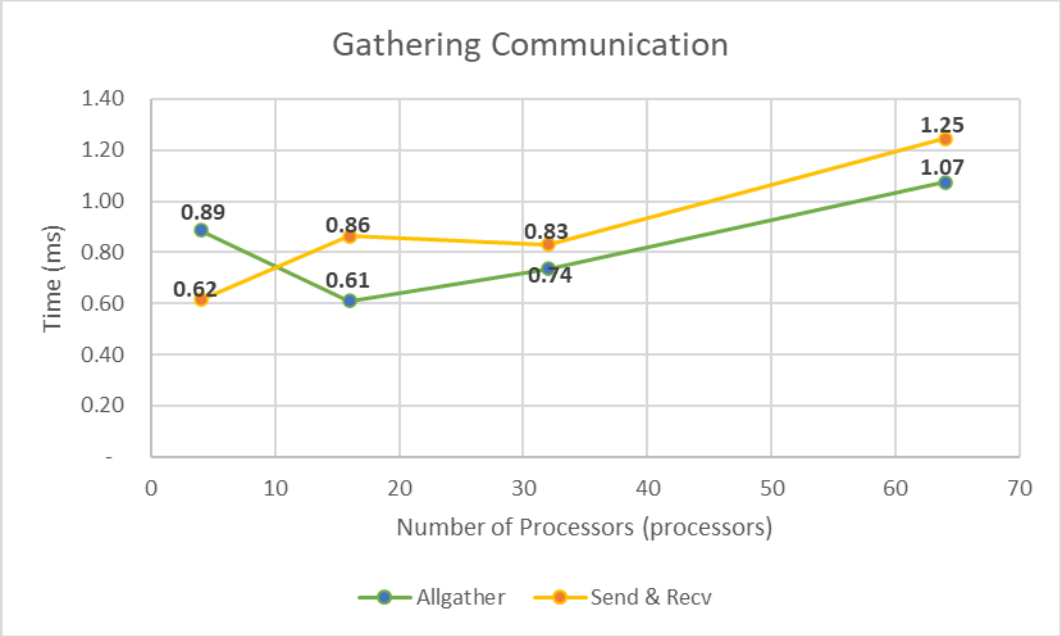

Figure 6 The gathering communication cost for 2 different communication patterns

The results of the shortest path for the serial code, parallel code, and parallel code with processor sharing are all the same. One visual representation of the best path starting at city 10 can be seen in Figure 7, which shows the best path as 10-1-4-14-15-9-6-11-13-5-8-3-16-2-17-7-12 with reference to the city locations.

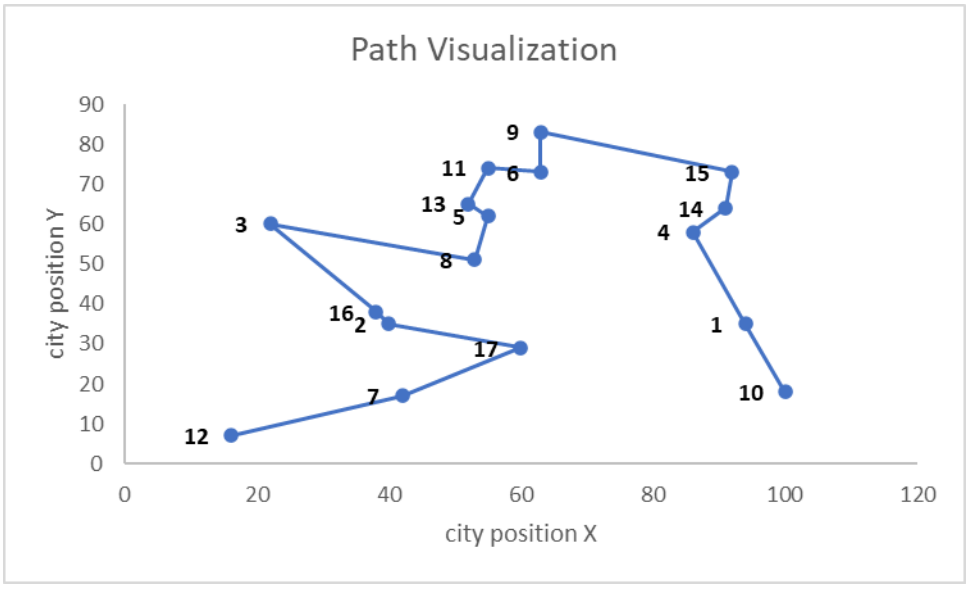

Figure 7 The shortest path starting from city 10

However, the shortest path depends on the starting city. The bound (distance) of the path and the computing time for all types of code are shown in Figure 8.

Start city	The Shortest Path	Path Bound	computing time (s)		
			Serial	Parallel v1	Parallel v2
1	1 10 4 14 15 9 6 11 13 5 8 3 16 2 17 7 12	278	261.03	57.15	25.93
2	2 16 3 8 5 13 11 6 9 15 14 4 1 10 17 7 12	281	201.24	36.39	10.88
3	3 16 2 12 7 17 8 5 13 11 6 9 15 14 4 1 10	267	82.09	17.16	6.72
4	4 14 15 9 6 11 13 5 8 3 16 2 12 7 17 1 10	286	229.09	46.65	16.66
5	5 13 11 6 9 15 14 4 1 10 17 8 3 16 2 7 12	290	187.82	31.19	8.59
6	6 9 11 13 5 8 3 16 2 12 7 17 10 1 4 14 15	291	194.84	34.39	12.94
7	7 12 17 2 16 3 8 5 13 11 6 9 15 14 4 1 10	288	272.95	53.14	19.79
8	8 5 13 11 6 9 15 14 4 1 10 17 7 12 2 16 3	285	136.17	20.83	8.92
9	9 6 11 13 5 8 3 16 2 12 7 17 10 1 4 14 15	287	209.20	31.14	13.05
10	10 1 4 14 15 9 6 11 13 5 8 3 16 2 17 7 12	260	44.74	25.71	13.54
11	11 13 5 6 9 15 14 4 1 10 17 8 3 16 2 7 12	295	197.79	27.22	15.24
12	12 7 17 2 16 3 8 5 13 11 6 9 15 14 4 1 10	260	78.84	18.14	8.26
13	13 17 7 12 2 16 3 8 5 11 6 9 15 14 4 1 10	311	193.62	27.12	8.06
14	14 17 7 12 2 16 3 8 5 13 11 6 9 15 4 1 10	322	248.98	48.55	23.99
15	15 14 4 1 10 17 8 5 6 9 11 13 3 16 2 7 12	282	277.43	45.79	19.38
16	16 2 3 8 5 13 11 6 9 15 14 4 1 10 17 7 12	284	160.74	31.32	11.16
17	17 7 12 2 16 3 8 5 13 11 6 9 15 14 4 1 10	276	121.81	27.40	12.29

Figure 8 the shortest path and bound for each starting city

Figure 9 and Figure 10 display the parallel speed up with varying number of processors in both table and graph format, respectively. Figure 11 also shows the result of parallel efficiency, using the average of 10 simulations for each number of processor and starting city 10.

number of processors	Ideal (S=N)	Parallel code speed-up	Parallel code V2 speed-up	number of access branch and bound function					
				parallel code v1		parallel code v2		% reduce V1 -> v2	
				max access	min access	max access	min access	reduce max	reduce min
1	1.00	1.00	1.00	3.25E+08	3.25E+08	3.25E+08	3.25E+08	0.00%	0.00%
2	2.00	0.33	0.34	2.49E+08	2.49E+08	2.49E+08	2.49E+08	0.00%	0.00%
4	4.00	0.49	0.41	2.35E+08	2.35E+08	2.35E+08	2.35E+08	0.00%	0.00%
8	8.00	0.54	0.45	2.13E+08	1.87E+08	2.13E+08	1.67E+08	0.00%	10.49%
12	12.00	0.69	0.65	2.27E+08	1.20E+08	2.27E+08	9.88E+07	0.00%	17.74%
16	16.00	0.78	0.88	2.10E+08	1.20E+08	2.10E+08	7.49E+07	0.00%	37.58%
20	20.00	1.09	2.02	1.20E+08	5.34E+07	1.08E+08	4.29E+07	9.78%	19.73%
24	24.00	0.92	2.05	1.48E+08	5.15E+07	1.28E+08	3.29E+07	13.39%	36.01%
28	28.00	1.15	2.30	1.66E+08	4.86E+07	1.53E+08	2.90E+07	7.89%	40.35%
32	32.00	1.21	2.10	1.43E+08	5.41E+07	1.23E+08	2.30E+07	13.66%	57.45%
36	36.00	1.11	2.64	9.52E+07	4.90E+07	8.29E+07	1.94E+07	12.90%	60.44%
40	40.00	1.18	2.91	7.65E+07	4.00E+07	6.22E+07	2.85E+07	18.70%	28.83%
44	44.00	1.60	2.72	7.07E+07	4.67E+07	5.11E+07	1.96E+07	27.65%	58.01%
48	48.00	1.67	2.45	1.31E+08	4.60E+07	1.10E+08	2.48E+07	15.88%	46.20%
52	52.00	1.38	2.57	1.05E+08	4.95E+07	8.41E+07	2.08E+07	19.73%	58.00%
56	56.00	1.39	2.79	1.32E+08	3.96E+07	1.12E+08	1.72E+07	15.27%	56.65%
60	60.00	1.39	2.99	7.87E+07	3.80E+07	6.67E+07	1.57E+07	15.29%	58.62%
64	64.00	1.62	3.12	1.25E+08	2.82E+07	9.29E+07	1.24E+07	25.94%	55.85%

