Università della Svizzera italiana

Institute of Computing CI

High-Performance Computing **2021**

Student: Ashutosh Singh Discussed with: Self

Solution for Project 5

Due date: 21.11.2021, 23:59

1. Ring maximum using MPI [10 Points]

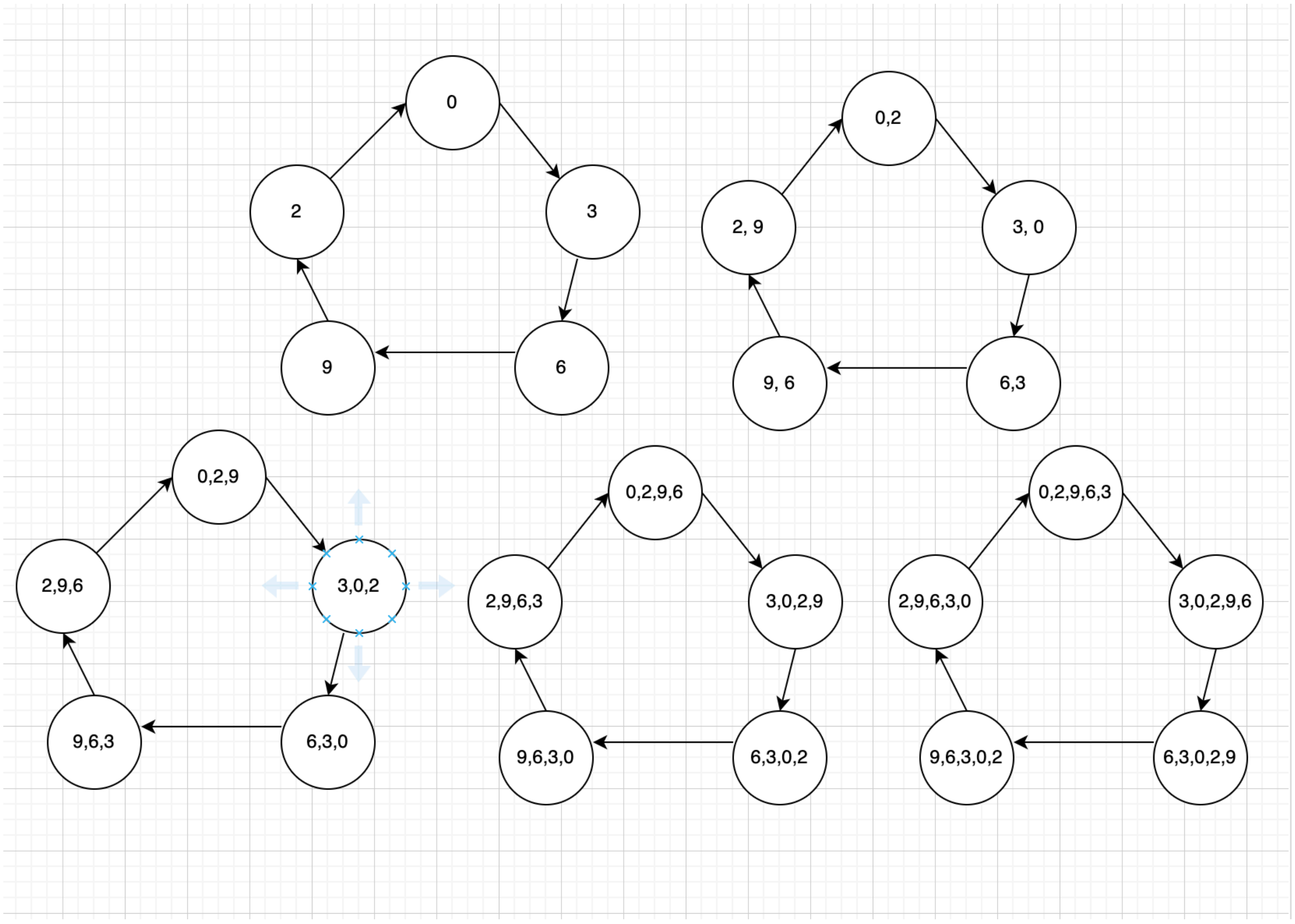

Figure 1: Grid Sizes Vs Log of Runtime(seconds) for Various number of threads. Click here for interactive figure

As clear from the Figure-1 each process has two neighbors, one of the left and one on right. The ranks of the neighbours processes are determined by considering a *period=Communicator Size*.The ranks of right and left neighbours of a process with *rank=my_rank* are given as -
$right = my\_rank + 1 \ mod \ Communicator\_Size$
$left = my\_rank + Communicator\_Size - 1 \ mod \ Communicator\_Size.$

At the start of the operation each $process_i$ has its send buffer filled according to the formula. For reference let's call it the original buffer of this process.

$$3 * process\_rank_i \text{ mod } 2 * communicator\_size, \text{ for i} = 0, 1, \ldots, communicator\_size - 1.$$

After the first the iteration the send buffer of each process in copied to receive buffer of the right neighbour. After *communicator_size* - 1 such iterations each process has once seen the original send buffer of all other processes.

