CS546 “Parallel and Distributed Processing”

Final Project
First name: Guillermo
Last Name: de la Puente
CWID: A20314328
Attached to README.md and programs folder, which contains all the source code

---

Parallelization strategy:

Starting from the task subdivision provided in the project description:

A=2D-FFT(Im1)	(task1)
B=2D-FFT(Im2)	(task2)
C=MM_Point(A,B)	(task3)
C=Inverse-2DFFT(C)	(task4)

The parallel algorithm using MPI is as follows (source process = 0):

	Source process loads input	
communication	Source process scatters A and B	(task1+2)
computation	1D-FFT over rows in A and B	(task1+2)
communication	Source process gathers A and B	(task1+2)
seq computation	Transpose sequentially A and B	(task1+2)
communication	Source process scatters A and B	(task1+2)
computation	1D-FFT over rows (cols) in A and B	(task1+2)
computation	C is calculated as MM_Point(A,B)	(task3)
computation	Inverse-1D-FFT over rows in C	(task4)
communication	Source process gathers C	(task4)
seq computation	Transpose sequentially C	(task4)
communication	Source process scatters C	(task4)
computation	Inverse-1D-FFT over rows in C	(task4)
communication	Source process gathers C	(task4)
	Source process writes output	

“seq computation” stands for sequential computation in one process

Development settings

While developing the four different versions of the code, the matrix size used was 16x16. This allowed to

verify that the communications were being handled properly by printing matrices when necessary.

A specific method called *print_matrix* has been included in all programs:

```
/* Print the matrix if its size is no more than 32x32 */
/* Rank is the processor that should print this */
void print_matrix ( complex matrix[N][N], const char* matrixname, int rank );
```

The size of the matrices is hard-coded at the beginning of each program file, but it can be modified. It is located near line 30 of the four .c files.

```
/* Size of matrix (NxN) */
const int N = 512;  // change this to 16 to debug
```

A) MPI using Send and Recv functions

The algorithm used is exactly the one showed in page 1. The source process (0) is present in all communications acting as a master.

By using the MPI function *MPI_Wtime()* at the end of every step, computation and communication times were obtained separately.

Notice: this results are the average among multiple executions. Also, I've observed sometimes different execution times, probably due to the Jarvis cluster being more busy at certain moments. The execution time is more unstable for p=8, where the variation due to Tcommunication is higher.

	Time (ms)	Tcomputation	Tcommunication	Speedup
p=1	122	122	0	-
p=2	69	64.2	4.8	1.8
p=4	44	35.5	8.5	2.8
p=8	36	22	14	3.4

B) MPI using collective calls

The only thing that changes respect the previous step is that the sends and receives are substituted by scatters and gathers.

The results don't change much. Some differences can be seen in the communication, it can be infered that the collective calls improve communication performance when the number of processes grows, but this is a too small sample to reach conclusions.

	Time (ms)	Tcomputation	Tcommunication	Speedup
p=1	122	121	1	-
p=2	69	63.5	5.5	1.8
p=4	44	34.5	8.5	2.8
p=8	36	23	13	3.4

C) MPI + OpenMP

Because there are no loop-carried dependencies, OpenMP has been used to parallelize all the loops in the computation tasks. The number of threads is defined inside the project_3.c code in line 43:

```
#define NUM_THREADS 2
```

Pthreads could have been used for the same purpose, but the implementation with OpenMP was simpler by far.

This table shows the execution times for the program with 8 threads per processor. It can be seen that the worst performance is reached when 8 processors with 8 threads each are used, which is very inefficient.

The MPI functions used are collective calls, a detail that explains the non-zero value of communication when p=1.

	Time (ms)	**Tcomputation**	**Tcommunication**	**Speedup**
p=1	46	42	4	2.7
p=2	240	220	20	0.5
p=4	550	380	170	0.2
p=8	1100	620	480	0.1

Additionally, other situations have been tested. Keeping always the amount of execution lines to 8, this results were obtained:

	Time (ms)	**Tcomputation**	**Tcommunication**	**Speedup**
p=1, threads=8	46	42	4	2.7
p=2, threads=4	32	25	7	3.8
p=4, threads=2	32	23	9	3.8

The last case, p=8 and threads=1, is the regular MPI algorithm from the previous page.