Figure 9 The table of parallel speed-up results for parallel code 1 and parallel code 2, based on a number of processors used

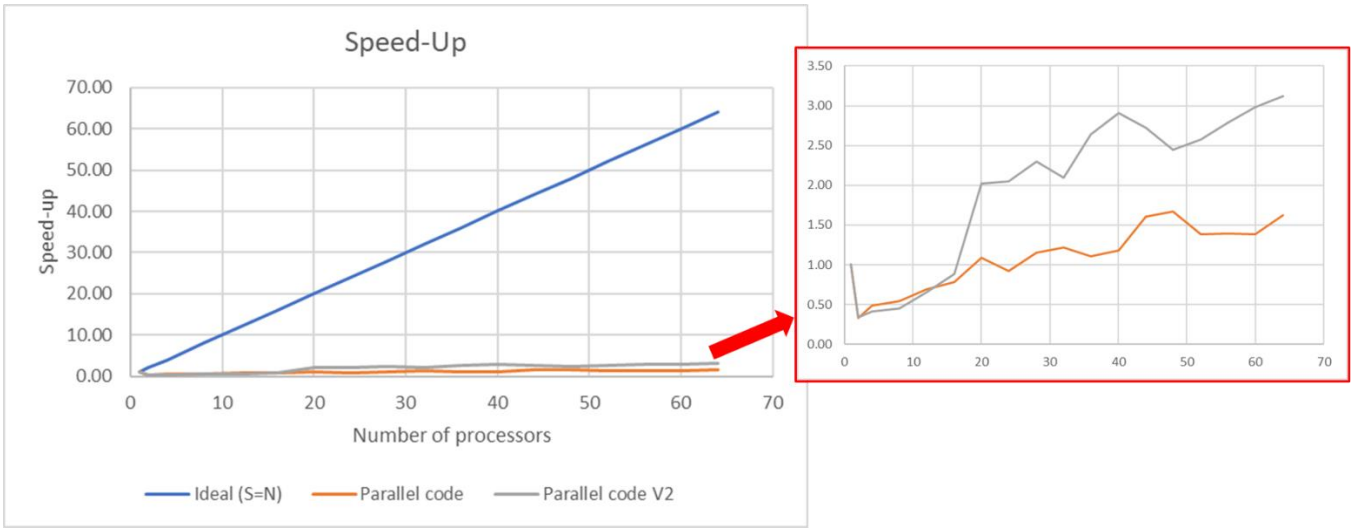

Figure 10 The graph of parallel speed-up results for parallel code 1 and parallel code 2, based on a number of processors used

number of processors	Parallel efficiency	
	parallel v1	parallel v2
1	100.16%	100.06%
2	16.69%	16.90%
4	12.21%	10.31%
8	6.81%	5.57%
12	5.78%	5.45%
16	4.90%	5.52%
20	5.45%	10.12%
24	3.84%	8.52%
28	4.12%	8.22%
32	3.79%	6.55%
36	3.07%	7.34%
40	2.94%	7.27%
44	3.65%	6.19%
48	3.48%	5.10%
52	2.65%	4.94%
56	2.48%	4.99%
60	2.31%	4.98%
64	2.53%	4.87%

Figure 11 The result of parallel efficiency

The performance of parallel code version 1 and version 2 was compared to the serial code in terms of computation time for various numbers of cities. The results are shown in Figure 12, where the starting city is city 10.

Code	number of processor	Execution time (s)		
		city size=5	city size=15	city size=17
Serial	N/A	0.00	1.58	41.22
Parallel code v1	4	0.00	3.97	80.31
	16	0.01	2.33	53.96
	32	0.02	1.60	34.31
	64	0.02	1.32	25.89
Parallel code v2	4	0.01	5.33	100.41
	16	0.01	1.94	53.26
	32	0.02	0.87	20.86
	64	0.03	0.88	15.57

Figure 12 The performance of parallel code with various numbers of cities, starting from city 10

Discussion and Analysis

The results of the communication time, as shown in Figure 5 and Figure 6, indicate that ***MPI_Bcast*** is the fastest sending communication at 0.78ms with 64 processors and ***MPI_Allgather*** is the best gathering communication at 1.07ms with 64 processors. However, the minimum total execution time, shown in Figure 8, is 6.72s, which is much larger than the sending and gathering communication time. The impact of sending and gathering communication on the total computing time is only about 0.012% and 0.016% respectively, making it clear that the sending and gathering communication patterns do not significantly affect the overall performance of the program.

The validity of the results can be seen in Figure 7. It can be noted that the shortest path connects the nearest cities. If the shortest path starts from city 10 and ends at city 12, the reverse path should be the same, which is clearly shown in Figure 8. Additionally, regarding WSP, the shortest path will vary based on the starting city, as demonstrated in Figure 8 where the best path starts from city 10 or city 12 with a total length of 260.

The execution times of parallel code v1 and parallel code v2 differ, as shown in Figure 9 and Figure 10. Sharing the ***best_path_bound*** among processors reduces the number of accesses to the ***branch_and_bound function***, leading to a boost in speed-up. According to Figure 9, when using 20 processors, the maximum number of accesses drops by about 9.78%, resulting in a doubling of speed-up from 1.09 times to 2.02 times. However, as depicted in Figure 11, the parallel efficiency trend decreases, meaning that adding more processors will not bring about a corresponding increase in speed-up.

Problem size is also a major factor in implementing parallel programming. A comparison of the execution time between serial code and parallel code is shown in Figure 12. It can be seen that parallel code executes slower than serial code for all numbers of cities when using low processing units. Even with more processors, if the problem size is small, the serial code can still be faster than some parallel code. For example, with 32 processing units and 15 cities, the serial code computes in 1.58s while both parallel code v1 and v2 are at 1.60s and 0.87s respectively. However, in the case of 17 cities with 4 processors, parallel code v1 is faster than parallel code v2, at around 80s and 100s respectively, while the serial code computes within 41.22s. In this case, if the number of processors is increased to 32 units, both parallel codes become faster than the serial code, with parallel code v2 being faster than parallel code v1.

Overall, the parallel algorithm attempts to evenly distribute the initiation paths among processors, leading to a balanced workload. However, there may be cases where the initiation paths cannot be divided equally, resulting in an unbalanced workload. The branch and bound approach also lead to

a non-uniform distribution of work as unnecessary paths are not computed. The use of parallel code v2, where all processors share results during computing, helps to mitigate this imbalance, as seen in Figure 9.

The future improvement for the algorithm could be to improve the processor sharing result approach by checking for the best bound at each iteration step instead of only checking at reaching ***level*=*n***. Another potential improvement would be to use header files for code reusability, making the code more readable and easier to maintain for future development and optimization of the algorithm.

Conclusion

The WSP problem was solved using a branch and bound method for both the serial and parallel algorithms. The parallel algorithm, implemented using MPI library, was found to perform better in terms of finding the best path faster than the serial algorithm, given a large enough problem size and sufficient number of processors. The approach of sharing results among processors reduced the number of iterations performed by each processor. Communication using MPI was implemented in all parallel code and found to have insignificant impact compared to the iteration time in the branch and bound function. Overall, the program effectively found the best path for the WSP problem. However, there is room for improvement in terms of load balancing by optimizing the point of sharing results and the code format can be improved by using header files to make it more readable and easier to maintain.

References

[1] Li Luxingz, 2015, “HIGH PERFORMANCE COMPUTING APPLIED TO CLOUD COMPUTING”, Thesis of Technology, Communication and Transport Programme in Information Technology, pp. 6.

[2] C. Guiffaut and K. Mahdjoubi, 2001, “A Parallel FDTD Algorithm Using the MPI Library”, IEEE Antennas and Propagation Magazine, Vol. AP-43, No. 2, pp. 94-103.

[3] Jakub Štencek, 2013, “Traveling salesman problem”, Bachelor’s Thesis of JAMK University of Applied sciences, pp. 5.

[4] Cartis, C, Fowkes, JM & Gould, NIM 2015, “Branching and bounding improvements for global optimization algorithms with Lipschitz continuity properties” Journal of Global Optimization, vol. 61, no. 3, pp. 429-457.