I have used MPI_Isend to send the data from one process to another to prevent the deadlock. Each process has a *result* variable where it stores the max of value in *result* and the receive buffer.

$$result = max\{result, \ receive\_buffer\}$$

When all the iterations are complete every process has the same value of *result.*

2. Ghost cells exchange between neighboring processes [15 Points]

The computation in this problem is defined on a 2D Cartesian Grid. The processes have to act as if they are 2-Dimensional ordered and they need to communicate with their north, south, east and west neighbours.
The mechanism to declare the structure of a computation to MPI is known as a virtual topology and for this problem the structure is a Cartesian Topology where a process has two neighbours in each dimension.

In Cartesian topology each process is defined by a co-ordinate on the given Cartesian plane. The Cartesian plane itself is described as a space with d dimensions and having $\{n_1, n_2, ...n_d\}$ length along each dimension.
In MPI `MPI_Cart_create` method is used to create a Cartesian topology.

```
int MPI_Cart_create(MPI_Comm comm_old,
                    int ndims, int *dims, int *periods,
                    int reorder, MPI_Comm *comm_cart)
```

For this problem the 2D Cartesian topology we create is *periodic* in both dimensions which means that the rows and columns at the edges round around themselves. So first row has last row as its neighbour and similarly for the first and last column. Also we do no re-order the ranks of the processes.
For each process to find the rank of its N/S/E/W neighbours we use `MPI_Cart_shift`. Snippet from my solution -

```
 MPI_Cart_shift(comm_cart, 1, +1, &rank_left, &rank_right);
 MPI_Cart_shift(comm_cart, 0, -1, &rank_bottom, &rank_top);
```

Now to send the data from one process to another I use `MPI_Isend` and `MPI_Irecv`