The results show that the fest performance is both matched with the combinations (2,4) and (4,2). The speedup obtained, 3.8, is **the best one of all configurations tested.**

D) Task parallelism

This part requires that the processing units are divided into 4 groups, and each one of them will be in charge of one task. Since we are using 8 processors, each group will have 2. However, the source code has been developed to support larger groups, there's nothing hard-coded. For this reason, 16 processors could be used with groups of 4.

Each group of processors has been grouped though *MPI_Group*, and internal communications are made using specific intra-communicators, called *P1_comm*, *P2_comm*, *P3_comm* and *P4_comm.*

For 8 processors:

Group	P1	P2	P3	P4
Communicator	P1_comm	P2_comm	P3_comm	P4_comm
Ranks	0,1	2,3	4,5	6,7

The algorithm now is a little bit more sophisticated, since the source processor is no longer the one doing the sequential computations.

Read and distribute input, performed by source processor:

	Source process loads input
communication	Source process scatters A through the P1 processors
communication	Source process scatters B through the P1 processors

Task 1, 2D-FFT on A, performed by group P1:

computation	P1 processes apply 1D-FFT on A
communication	P1 first process gathers A from the other P1 processes
seq computation	P1 first process transposes A
communication	P1 first process scatters A to through other P1 processes
computation	P1 processes apply 1D-FFT on A
communication	Each P1 process sends its chunk of A to the corresponding P3 process

Task 2, 2D-FFT on B, performed by group P2 in parallel to task 1:

computation	P2 processes apply 1D-FFT on B
communication	P2 first process gathers B from the other P2 processes
seq computation	P2 first process transposes B
communication	P2 first process scatters B to through other P2 processes
computation	P2 processes apply 1D-FFT on B
communication	Each P2 process sends its chunk of A to the corresponding P3 process

Task 3, point to point multiplication performed by group P3 after tasks 1 and 2:

computation	P3 processes apply point to point multiplication of A and B to obtain C, each one in the chunks they received
communication	Each P3 process sends its chunk of C to the corresponding P4 process

Task 4, inverse 2D-FFT on C, performed by group P4 after task 3:

computation	P4 processes apply inverse 1D-FFT on C
communication	P4 first process gathers C from the other P4 processes
seq computation	P4 first process transposes C
communication	P4 first process scatters C to through other P4 processes
computation	P4 processes apply inverse 1D-FFT on C
communication	Each P4 process sends its chunk of C to the source process, the one that read the input originally

Gather output and write it to file

	Write to a file all the combined chunks of C received from P4 processors

As it can be observed, this algorithm requires more communications. The reason is that task 3 is now surrounded by two communications, while before they weren't necessary.

This is the execution time for P1=P2=P3=P4=2

	Time (ms)	Tcomputation	Tcommunication	Speedup
Px=2	55	47	8	2.2

E) Comparison between cases A and D

Of course, for 8 processors, the performance is much better in case A. The reason is that all processors are working all the time, while in case D, there are always free processors.

Where the potential of the option D comes is if multiple jobs came, different groups could be working in different jobs. The case needs to be studied, but we can suppose that the performance would be better because all processors would be busy.

Appendix: source code

The source code can be found together with this document. I decided not to put it here because it consists on more than 1,600 lines of code.

All the source code is located inside the folder "programs"
Go inside this folder with the command line

There, you will find different files:

- sample: folder with the samples given with the project

- project_0_sequential.c: sequential algorithm (not part of the project requirements)
- project_1.c: version of the program that uses MPI with Send & Recv
- project_2.c: version of the program that uses MPI with collective calls
- project_3.c: version of the program that uses MPI + OpenMP
- project_4.c: version of the program that uses task parallelism with four groups

- run_project_1.bash: script to execute project_1 in the Jarvis cluster
- run_project_2.bash: script to execute project_2 in the Jarvis cluster
- run_project_3.bash: script to execute project_3 in the Jarvis cluster
- run_project_4.bash: script to execute project_4 in the Jarvis cluster

- output_matrix: file containing the output of the last execution,
  it has the same format as the sample files provided with the project