Appendices

All of source can be also found at https://github.com/kign17019999/github-hpc-v2.git

main_serial.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <time.h>
#include <string.h>

#define MAX_CITIES 20   // maximum number of cities that can be considered
#define INFINITE INT_MAX // a large constant used to initiation
#define START_CITIES 9 //start with 0 to n-1

int n; // number of cities
int (*dist)[MAX_CITIES]; // a 2D array to store distances between cities
int *best_path; // an array to store the best path found so far
int best_path_bound = INFINITE; // the length of the best path found so far

void get_cities_info(char* file_path);
void branch_and_bound(int *path, int path_bound, int *visited, int level);
void save_result(char* dist_file, double computing_time);
double power(double base, int exponent);

/*  Main function for WSP
    The function performs the following tasks:
    - Sets default file path to "input/distX" or uses the file path provided with
"-i" option
    - Calls get_cities_info to parse the distance information of the cities
    - Calls branch_and_bound to find the shortest path that visits all cities
    - Prints the best path, the best path distance and the computing time
    - Saves the result in a file
*/
int main(int argc, char *argv[]) {
    clock_t start = clock();

    /* set default array to the size of MAX_CITIES */
    dist = malloc(sizeof(int[MAX_CITIES][MAX_CITIES]));
    best_path = malloc(sizeof(int[MAX_CITIES]));

    /* retrive city information */
    char* file_path;
```

```
    if(argc >=3 && strcmp("-i", argv[1]) == 0){ // check there are enough input
to consider or not
        char* myArg = argv[2];
        while (myArg[0] == '\'') myArg++;   // filter some input mistake at the
front of file name
        while (myArg[strlen(myArg)-1] == '\'') myArg[strlen(myArg)-1] = '\0';  //
filter some input mistake at the last of file name
        file_path = myArg;
    }else{  // if there are no input of filename
        char *df_file = "input/dist13";
        printf("[System] The default file (%s) will be used if no input is
provided  \n", df_file);
        file_path  = df_file;
    }
    get_cities_info(file_path); // compute saving data from file to program

    /* initial path and vistied array */
    int *path = malloc(MAX_CITIES * sizeof(int));
    int *visited = malloc(MAX_CITIES * sizeof(int));
    for(int i=0; i<MAX_CITIES; i++) visited[i]=0;   //intiate vistied with value
0 (unvisited)
    path[0] = START_CITIES; //assign first city to START_CITIES
    visited[START_CITIES] = 1;  //mark that START_CITIES has been visited

    /* compute branch_and_bound */
    branch_and_bound(path, 0, visited, 1);
    double computing_time = (double)(clock() - start) / CLOCKS_PER_SEC;

    /* print the result */
    printf("best path bound = %d | total time = %f s | path = ", best_path_bound,
computing_time);
    for(int i=0; i<n; i++) printf("%d ", best_path[i]);
    printf("| Mode = Serial");
    printf("\n");

    /* save result to the file */
    save_result(file_path, computing_time);

    /* free memory */
    free(dist);
    free(best_path);
    return 0;
}
/*  get_cities_info: reads information about the cities and their distances
    from a file and stores it in the 2D array "dist".
```

```
    Input: file_path - a string specifying the file path of the input file.
*/
void get_cities_info(char* file_path) {
    FILE* file = fopen(file_path, "r");
    char line[256];
    fscanf(file, "%d", &n); //read number of city by the fist element in file

    int row=1;
    int col=0;

    dist[row-1][col]=0; //intial first value of dist array is 0 because it is
itself travelled

    while (fgets(line, sizeof(line), file)) {
        col=0;
        char *token = strtok(line, " ");    //get element information by checking
" " as a seperater
        while (token != NULL) {
            if(atoi(token)>0){
                dist[row-1][col] = atoi(token);
                dist[col][row-1] = atoi(token);
            }
            token = strtok(NULL, " ");
            col++;  // go to next elememement in a row
        }
        dist[row][col]=0;
        row++;  //go to next row
    }
    fclose(file);
}

/*  branch_and_bound is a recursive function to calculate the best path bound for
the WSP
    path: array to store the order of visiting cities
    path_bound: current bound of the path
    visited: array to store the visited status of the cities
    level: current level (city) of visiting
*/
void branch_and_bound(int *path, int path_bound, int *visited, int level) {
    if (level == n) {   // check that the travelling is going to the end of path
or not
        if (path_bound < best_path_bound) { //check the new path is better than
the previous best or not
            best_path_bound = path_bound;
            for (int i = 0; i < n; i++) best_path[i] = path[i]+1;
```

```
        }
    } else {
        for (int i = 0; i < n; i++) {
            if (!visited[i]) {  // check there are unvisited city persist or not
                path[level] = i;    // mark what is the path we are going to
evaluate
                visited[i] = 1; // mark the currect city is visited
                int new_bound = path_bound + dist[i][path[level - 1]]; //
calculate new bound
                if (new_bound < best_path_bound) branch_and_bound(path,
new_bound, visited, level + 1);
                visited[i] = 0; // reset the visited index, prepare to check new
unvisited city
            }
        }
    }
}
/*  This function saves WSP results to a .csv file. */
void save_result(char* dist_file, double computing_time) {
    /* make best path into a double format that can save into the same result
array which is a double type*/
    double double_path = power(10, 2*n)*404;
    for(int i=0; i<n; i++){ // add each city order into a large number sequently
        double_path+=power(10, (n-i-1)*2)*best_path[i];
    }

    FILE *file;
    char date[20];
    time_t t = time(NULL);
    struct tm tm = *localtime(&t);

    char* fileName="result_serial.csv";
    file = fopen(fileName, "r"); // open the file in "read" mode
    if (file == NULL) { // check there are existing file or not
        file = fopen(fileName, "w"); //create new file in "write" mode
        fprintf(file, "date-time, dist file, computing time (s), best_bound,
best_path\n"); // add header to the file
    } else {       // if no any existing file >> create the new file
        fclose(file);
        file = fopen(fileName, "a"); // open the file in "append" mode
    }

    // Get the current date and time
    strftime(date, sizeof(date), "%Y-%m-%d %H:%M:%S", &tm);
```

```
    fprintf(file, "%s,%s,%f, %d, %f\n", date, dist_file, computing_time,
best_path_bound, double_path); // add new data to the file

    fclose(file); // close the file
}

/* This function calculates the power of a base number to an exponent.*/
double power(double base, int exponent) {
    double result_power = 1;
    while (exponent > 0) {
        if (exponent & 1) result_power *= base;
        base *= base;
        exponent >>= 1;
    }
    return result_power;
}
```

main_parallel_v1.c

```c
#include <stdio.h>
#include <string.h>
#include <limits.h>
#include <stdlib.h>
#include <time.h>
#include <mpi.h>

#define MAX_CITIES 20   // maximum number of cities that can be considered
#define INFINITE INT_MAX    // a large constant used to initiation
#define ROOT 0  // define what is the ROOT processor of the program

#define LOOP_1ST 0 //set 1 to eneble loop all start city from START_CITIES to n
int START_CITIES=9; //start with 0 to n-1

/* MODE_SEND 0 = send Dist by Bcast       | MODE_SEND 1 = send Dist by Ibcast
   MODE_SEND 2 = send Dist by Send & Recv | MODE_SEND 3 = send Dist by Isend &
Irecv */
#define MODE_SEND 0

/* MODE_GATHER 0 = gather by Allgather | MODE_GATHER 1 = gather by Send & Recv */
#define MODE_GATHER 0

#define PRINT_ALL 0 //set 1 to eneble print the detial of result (each
start_city)
#define SAVE_CSV 1  //set 1 to enable CSV save function
#define NUM_RESULT 7    //number element in array of result

int n;  // number of cities
int (*dist)[MAX_CITIES], (*best_path)[MAX_CITIES];  // 2D array of dist_city and
best_path each proc
int *best_path_bound;   //1D array for best path bound each procs

/* city initiation variable */
int (*init_path)[MAX_CITIES], (*init_visited)[MAX_CITIES];  //2D array of
init_path and init_visited each procs
int *init_bound, *init_path_rank;   // 1D array of init_path_bound and array of
init_rank that tell procs should do what's init_path
int init_position, init_level, init_rank, num_init_path;

//other variable
double (*result)[NUM_RESULT]; // 2D array of result of each procs

#define NUM_BRA_BOU 1 //set 1 to eneble countinf of number of accessing to
brach_and_bound function
```

```c
double count_branch_bound;  // variable to count accessing

void get_cities_info(char* file_path);
void send_data_to_worker(int rank, int size);
void do_wsp(int rank, int size);
void level_initiation(int size);
void path_initiation(int *path_i, int path_bound, int *visited_i, int level, int
size);
void branch_and_bound(int *path, int path_bound, int *visited, int level, int
rank);
void gather_result(int rank, int size);
void save_result(int rank, int size, double total_computing_time, double
sending_time, double BaB_computing_time, double gathering_time, char* file_path);
void save_result_csv(double index_time, int rank, char *dist_file, double
total_computing_time, double sending_time, double BaB_computing_time, double
gathering_time, double count_bab, double r_best_bound, double r_best_path);
double power(double base, int exponent);

/*  Main function for WSP
    The function performs the following tasks:
    - Sets default file path to "input/distX" or uses the file path provided with
"-i" option
    - Calls get_cities_info to parse the distance information of the cities
    - Calls do_wsp which perform level_initiation > path_initiation >
branch_and_bound
    - Prints the best path, the best path distance and the computing time
    - Saves the result in a file
*/
int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    /* set the best parameter to print of from loop through
    all starter if needed*/
    int the_best_path_bound=INFINITE;
    int the_best_start, check_best=0;
    double time_of_the_best;
    int *the_best_path = malloc(sizeof(int[MAX_CITIES]));;

    while(1){   // loop all possible start city if needed (if not it will break
at the end of first iteration)
        if(rank==ROOT){
```