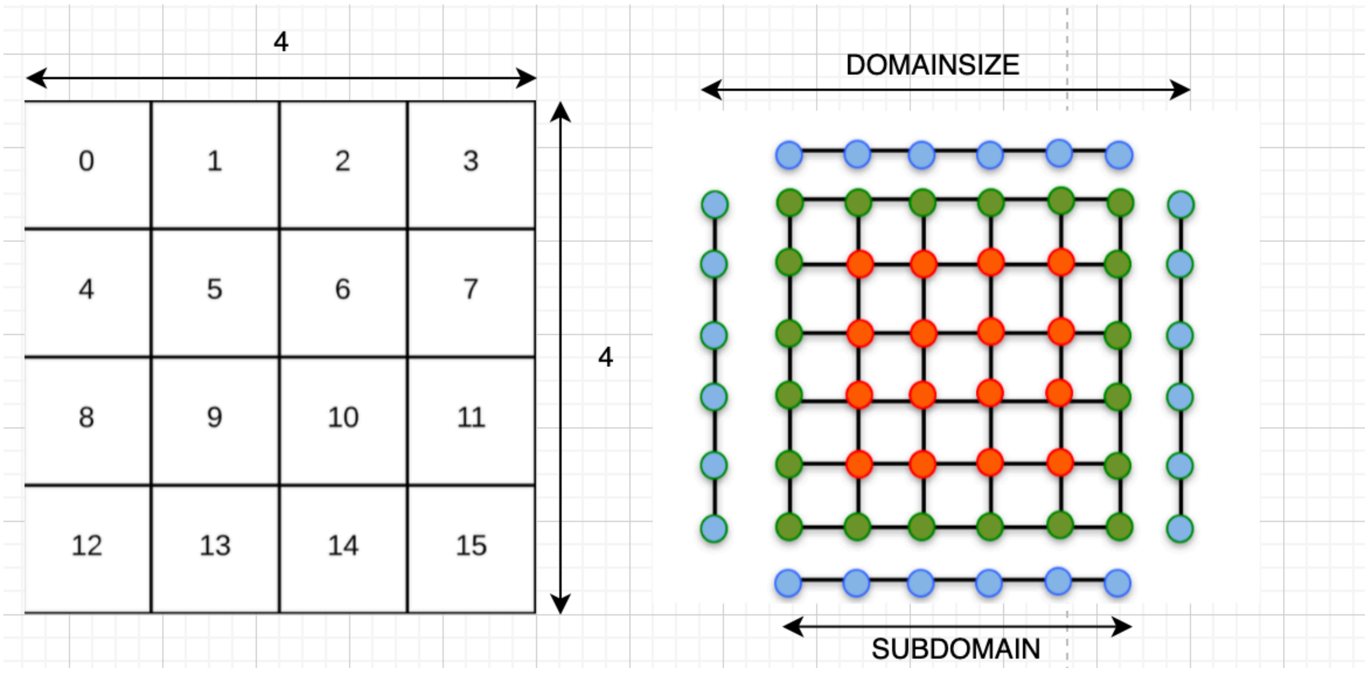

Figure 2: The Cartesian topology in problem(left) and one process of the topology(right) , each circular element(blue, green and orange) represents one block of memory or one element in a 2D array stored in a row major format.

As can be see from the Figure-2 each process works on array with $DOMAINSIZE * DOMAINSIZE$ elements. This array is stored in a row major format instead of a 2D array. While communicating with other processes, $SUBDOMAIN$ number of elements from top(in blue) are to be sent to the N-neighbour and similarly the bottom elements to the S-neighbour; sending to N/S neighbours is straightforward because a whole row is to be transmitted and the data array is stored in row major format so it is contiguous for each row. Snippet from code -

```
MPI_Isend(&data[1], SUBDOMAIN, MPI_DOUBLE, rank_top, ping, comm_cart, &request);
MPI_Isend(&data[DOMAINSIZE * DOMAINSIZE - 1 - SUBDOMAIN], SUBDOMAIN, MPI_DOUBLE,
    rank_bottom, ping, comm_cart, &request);
```

However for W/E neighbours it is not so simple since the blue points(from Figure) on the left and right and not stored as contiguous block of memory, but the data needs to accessed one blue point at a time after a stride of *DOMAINSIZE* elements. MPI provides a method to deal with such patterns , called `MPI_Type_vector`. Code Snippet depicting data access and transmission to E-Neighbour.

```
MPI_Type_vector(SUBDOMAIN, 1, DOMAINSIZE, MPI_DOUBLE, &data_ghost);
MPI_Type_commit(&data_ghost);
MPI_Isend(&data, 1, data_ghost, rank_left, ping, comm_cart, &request);
```

In the first line, `MPI_Type_vector` reads 1 memory block(blue element) at a time with a stride of *DOMAINSIZE*, *SUBDOMAIN* number of times from the *data* array (this array represents data in a row major format in one process). In the same way data is transmitted to the W-Neighbour.

The procedure is done for all processes - all sixteen blocks from the picture of left in Figure-2.
Code for receiving data is fairly straightforward and submitted with the solution. For E/W neighbours, they receive data in an array of size *SUBDOMAIN* i.e. the blocklength used by `MPI_Type_vector`.

3. Parallelizing the Mandelbrot set using MPI [20 Points]

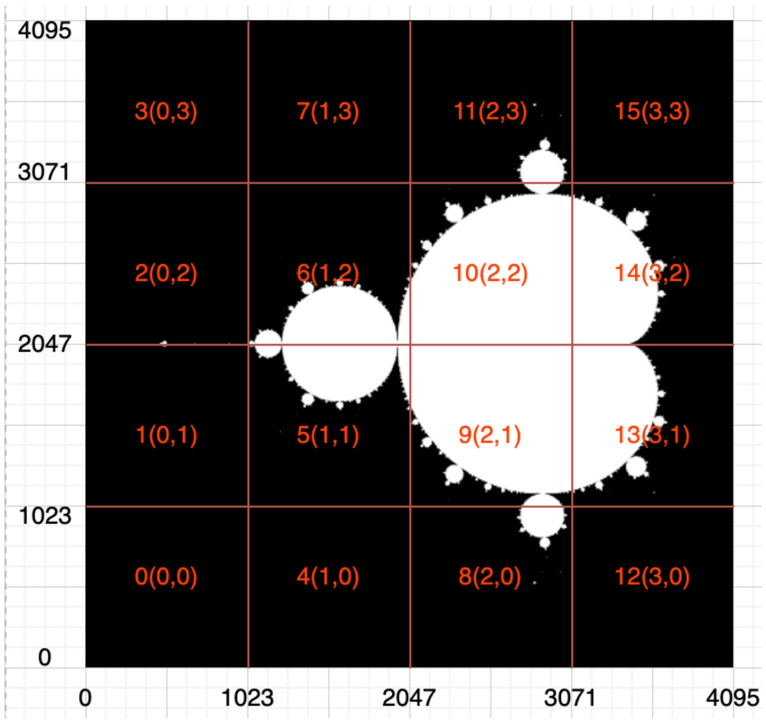

Figure 3: Image from the problem divided between 16 processes. Each process is labelled as rank(x, y). Where rank is the rank of the process in MPI Communicator and (x,y) are the co-ordinates of the process in the Cartesian Topology. Example 5(1,1); rank=5, x=1 and y=1

This problem requires partitioning an image among multiple processes in a Cartesian Topology. In the Figure I have partitioned the image into 16 parts for 16 processes.

A partition is described by a `Partition` object, contains a how many processes are there along each dimension of Cartesian Topology. It also contains the co-ordinates of the a given process in that topology.

The work of each processor is described a `Domain` object that is linked to a `Partition` and contains the number of pixels to operate on in each dimension and the start and end index in each dimension. `Domain` uses the co-ordinates of the process from `Partition` object to decide the start and end indices.
Each process works on only one `Domain` object to compute a results and stores them in a result array in row major format. For each process we have `c = malloc((d.nx) * (d.ny) * sizeof(int));` where `c` is the result array and `d` is the local domain for the process.

Each process(except for the root process) sends this result `c` top the root process using `MPI_Send` or others methods like it.

```
MPI_Isend(&c[0], d.nx*d.ny, MPI_INT, 0, ping, p.comm, &request);
```

The master process also works on its local domain just like other processes; but it does not send out any information. Also for master process result array `c` is defined as -

```
int extrax = IMAGE_WIDTH % p.nx;
int extray = IMAGE_HEIGHT % p.ny;
c = malloc((d.nx + extrax) * (d.ny + extray) * sizeof(int))
```

where the variables `extrax` and `extray` are the extra space to accommodate for the case where image size along each dimension is not completely divisible by number of processes along that dimension.
As already stated the master process does not send out any information; it is responsible for receiving the results from all other processes and writes them on the image.
It starts with its own `c` and then for each $process_i$(non master) -

- Updates its `Partition` object by using the rank of $process_i$
- Creates a local domain from this updated `Partition`
- Receives the result from $process_i$ and stores them in `c`
- Writes the result to image using start and end pixel indices from the `Domain` object

3.1. Results from run_perf.sh

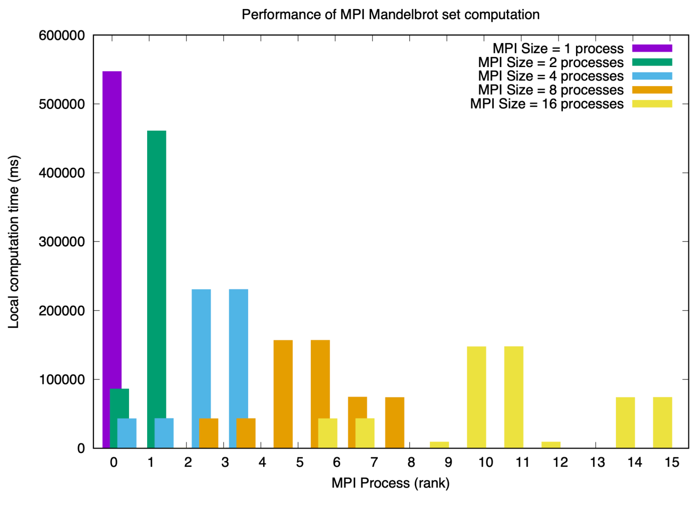

Figure 4: Plots from run_perf.sh

Total Processes	Rank	Local Comp. Time(ms)
16	0	4.377
16	1	109.403
16	2	258.156
16	3	4.297
16	4	8.149
16	5	42940.4
16	6	43109.8
16	7	8.108
16	8	9173.25
16	9	147508
16	10	147640
16	11	9216.06
16	12	10.878
16	13	73820.3
16	14	73882.5
16	15	10.882

Table 1: Table copied from perf.data file generated by run_perf.sh for mandel run with 16 processes

As we can see clearly from the Table-1 and Figure-3 the process whose local domain cover the white pixels do most of the work. This is somehow not reflected in the plot in Figure-4 but can be clearly seen from the table.

4. Option A: Parallel matrix-vector multiplication and the power method [40 Points]

The Power method is an iterative method to find the largest(in absolute value) eigen value of a diagonalizable matrix A, λ and the a vector x, which is the eigen-vector corresponding to λ.