```
            if(PRINT_ALL==1)
printf("=================================================\n");
            if(PRINT_ALL==1) if(LOOP_1ST==1) printf("Start with City %d \n",
START_CITIES);
        }

        double start_time1 = MPI_Wtime();   //total time

        MPI_Request request1, request2;

        /* set default array */
        dist = malloc(sizeof(int[MAX_CITIES][MAX_CITIES])); //city information
        best_path = malloc(sizeof(int[size][MAX_CITIES]));  //size of array
depend on number of procs
        best_path_bound = malloc(sizeof(int[size]));    //size of best_bound
depend on number of procs
        best_path_bound[rank] = INFINITE;   //initiate by infinity

        /* retrive city information */
        char* file_path;
        if(argc >=3 && strcmp("-i", argv[1]) == 0){ // check there are enough
input to consider or not
            char* myArg = argv[2];
            while (myArg[0] == '\'') myArg++;   // filter some input mistake at
the front of file name
            while (myArg[strlen(myArg)-1] == '\'') myArg[strlen(myArg)-1] =
'\0';   // filter some input mistake at the last of file name
            file_path = myArg;
        }else{  // if there are no input of filename
            char *df_file = "input/dist10";
            if(PRINT_ALL==1) if (rank==ROOT) printf("    [ROOT] The default file
(%s) will be used if no input is provided  \n", df_file);
            file_path  = df_file;
        }


        if(rank==ROOT){ // compute saving data from file to program
            get_cities_info(file_path);
            if(PRINT_ALL==1) printf("    [ROOT] number of cities = %d \n", n);
            if(PRINT_ALL==1) printf("    [ROOT] number of processor = %d \n",
size);
        }

        //send city data to all processor
        double start_time2 = MPI_Wtime();   // timing for sending communication
```

```
        send_data_to_worker(rank, size);
        double end_time2 = MPI_Wtime();

        //solving wsp
        double start_time3 = MPI_Wtime();   // timing for wsp task
        do_wsp(rank, size);
        double end_time3 = MPI_Wtime();

        //gathering result
        double start_time4 = MPI_Wtime();   // timing for gathering communication
        gather_result(rank, size);
        double end_time4 = MPI_Wtime();

        /* update the best result from all procs*/
        if(rank==ROOT){
            int min_dist=INFINITE;
            int index_best_path;
            for(int i=0; i<size; i++){
                if(best_path_bound[i]<min_dist){
                    index_best_path = i;
                    min_dist = best_path_bound[i];
                }
            }

            if(PRINT_ALL==1) printf("    [ROOT] best of the best is in rank %d,
\n", index_best_path);
            if(PRINT_ALL==1) printf("      | best_path: ");
            for(int i = 0; i < n ; i++){
                if(PRINT_ALL==1) printf("%d ", best_path[index_best_path][i]);
            }
            if(PRINT_ALL==1) printf("\n");
            if(PRINT_ALL==1) printf("      | best_path_bound: %d \n",
best_path_bound[index_best_path]);

            /* update the best of all starting city */
            if(the_best_path_bound>best_path_bound[index_best_path]){
                the_best_path_bound = best_path_bound[index_best_path];
                the_best_start = START_CITIES;
                for(int i=0; i<n; i++)
the_best_path[i]=best_path[index_best_path][i];
                check_best=1;
            }

        }
```

```
        double end_time1 = MPI_Wtime(); //total time

        /* compute timing & printing the result */
        double total_computing_time = end_time1 - start_time1;
        double sending_time = end_time2 - start_time2;
        double BaB_computing_time = end_time3 - start_time3;
        double gathering_time = end_time4 - start_time4;
        if (rank ==ROOT){
            if(PRINT_ALL==1) printf("    [ROOT] spent total : %f seconds\n",
total_computing_time);
            if(PRINT_ALL==1) printf("    [ROOT] spent Send  : %f seconds\n",
sending_time);
            if(PRINT_ALL==1) printf("    [ROOT] spent BaB   : %f seconds\n",
BaB_computing_time);
            if(PRINT_ALL==1) printf("    [ROOT] spent Gather: %f seconds\n",
gathering_time);
        }

        if(check_best==1){ // update best time of starting city
            time_of_the_best = total_computing_time;
            check_best=0;
        }

        /* save result to the file */
        if(SAVE_CSV==1) save_result(rank, size, total_computing_time,
sending_time, BaB_computing_time, gathering_time, file_path);

        /* free memory */
        free(dist);
        free(best_path);
        free(best_path_bound);
        free(init_path);
        free(init_bound);
        free(init_visited);
        free(init_path_rank);

        /* check user need to loop or starting city or not */
        if(LOOP_1ST==1){
            START_CITIES++;
            if(START_CITIES==n) break;  // if there are no further starting city
> let break
        }else break;    //if not break here
    }

    /* print the best result of all starting city */
```

```
    if(rank==ROOT){
        printf("best path bound = %d | total time = %f s | path = ",
the_best_path_bound, time_of_the_best);
        for(int i=0; i<n; i++) printf("%d ", the_best_path[i]);
        printf("| RT = FALSE");
        printf("\n");
    }

    MPI_Finalize();
    return 0;
}

/*  get_cities_info: reads information about the cities and their distances
    from a file and stores it in the 2D array "dist".
    Input: file_path - a string specifying the file path of the input file.
*/
void get_cities_info(char* file_path) {
    FILE* file = fopen(file_path, "r");
    char line[256];
    fscanf(file, "%d", &n); //read number of city by the fist element in file

    int row=1;
    int col=0;
    dist[row-1][col]=0; //intial first value of dist array is 0 because it is
itself travelled

    while (fgets(line, sizeof(line), file)) {
        col=0;
        char *token = strtok(line, " ");    //get element information by checking
" " as a seperater
        while (token != NULL) {
            if(atoi(token)>0){
                dist[row-1][col] = atoi(token);
                dist[col][row-1] = atoi(token);
            }
            token = strtok(NULL, " ");
            col++;  // go to next elememement in a row
        }
        dist[row][col]=0;
        row++;  //go to next row
    }
    fclose(file);
}

/*  sending communication depending specify communicatin mode
```

```
*/
void send_data_to_worker(int rank, int size){
    MPI_Request request1, request2;
    if(rank==ROOT){
        if(MODE_SEND==0){
            // mode 0 = send Dist by Bcast
            if(PRINT_ALL==1) printf("    [ROOT] MODE_SEND = 0 (send Dist by
Bcast) \n");
            MPI_Bcast(&n, 1, MPI_INT, ROOT, MPI_COMM_WORLD);
            MPI_Bcast(dist, MAX_CITIES*MAX_CITIES, MPI_INT, ROOT,
MPI_COMM_WORLD);

        }else if(MODE_SEND==1){
            // mode 1 = send Dist by Ibcast
            if(PRINT_ALL==1) printf("    [ROOT] MODE_SEND = 1 (send Dist by
Ibcast) \n");
            MPI_Ibcast(&n, 1, MPI_INT, ROOT, MPI_COMM_WORLD, &request1);
            MPI_Ibcast(dist, MAX_CITIES*MAX_CITIES, MPI_INT, ROOT,
MPI_COMM_WORLD, &request2);

        }else if(MODE_SEND==2){
            // mode 2 = send Dist by Send & Recv
            if(PRINT_ALL==1) printf("    [ROOT] MODE_SEND = 2 (send Dist by Send
& Recv) \n");
            for (int i = 1; i < size; i++) {
                MPI_Send(&n, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
                MPI_Send(dist, MAX_CITIES*MAX_CITIES, MPI_INT, i, 0,
MPI_COMM_WORLD);
            }

        }else if(MODE_SEND==3){
            // mode 3 = send Dist by Isend & Irecv
            if(PRINT_ALL==1) printf("    [ROOT] MODE_SEND = 3 (send Dist by Isend
& Irecv) \n");
            for (int i = 1; i < size; i++) {
                    MPI_Isend(&n, 1, MPI_INT, i, 0, MPI_COMM_WORLD, &request1);
                    MPI_Isend(dist, MAX_CITIES*MAX_CITIES, MPI_INT, i, 0,
MPI_COMM_WORLD, &request2);
            }
        }
    }else {
        if(MODE_SEND==0){
            // mode 0 = send Dist by Bcast
            MPI_Bcast(&n, 1, MPI_INT, ROOT, MPI_COMM_WORLD);
```

```
            MPI_Bcast(dist, MAX_CITIES*MAX_CITIES, MPI_INT, ROOT,
MPI_COMM_WORLD);
        }else if(MODE_SEND==1){
            // mode 1 = send Dist by Ibcast
            int flag;
            MPI_Ibcast(&n, 1, MPI_INT, ROOT, MPI_COMM_WORLD, &request1);
            MPI_Ibcast(dist, MAX_CITIES*MAX_CITIES, MPI_INT, ROOT,
MPI_COMM_WORLD, &request2);
            do {
                MPI_Test(&request1, &flag, MPI_STATUS_IGNORE);
            } while (!flag);

            do {
                MPI_Test(&request2, &flag, MPI_STATUS_IGNORE);
            } while (!flag);

        }else if(MODE_SEND==2){
            // mode 2 = send Dist by Send & Recv
            MPI_Recv(&n, 1, MPI_INT, ROOT, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            MPI_Recv(dist, MAX_CITIES*MAX_CITIES, MPI_INT, ROOT, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        }else if(MODE_SEND==3){
            // mode 3 = send Dist by Isend & Irecv
            MPI_Irecv(&n, 1, MPI_INT, ROOT, 0, MPI_COMM_WORLD, &request1);
            MPI_Irecv(dist, MAX_CITIES*MAX_CITIES, MPI_INT, ROOT, 0,
MPI_COMM_WORLD, &request2);
            MPI_Wait(&request1, MPI_STATUS_IGNORE);
            MPI_Wait(&request2, MPI_STATUS_IGNORE);

        }
    }
}

/* perform level_initiation > path_initiation > branch_and_bound
*/
void do_wsp(int rank, int size){
    //set initial value
    init_position = init_level = init_rank = count_branch_bound = 0;
    num_init_path=1;
    if(START_CITIES>=n) START_CITIES=0; //if START_CITIES is larger than n > make
start with city 0

    //path and visit for initiation
    int *path_i = malloc(MAX_CITIES * sizeof(int));
```

```
    int *visited_i = malloc(MAX_CITIES * sizeof(int));
    for(int i=0; i<MAX_CITIES; i++) visited_i[i]=0; //intiate vistied with value
0 (unvisited)
    path_i[0] = START_CITIES;   //assign first city to START_CITIES
    visited_i[START_CITIES] = 1;    //mark that START_CITIES has been visited
    level_initiation(size); //calcuate initiate level (which level should enough
to distribute task to all processor)

    //path initiation
    init_path=malloc(sizeof(int[num_init_path][MAX_CITIES]));
    init_bound=malloc(num_init_path * sizeof(int));
    init_visited=malloc(sizeof(int[num_init_path][MAX_CITIES]));
    init_path_rank=malloc(num_init_path * sizeof(int));
    path_initiation(path_i, 0, visited_i, 1, size); //decompose task into array
of initiation path

    //path and visit for branch_and_bound
    int *path = malloc(MAX_CITIES * sizeof(int));
    int *visited = malloc(MAX_CITIES * sizeof(int));
    for(int i=0; i<num_init_path; i++){
        if(init_path_rank[i]==rank){ //check which initate that the procs should
perform refer to init_rank
            for(int j=0; j<n; j++){ //copy init_path into brach and bound path
                path[j] = init_path[i][j];
                visited[j] = init_visited[i][j];
            }
            branch_and_bound(path, init_bound[i], visited, init_level, rank); //
compute branch_and_bound
        }

    }

    /* free memory */
    free(path_i);
    free(visited_i);
    free(path);
    free(visited);
}

/*  calcuate a suitable level that the system can distribute task to all procs
equally
*/
void level_initiation(int size){
    for(int level=1; level<n; level++){
```

```
        num_init_path = num_init_path*(n-level);    // number of initiation path
in this level
        if(num_init_path >= size || level==n-1){    // if number of initiation
path is larger than  number of procs > enough!
            if(level==n-1) init_level=level;
            else init_level=level+1;
            break;
        }
    }
}

/*  do branch and bound to save intiate path until reaching a proper init_level
*/
void path_initiation(int *path_i, int path_bound, int *visited_i, int level, int
size) {
    if (level == init_level) {  // check that the travelling is going to reach
the end of init_level or not
        for(int i=0; i<n; i++){
            init_path[init_position][i] = path_i[i];
            init_visited[init_position][i] = visited_i[i];
        }
        init_bound[init_position]=path_bound;
        init_path_rank[init_position]=init_rank;
        init_position++;

        if(init_rank==size-1) init_rank=0;
        else init_rank++;
    } else {
        for (int i = 0; i < n; i++) {
            if (!visited_i[i]) {    // check there are unvisited city persist or
not
                path_i[level] = i;  // mark what is the path we are going to
evaluate
                visited_i[i] = 1;   // mark the currect city is visited
                int new_bound = path_bound + dist[i][path_i[level - 1]];    //
calculate new bound
                path_initiation(path_i, new_bound, visited_i, level + 1, size);
                visited_i[i] = 0;   // reset the visited index, prepare to check
new unvisited city
            }
        }
    }
}
```

```c
/*  branch_and_bound is a recursive function to calculate the best path bound for the WSP
    path: array to store the order of visiting cities
    path_bound: current bound of the path
    visited: array to store the visited status of the cities
    level: current level (city) of visiting
    rank: rank of proc in the system
*/
void branch_and_bound(int *path, int path_bound, int *visited, int level, int rank) {
    if(NUM_BRA_BOU==1) count_branch_bound+=1;   // incease number of access to branch and bound
    if (level == n) {   // check that the travelling is going to the end of path or not
        if (path_bound < best_path_bound[rank]) {   //check the new path is better than the previous best or not
            best_path_bound[rank] = path_bound;
            for (int i = 0; i < n; i++) best_path[rank][i] = path[i]+1;
        }
    } else {
        for (int i = 0; i < n; i++) {
            if (!visited[i]) {  // check there are unvisited city persist or not
                path[level] = i;    // mark what is the path we are going to evaluate
                visited[i] = 1; // mark the currect city is visited
                int new_bound = path_bound + dist[i][path[level - 1]];  // calculate new bound
            if (new_bound < best_path_bound[rank]) branch_and_bound(path, new_bound, visited, level + 1, rank);
            visited[i] = 0; // reset the visited index, prepare to check new unvisited city
            }
        }
    }
}

/*  Gathering communication depending specify communicatin mode
*/
void gather_result(int rank, int size){
    int send_buf_bound = best_path_bound[rank]; // set variable to store send buf (avoid buf issue)
    int row_to_gather[MAX_CITIES];  // set variable to store send buf (avoid buf issue)
    for (int i = 0; i < MAX_CITIES; i++) {
        row_to_gather[i] = best_path[rank][i];
```

```
    }

    if(MODE_GATHER==0){
        // MODE_GATHER 0 = gather by Allgather
        if(rank==ROOT) if(PRINT_ALL==1) printf("    ROOT] MODE_GATHER = 0 (gather
by Allgather) \n");
        MPI_Allgather(&send_buf_bound, 1, MPI_INT, best_path_bound, 1, MPI_INT,
MPI_COMM_WORLD);
        MPI_Allgather(&row_to_gather, MAX_CITIES, MPI_INT, best_path, MAX_CITIES,
MPI_INT, MPI_COMM_WORLD);
    }else if(MODE_GATHER==1){
        // MODE_GATHER 1 = gather by Send & Recv
        if(rank==ROOT){
            if(PRINT_ALL==1) printf("    [ROOT] MODE_GATHER = 1 (gather by Send &
Recv) \n");
            for(int i=1; i<size; i++){
                MPI_Recv(&best_path_bound[i], 1, MPI_INT, i, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
            }
            for(int i=1; i<size; i++){
                MPI_Recv(&row_to_gather, MAX_CITIES, MPI_INT, i, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                for(int j=0; j<n; j++) best_path[i][j]=row_to_gather[j];
            }
        }else{
            MPI_Send(&send_buf_bound, 1, MPI_INT, ROOT, 0, MPI_COMM_WORLD);
            MPI_Send(&row_to_gather, MAX_CITIES, MPI_INT, ROOT, 0,
MPI_COMM_WORLD);
        }
    }
}

/*  function share result among all procs */
void save_result(int rank, int size, double total_computing_time, double
sending_time, double BaB_computing_time, double gathering_time, char* file_path){
    result = malloc(sizeof(double[size][NUM_RESULT]));  // 2D array of result
depend on number of procs
    result[rank][0] = total_computing_time;
    result[rank][1] = sending_time;
    result[rank][2] = BaB_computing_time;
    result[rank][3] = gathering_time;
    result[rank][4] = count_branch_bound;
    result[rank][5] = best_path_bound[rank];
```

```
    double double_path = power(10, 2*n)*404;    /* make best path into a double
format that can save into the same result array which is a double type*/
    for(int i=0; i<n; i++){ // add each city order into a large number sequently
        double_path+=power(10, (n-i-1)*2)*best_path[rank][i];
    }
    result[rank][6] = double_path;

    // set buf of sending result (avoid duff issue)
    double row_to_gather_result[NUM_RESULT];
    for (int i = 0; i < NUM_RESULT; i++) row_to_gather_result[i] =
result[rank][i];

    // allgather result to all procs
    MPI_Allgather(row_to_gather_result, NUM_RESULT, MPI_DOUBLE  , result,
NUM_RESULT, MPI_DOUBLE  , MPI_COMM_WORLD);

    if(rank==ROOT){ // only save the result if rank = ROOT (avoid file access
problem)
        double index_time = MPI_Wtime();
        for(int i=0; i<size;i++) save_result_csv(index_time, i, file_path,
result[i][0], result[i][1], result[i][2], result[i][3], result[i][4],
result[i][5], result[i][6]);
    }