The power iteration algorithm starts with a vector x_0, which may be an approximation to the dominant eigen-vector or a random vector. The method is described by the recurrence relation

$$x_{k+1} = \frac{Ax_k}{\|x_k\|_2}$$

For this problem matrix maultiplication is parallelized by spliting the rows of matrix A among the processes and copying the vector x_k to all the processes such that each process has $\frac{n}{p}$ rows of A; where n=size of square matrix and p=number of processes Each process computes the $\frac{n}{p}$ elements of x_{k+1}.

The master broadcasts x_k to all the processes using `MPI_Bcast` and collects $\frac{n}{p}$ elements from all process using `MPI_Gather`; this process is repeated many iterations till the power method converges.

4.1. Experiments

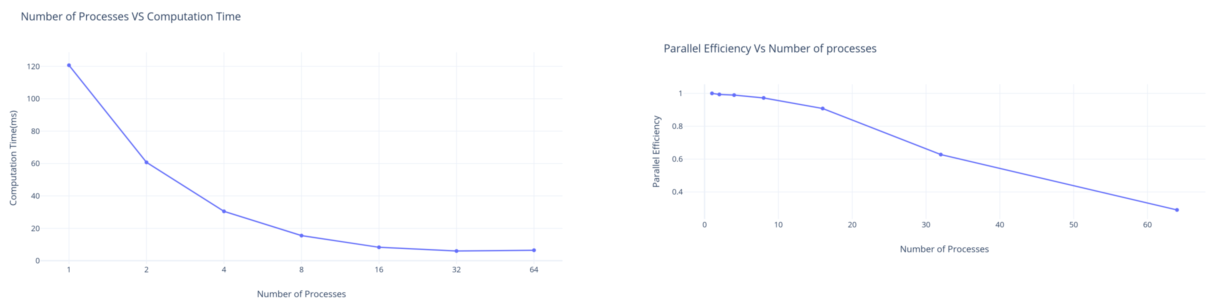

Figure 5: Strong Scaling Plot for n=10000 and p=1,2,4,8,16,32 and 64. Power method is run for 1000 iterations for each run. (Left)Computation Time vs Processes and (Right)Parallel Efficiency vs Processes

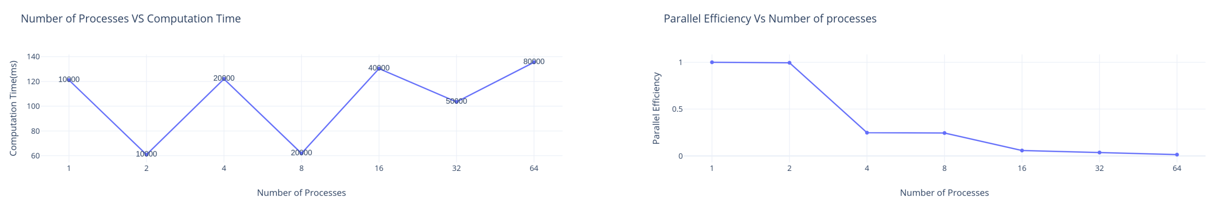

Figure 6: Weak Scaling.Plot of processes vs computation time for n=int(sqrt(p)) * 10000 and p=1,2,4,8,16,32 and 64. Power method is run for 100 iterations for each run

From Figure-5 it is clear that parallel efficiency decreases sharply after 16 processes as each process is not fully utilized and communication overhead becomes significant. Especially after 16 processes, where computation is faster for 32 processes than 64 processes.

From Figure-6 it can be seen that even though work per process is equal(except at p=2 and p=4) the trend in computation time is increasing again pointing to the state where inter-process communication overhead becomes significant and affects the computation speed overall.

5. Option B: Parallel PageRank Algorithm and the Power method [40 Points]

6. Task: Quality of the Report [15 Points]

Each project will have 100 points (out of which 15 points will be given to the general quality of the written report).

Additional notes and submission details

Submit the source code files (together with your used `Makefile`) in an archive file (tar, zip, etc.), and summarize your results and the observations for all exercises by writing an extended Latex report. Use the Latex template from the webpage and upload the Latex summary as a PDF to iCorsi.

- Your submission should be a gzipped tar archive, formatted like project_number_lastname_firstname.zip or project_number_lastname_firstname.tgz. It should contain:
  - all the source codes of your MPI solutions;
  - your write-up with your name project_number_lastname_firstname.pdf,
- Submit your .tgz through Icorsi.