    /* free memory */
    free(result);
}

/*  This function saves WSP results to a .csv file. */
void save_result_csv(double index_time, int rank, char *dist_file, double
total_computing_time, double sending_time, double BaB_computing_time, double
gathering_time, double count_bab, double r_best_bound, double r_best_path) {
    FILE *file;
    char date[20];
    time_t t = time(NULL);
    struct tm tm = *localtime(&t);

    char* fileName="result_parallel.csv";
    file = fopen(fileName, "r"); // open the file in "read" mode
    if (file == NULL) { // check there are existing file or not
        file = fopen(fileName, "w"); //create new file in "write" mode
        fprintf(file, "index_time, rank, date-time, dist file,
total_computing_time (s), sending_time (s), BaB_computing_time (s),
gathering_time (s), count_BaB, best_bound, best_path, MODE_SEND, MODE_GATHER\n");
// add header to the file
```

```
    } else {    // if no any existing file >> create the new file
        fclose(file);
        file = fopen(fileName, "a"); // open the file in "append" mode
    }

    // Get the current date and time
    strftime(date, sizeof(date), "%Y-%m-%d %H:%M:%S", &tm);
    fprintf(file, "%f, %d, %s, %s, %f, %f, %f, %f, %f, %f, %f, %d, %d\n",
index_time, rank, date, dist_file, total_computing_time, sending_time,
BaB_computing_time, gathering_time, count_bab, r_best_bound, r_best_path,
MODE_SEND, MODE_GATHER); // add new data to the file

    fclose(file); // close the file
}

/* This function calculates the power of a base number to an exponent.*/
double power(double base, int exponent) {
    double result_power = 1;
    while (exponent > 0) {
        if (exponent & 1) result_power *= base;
        base *= base;
        exponent >>= 1;
    }
    return result_power;
}
```

main_parallel_v2.c

```c
#include <stdio.h>
#include <string.h>
#include <limits.h>
#include <stdlib.h>
#include <time.h>
#include <mpi.h>

#define MAX_CITIES 20   // maximum number of cities that can be considered
#define INFINITE INT_MAX    // a large constant used to initiation
#define ROOT 0  // define what is the ROOT processor of the program

#define LOOP_1ST 0 //set 1 to eneble loop all start city from START_CITIES to n
int START_CITIES=9; //start with 0 to n-1

/* MODE_SEND 0 = send Dist by Bcast       | MODE_SEND 1 = send Dist by Ibcast
   MODE_SEND 2 = send Dist by Send & Recv | MODE_SEND 3 = send Dist by Isend &
Irecv */
#define MODE_SEND 0

/* MODE_GATHER 0 = gather by Allgather | MODE_GATHER 1 = gather by Send & Recv */
#define MODE_GATHER 0

#define PRINT_ALL 0 //set 1 to eneble print the detial of result (each
start_city)
#define SAVE_CSV 1  //set 1 to enable CSV save function
#define NUM_RESULT 7    //number element in array of result

int n;  // number of cities
int (*dist)[MAX_CITIES], (*best_path)[MAX_CITIES];  // 2D array of dist_city and
best_path each proc
int *best_path_bound;   //1D array for best path bound each procs

/* city initiation variable */
int (*init_path)[MAX_CITIES], (*init_visited)[MAX_CITIES];  //2D array of
init_path and init_visited each procs
int *init_bound, *init_path_rank;   // 1D array of init_path_bound and array of
init_rank that tell procs should do what's init_path
int init_position, init_level, init_rank, num_init_path;

//other variable
double (*result)[NUM_RESULT];   // 2D array of result of each procs

#define NUM_BRA_BOU 1 //set 1 to eneble countinf of number of accessing to
brach_and_bound function
```

```c
double count_branch_bound;  // variable to count accessing

/* variable to perform Sharing communication*/
int all_best_bound; // the best_bound of all porc
MPI_Request *global_request_Isend;  //sending request address to track sharing
commmunication
MPI_Request *global_request_Irecv;  //gathering request address to track sharing
commmunication
int incoming_bound; //variable to store new sharing bound (checking state before
save into all_best_bound)
int global_flag;    //flag variable (easu access by all function)

void get_cities_info(char* file_path);
void send_data_to_worker(int rank, int size);
void do_wsp(int rank, int size);
void level_initiation(int size);
void path_initiation(int *path_i, int path_bound, int *visited_i, int level, int
size);
void branch_and_bound(int *path, int path_bound, int *visited, int level, int
rank, int size);
void gather_result(int rank, int size);
void save_result(int rank, int size, double total_computing_time, double
sending_time, double BaB_computing_time, double gathering_time, char* file_path);
void save_result_csv(double index_time, int rank, char *dist_file, double
total_computing_time, double sending_time, double BaB_computing_time, double
gathering_time, double count_bab, double r_best_bound, double r_best_path);
double power(double base, int exponent);

/*  Main function for WSP
    The function performs the following tasks:
    - Sets default file path to "input/distX" or uses the file path provided with
"-i" option
    - Calls get_cities_info to parse the distance information of the cities
    - Calls do_wsp which perform level_initiation > path_initiation >
branch_and_bound
    - Prints the best path, the best path distance and the computing time
    - Saves the result in a file
*/
int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);

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

```
    /* set the best parameter to print of from loop through
    all starter if needed*/
    int the_best_path_bound=INFINITE;
    int the_best_start, check_best=0;
    double time_of_the_best;
    int *the_best_path = malloc(sizeof(int[MAX_CITIES]));;

    while(1){   // loop all possible start city if needed (if not it will break
at the end of first iteration)
        if(rank==ROOT){
            if(PRINT_ALL==1)
printf("================================================\n");
            if(PRINT_ALL==1) if(LOOP_1ST==1) printf("Start with City %d \n",
START_CITIES);
        }

        double start_time1 = MPI_Wtime();   //total time

        MPI_Request request1, request2;

        /* set default array */
        dist = malloc(sizeof(int[MAX_CITIES][MAX_CITIES])); //city information
        best_path = malloc(sizeof(int[size][MAX_CITIES]));  //size of array
depend on number of procs
        best_path_bound = malloc(sizeof(int[size]));    //size of best_bound
depend on number of procs
        best_path_bound[rank] = INFINITE;   //initiate by infinity

        /* retrive city information */
        char* file_path;
        if(argc >=3 && strcmp("-i", argv[1]) == 0){ // check there are enough
input to consider or not
            char* myArg = argv[2];
            while (myArg[0] == '\'') myArg++;   // filter some input mistake at
the front of file name
            while (myArg[strlen(myArg)-1] == '\'') myArg[strlen(myArg)-1] =
'\0';   // filter some input mistake at the last of file name
            file_path = myArg;
        }else{  // if there are no input of filename
            char *df_file = "input/dist10";
            if(PRINT_ALL==1) if (rank==ROOT) printf("    [ROOT] The default file
(%s) will be used if no input is provided  \n", df_file);
            file_path  = df_file;
        }
```

```
        if(rank==ROOT){ // compute saving data from file to program
            get_cities_info(file_path);
            if(PRINT_ALL==1) printf("    [ROOT] number of cities = %d \n", n);
            if(PRINT_ALL==1) printf("    [ROOT] number of processor = %d \n",
size);
        }

        //send city data to all processor
        double start_time2 = MPI_Wtime();   // timing for sending communication
        send_data_to_worker(rank, size);
        double end_time2 = MPI_Wtime();

        //solving wsp
        double start_time3 = MPI_Wtime();   // timing for wsp task
        do_wsp(rank, size);
        double end_time3 = MPI_Wtime();

        //gathering result
        double start_time4 = MPI_Wtime();   // timing for gathering communication
        gather_result(rank, size);
        double end_time4 = MPI_Wtime();

        /* update the best result from all procs*/
        if(rank==ROOT){
            int min_dist=INFINITE;
            int index_best_path;
            for(int i=0; i<size; i++){
                if(best_path_bound[i]<min_dist){
                    index_best_path = i;
                    min_dist = best_path_bound[i];
                }
            }

            if(PRINT_ALL==1) printf("    [ROOT] best of the best is in rank %d,
\n", index_best_path);
            if(PRINT_ALL==1) printf("      | best_path: ");
            for(int i = 0; i < n ; i++){
                if(PRINT_ALL==1) printf("%d ", best_path[index_best_path][i]);
            }
            if(PRINT_ALL==1) printf("\n");
            if(PRINT_ALL==1) printf("      | best_path_bound: %d \n",
best_path_bound[index_best_path]);

            /* update the best of all starting city */
```

```
            if(the_best_path_bound>best_path_bound[index_best_path]){
                the_best_path_bound = best_path_bound[index_best_path];
                the_best_start = START_CITIES;
                for(int i=0; i<n; i++)
the_best_path[i]=best_path[index_best_path][i];
                check_best=1;
            }

        }

        double end_time1 = MPI_Wtime(); //total time

        /* compute timing & printing the result */
        double total_computing_time = end_time1 - start_time1;
        double sending_time = end_time2 - start_time2;
        double BaB_computing_time = end_time3 - start_time3;
        double gathering_time = end_time4 - start_time4;
        if (rank ==ROOT){
            if(PRINT_ALL==1) printf("    [ROOT] spent total : %f seconds\n",
total_computing_time);
            if(PRINT_ALL==1) printf("    [ROOT] spent Send  : %f seconds\n",
sending_time);
            if(PRINT_ALL==1) printf("    [ROOT] spent BaB   : %f seconds\n",
BaB_computing_time);
            if(PRINT_ALL==1) printf("    [ROOT] spent Gather: %f seconds\n",
gathering_time);
        }
        if(check_best==1){  // update best time of starting city
            time_of_the_best = total_computing_time;
            check_best=0;
        }

        /* save result to the file */
        if(SAVE_CSV==1) save_result(rank, size, total_computing_time,
sending_time, BaB_computing_time, gathering_time, file_path);

        /* free memory */
        free(dist);
        free(best_path);
        free(best_path_bound);
        free(init_path);
        free(init_bound);
        free(init_visited);
        free(init_path_rank);
```

```
        /* check user need to loop or starting city or not */
        if(LOOP_1ST==1){
            START_CITIES++;
            if(START_CITIES==n) break;  // if there are no further starting city
> let break
        }else break;    //if not break here
    }

    /* print the best result of all starting city */
    if(rank==ROOT){
        printf("best path bound = %d | total time = %f s | path = ",
the_best_path_bound, time_of_the_best);
        for(int i=0; i<n; i++) printf("%d ", the_best_path[i]);
        printf("| RT = TRUE");
        printf("\n");
    }

    MPI_Finalize();
    return 0;
}

/*  get_cities_info: reads information about the cities and their distances
    from a file and stores it in the 2D array "dist".
    Input: file_path - a string specifying the file path of the input file.
*/
void get_cities_info(char* file_path) {
    FILE* file = fopen(file_path, "r");
    char line[256];
    fscanf(file, "%d", &n);     //read number of city by the fist element in file

    int row=1;
    int col=0;
    dist[row-1][col]=0; //intial first value of dist array is 0 because it is
itself travelled

    while (fgets(line, sizeof(line), file)) {
        col=0;
        char *token = strtok(line, " ");    //get element information by checking
" " as a seperater
        while (token != NULL) {
            if(atoi(token)>0){
                dist[row-1][col] = atoi(token);
                dist[col][row-1] = atoi(token);
            }
            token = strtok(NULL, " ");
```

```
            col++;  // go to next elemement in a row
        }
        dist[row][col]=0;
        row++;  //go to next row
    }
    fclose(file);
}

/*  sending communication depending specify communicatin mode
*/
void send_data_to_worker(int rank, int size){
    MPI_Request request1, request2;
    if(rank==ROOT){
        if(MODE_SEND==0){
            // mode 0 = send Dist by Bcast
            if(PRINT_ALL==1) printf("    [ROOT] MODE_SEND = 0 (send Dist by
Bcast) \n");
            MPI_Bcast(&n, 1, MPI_INT, ROOT, MPI_COMM_WORLD);
            MPI_Bcast(dist, MAX_CITIES*MAX_CITIES, MPI_INT, ROOT,
MPI_COMM_WORLD);

        }else if(MODE_SEND==1){
            // mode 1 = send Dist by Ibcast
            if(PRINT_ALL==1) printf("    [ROOT] MODE_SEND = 1 (send Dist by
Ibcast) \n");
            MPI_Ibcast(&n, 1, MPI_INT, ROOT, MPI_COMM_WORLD, &request1);
            MPI_Ibcast(dist, MAX_CITIES*MAX_CITIES, MPI_INT, ROOT,
MPI_COMM_WORLD, &request2);

        }else if(MODE_SEND==2){
            // mode 2 = send Dist by Send & Recv
            if(PRINT_ALL==1) printf("    [ROOT] MODE_SEND = 2 (send Dist by Send
& Recv) \n");
            for (int i = 1; i < size; i++) {
                MPI_Send(&n, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
                MPI_Send(dist, MAX_CITIES*MAX_CITIES, MPI_INT, i, 0,
MPI_COMM_WORLD);
            }

        }else if(MODE_SEND==3){
            // mode 3 = send Dist by Isend & Irecv
            if(PRINT_ALL==1) printf("    [ROOT] MODE_SEND = 3 (send Dist by Isend
& Irecv) \n");
            for (int i = 1; i < size; i++) {
                    MPI_Isend(&n, 1, MPI_INT, i, 0, MPI_COMM_WORLD, &request1);
```

```
                    MPI_Isend(dist, MAX_CITIES*MAX_CITIES, MPI_INT, i, 0,
MPI_COMM_WORLD, &request2);
                }
            }
        }else {
            if(MODE_SEND==0){
                // mode 0 = send Dist by Bcast
                MPI_Bcast(&n, 1, MPI_INT, ROOT, MPI_COMM_WORLD);
                MPI_Bcast(dist, MAX_CITIES*MAX_CITIES, MPI_INT, ROOT,
MPI_COMM_WORLD);
            }else if(MODE_SEND==1){
                // mode 1 = send Dist by Ibcast
                int flag;
                MPI_Ibcast(&n, 1, MPI_INT, ROOT, MPI_COMM_WORLD, &request1);
                MPI_Ibcast(dist, MAX_CITIES*MAX_CITIES, MPI_INT, ROOT,
MPI_COMM_WORLD, &request2);
                do {
                    MPI_Test(&request1, &flag, MPI_STATUS_IGNORE);
                } while (!flag);

                do {
                    MPI_Test(&request2, &flag, MPI_STATUS_IGNORE);
                } while (!flag);

            }else if(MODE_SEND==2){
                // mode 2 = send Dist by Send & Recv
                MPI_Recv(&n, 1, MPI_INT, ROOT, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                MPI_Recv(dist, MAX_CITIES*MAX_CITIES, MPI_INT, ROOT, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);

            }else if(MODE_SEND==3){
                // mode 3 = send Dist by Isend & Irecv
                MPI_Irecv(&n, 1, MPI_INT, ROOT, 0, MPI_COMM_WORLD, &request1);
                MPI_Irecv(dist, MAX_CITIES*MAX_CITIES, MPI_INT, ROOT, 0,
MPI_COMM_WORLD, &request2);
                MPI_Wait(&request1, MPI_STATUS_IGNORE);
                MPI_Wait(&request2, MPI_STATUS_IGNORE);

            }
        }
    }

/* perform level_initiation > path_initiation > branch_and_bound
*/
void do_wsp(int rank, int size){
```

```
    //set initial value
    init_position = init_level = init_rank = count_branch_bound = 0;
    num_init_path=1;
    if(START_CITIES>=n) START_CITIES=0; //if START_CITIES is larger than n > make
start with city 0

    //path and visit for initiation
    int *path_i = malloc(MAX_CITIES * sizeof(int));
    int *visited_i = malloc(MAX_CITIES * sizeof(int));
    for(int i=0; i<MAX_CITIES; i++) visited_i[i]=0; //intiate vistied with value
0 (unvisited)
    path_i[0] = START_CITIES;   //assign first city to START_CITIES
    visited_i[START_CITIES] = 1;    //mark that START_CITIES has been visited
    level_initiation(size); //calcuate initiate level (which level should enough
to distribute task to all processor)

    //path initiation
    init_path=malloc(sizeof(int[num_init_path][MAX_CITIES]));
    init_bound=malloc(num_init_path * sizeof(int));
    init_visited=malloc(sizeof(int[num_init_path][MAX_CITIES]));
    init_path_rank=malloc(num_init_path * sizeof(int));
    path_initiation(path_i, 0, visited_i, 1, size);    //decompose task into
array of initiation path

    /* intiate Sharing communication by read and send some data to start the
sharing commu w/o check existing of request */
    all_best_bound = INFINITE;  // start with infinite bound
    global_request_Isend = malloc(size * sizeof(MPI_Request));  //define array
sie of sending request
    global_request_Irecv = malloc(size * sizeof(MPI_Request));  //define array
size of sending request
    for(int i=0; i<size; i++){  // looping to send all and get or data
        if(i!=rank){
            MPI_Isend(&all_best_bound, 1, MPI_INT, i, 2, MPI_COMM_WORLD,
&global_request_Isend[i]);
            MPI_Irecv(&incoming_bound, 1, MPI_INT, i, 2, MPI_COMM_WORLD,
&global_request_Irecv[i]);
        }
    }

    //path and visit for branch_and_bound
    int *path = malloc(MAX_CITIES * sizeof(int));
    int *visited = malloc(MAX_CITIES * sizeof(int));
    for(int i=0; i<num_init_path; i++){
```

```
        if(init_path_rank[i]==rank){    //check which initate that the procs
should perform refer to init_rank
            for(int j=0; j<n; j++){     //copy init_path into brach and bound
path
                path[j] = init_path[i][j];
                visited[j] = init_visited[i][j];
            }
            branch_and_bound(path, init_bound[i], visited, init_level, rank,
size);     // compute branch_and_bound
        }

    }

    /* free memory */
    free(path_i);
    free(visited_i);
    free(path);
    free(visited);
    free(global_request_Irecv);
    free(global_request_Isend);

}

/*  calcuate a suitable level that the system can distribute task to all procs
equally
*/
void level_initiation(int size){
    for(int level=1; level<n; level++){
        num_init_path = num_init_path*(n-level);    // number of initiation path
in this level
        if(num_init_path >= size || level==n-1){    // if number of initiation
path is larger than  number of procs > enough!
            if(level==n-1) init_level=level;
            else init_level=level+1;
            break;
        }
    }
}

/*  do branch and bound to save intiate path until reaching a proper init_level
*/
void path_initiation(int *path_i, int path_bound, int *visited_i, int level, int
size) {
    if (level == init_level) {  // check that the travelling is going to reach
the end of init_level or not
```

```
        for(int i=0; i<n; i++){
            init_path[init_position][i] = path_i[i];
            init_visited[init_position][i] = visited_i[i];
        }
        init_bound[init_position]=path_bound;
        init_path_rank[init_position]=init_rank;
        init_position++;

        if(init_rank==size-1) init_rank=0;
        else init_rank++;
    } else {
        for (int i = 0; i < n; i++) {
            if (!visited_i[i]) {    // check there are unvisited city persist or
not
                path_i[level] = i;  // mark what is the path we are going to
evaluate
                visited_i[i] = 1;   // mark the currect city is visited
                int new_bound = path_bound + dist[i][path_i[level - 1]];    //
calculate new bound
                path_initiation(path_i, new_bound, visited_i, level + 1, size);
                visited_i[i] = 0;   // reset the visited index, prepare to check
new unvisited city
            }
        }
    }
}

/*  branch_and_bound is a recursive function to calculate the best path bound for
the WSP
    path: array to store the order of visiting cities
    path_bound: current bound of the path
    visited: array to store the visited status of the cities
    level: current level (city) of visiting
    rank: rank of proc in the system
    sizeL number of proc in the system
*/
void branch_and_bound(int *path, int path_bound, int *visited, int level, int
rank, int size) {
    if(NUM_BRA_BOU==1) count_branch_bound+=1;   // incease number of access to
branch and bound
    if (level == n) {   // check that the travelling is going to the end of path
or not
        for(int i=0; i<size;i++){   // check the current global new best path
bound from all procs
            if(i!=rank){
```

```
                global_flag=0;
                MPI_Test(&global_request_Irecv[i], &global_flag,
MPI_STATUS_IGNORE);    //check the status of previos msg recv
                if(global_flag){
                    if(incoming_bound < all_best_bound){
                      MPI_Cancel(&global_request_Isend[i]);    // if the new
global bound is better than prebious information it has > cancle previous sharing
                      all_best_bound = incoming_bound;
                    }
                    MPI_Irecv(&incoming_bound, 1, MPI_INT, i, 2, MPI_COMM_WORLD,
&global_request_Irecv[i]); // read new update (inadvance! > it will no data until
other procs send something > it will save into incomming)
                }
            }
        }
        if (path_bound < all_best_bound) {  //check the new path is better than
the previous best or not
            best_path_bound[rank] = path_bound;
            all_best_bound = path_bound;
            for(int i = 0; i < n; i++) best_path[rank][i] = path[i]+1;
            for(int i = 0; i < size; i++) { // if new calucate found new best >>
sharing the best to all procs
                if(i != rank) {
                    MPI_Cancel(&global_request_Isend[i]);   // cancel previous
send (if existing) (MPI_cancel will not block any process)
                    MPI_Isend(&all_best_bound, 1, MPI_INT, i, 2, MPI_COMM_WORLD,
&global_request_Isend[i]); // sharing
                }
            }
        }
    } else {
        for (int i = 0; i < n; i++) {
            if (!visited[i]) {  // check there are unvisited city persist or not
                path[level] = i;    // mark what is the path we are going to
evaluate
                visited[i] = 1; // mark the currect city is visited
                int new_bound = path_bound + dist[i][path[level - 1]];  //
calculate new bound
            if (new_bound < all_best_bound) branch_and_bound(path, new_bound,
visited, level + 1, rank, size);
            visited[i] = 0; // reset the visited index, prepare to check new
unvisited city
            }
        }
    }
```

```
}

/*  Gathering communication depending specify communicatin mode
*/
void gather_result(int rank, int size){
    int send_buf_bound = best_path_bound[rank]; // set variable to store send buf
(avoid buf issue)
    int row_to_gather[MAX_CITIES];  // set variable to store send buf (avoid buf
issue)
    for (int i = 0; i < MAX_CITIES; i++) {
        row_to_gather[i] = best_path[rank][i];
    }

    if(MODE_GATHER==0){
        // MODE_GATHER 0 = gather by Allgather
        if(rank==ROOT) if(PRINT_ALL==1) printf("    [ROOT] MODE_GATHER = 0
(gather by Allgather) \n");
        MPI_Allgather(&send_buf_bound, 1, MPI_INT, best_path_bound, 1, MPI_INT,
MPI_COMM_WORLD);
        MPI_Allgather(&row_to_gather, MAX_CITIES, MPI_INT, best_path, MAX_CITIES,
MPI_INT, MPI_COMM_WORLD);
    }else if(MODE_GATHER==1){
        // MODE_GATHER 1 = gather by Send & Recv
        if(rank==ROOT){
            if(PRINT_ALL==1) printf("    [ROOT] MODE_GATHER = 1 (gather by Send &
Recv) \n");
            for(int i=1; i<size; i++){
                MPI_Recv(&best_path_bound[i], 1, MPI_INT, i, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
            }
            for(int i=1; i<size; i++){
                MPI_Recv(&row_to_gather, MAX_CITIES, MPI_INT, i, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                for(int j=0; j<n; j++) best_path[i][j]=row_to_gather[j];
            }
        }else{
            MPI_Send(&send_buf_bound, 1, MPI_INT, ROOT, 0, MPI_COMM_WORLD);
            MPI_Send(&row_to_gather, MAX_CITIES, MPI_INT, ROOT, 0,
MPI_COMM_WORLD);
        }
    }
}

/*  function share result among all procs */
```

```
void save_result(int rank, int size, double total_computing_time, double
sending_time, double BaB_computing_time, double gathering_time, char* file_path){
    result = malloc(sizeof(double[size][NUM_RESULT]));  // 2D array of result
depend on number of procs
    result[rank][0] = total_computing_time;
    result[rank][1] = sending_time;
    result[rank][2] = BaB_computing_time;
    result[rank][3] = gathering_time;
    result[rank][4] = count_branch_bound;
    result[rank][5] = best_path_bound[rank];
    double double_path = power(10, 2*n)*404;    /* make best path into a double
format that can save into the same result array which is a double type*/
    for(int i=0; i<n; i++){ // add each city order into a large number sequently
        double_path+=power(10, (n-i-1)*2)*best_path[rank][i];
    }
    result[rank][6] = double_path;

    // set buf of sending result (avoid duff issue)
    double row_to_gather_result[NUM_RESULT];
    for (int i = 0; i < NUM_RESULT; i++) row_to_gather_result[i] =
result[rank][i];

    // allgather result to all procs
    MPI_Allgather(row_to_gather_result, NUM_RESULT, MPI_DOUBLE  , result,
NUM_RESULT, MPI_DOUBLE  , MPI_COMM_WORLD);

    if(rank==ROOT){ // only save the result if rank = ROOT (avoid file access
problem)
        double index_time = MPI_Wtime();
        for(int i=0; i<size;i++) save_result_csv(index_time, i, file_path,
result[i][0], result[i][1], result[i][2], result[i][3], result[i][4],
result[i][5], result[i][6]);
    }

    /* free memory */
    free(result);
}

/*  This function saves WSP results to a .csv file. */
void save_result_csv(double index_time, int rank, char *dist_file, double
total_computing_time, double sending_time, double BaB_computing_time, double
gathering_time, double count_bab, double r_best_bound, double r_best_path) {
    FILE *file;
    char date[20];
    time_t t = time(NULL);
```

```
    struct tm tm = *localtime(&t);

    char* fileName="result_parallel_RT.csv";
    file = fopen(fileName, "r"); // open the file in "read" mode
    if (file == NULL) { // check there are existing file or not
        file = fopen(fileName, "w"); //create new file in "write" mode
        fprintf(file, "index_time, rank, date-time, dist file,
total_computing_time (s), sending_time (s), BaB_computing_time (s),
gathering_time (s), count_BaB, best_bound, best_path, MODE_SEND, MODE_GATHER\n");
// add header to the file
    } else {    // if no any existing file >> create the new file
        fclose(file);
        file = fopen(fileName, "a"); // open the file in "append" mode
    }

    // Get the current date and time
    strftime(date, sizeof(date), "%Y-%m-%d %H:%M:%S", &tm);
    fprintf(file, "%f, %d, %s, %s, %f, %f, %f, %f, %f, %f, %f, %d, %d\n",
index_time, rank, date, dist_file, total_computing_time, sending_time,
BaB_computing_time, gathering_time, count_bab, r_best_bound, r_best_path,
MODE_SEND, MODE_GATHER); // add new data to the file

    fclose(file); // close the file
}

/* This function calculates the power of a base number to an exponent.*/
double power(double base, int exponent) {
    double result_power = 1;
    while (exponent > 0) {
        if (exponent & 1) result_power *= base;
        base *= base;
        exponent >>= 1;
    }
    return result_power;
}
```

hpc.sub

```
#!/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 wsp
##
## 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=4: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
## 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 purin.tanirat.240@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_v1.out -i input/distances
mpirun -machinefile $PBS_NODEFILE -np ${cpus} ./a_v2.out -i input/distances

## Tidy up the log directory
## DO NOT CHANGE THE LINE BELOW
```

```
## ============================
rm $PBS_O_WORKDIR/$PBS_JOBID
#
```

the solution to the input/distances