Università della Svizzera italiana

Institute of Computing
CI

High-Performance Computing Lab **Institute of Computing**

Student: Paolo Deidda Discussed with: FULL NAME

Solution for Project 4

HPC Lab — Submission Instructions
(Please, notice that following instructions are mandatory: submissions that don't comply with, won't be considered)

- Assignments must be submitted to iCorsi (i.e. in electronic format).
- Provide sources (e.g. C/C++ files, Matlab). If you are using libraries, please add them in the file. Sources must be organized in directories called:
  Project_number_lastname_firstname
  and the file must be called:
  project_number_lastname_firstname.zip
  project_number_lastname_firstname.pdf
- The TAs will grade your project by reviewing your project write-up, and looking at the implementation you attempted, and benchmarking your code's performance.
- You are allowed to discuss all questions with anyone you like; however: (i) your submission must list anyone you discussed problems with and (ii) you must write up your submission independently.

Contents

1. Task: Ring maximum using MPI [10 Points]

I detrmine each process message destination and source using its rank.

The communication can be implemented using the standard MPI functions `MPI_Send` and `MPI_Recv`, which take the following arguments:

- buffer address (to store sent or received data)
- number of elements (to send or receive)
- MPI datatype (of the elements to send or receive)
- destination (or source) rank (of the process to send to or receive from)
- message tag (to identify the message with an id)
- communicator (to identify the group of processes involved in the communication, useless in this case)

When using `MPI_Send` and `MPI_Recv` separately, the ranks must be split into two groups like even and odd ranks, otherwise all processes may block by attempting to send or receive at the same time.

Alternatively, we can use the `MPI_Sendrecv` function, which takes care of both sending and receiving in a single call, avoiding deadlocks.

```
21      MPI_Sendrecv(
22        &send_val, 1, MPI_INT, next, 0,
23        &recv_val, 1, MPI_INT, prev, 0,
24        MPI_COMM_WORLD, MPI_STATUS_IGNORE
25      );
```

Listing 1: Implemetation from file ring_sum.c

Once launched the job we can see the folloring output with `-ntasks=4` on the left and `-ntasks=8` on the right.

```
Process 0: Sum = 6
Process 2: Sum = 6
Process 1: Sum = 6
Process 3: Sum = 6
```

Listing 2: Output from file ring_57313.out

```
Process 7: Sum = 28
Process 5: Sum = 28
Process 6: Sum = 28
Process 3: Sum = 28
Process 0: Sum = 28
Process 1: Sum = 28
Process 2: Sum = 28
Process 4: Sum = 28
```

Listing 3: Output from file ring_57315.out

With flag `-ntasks=4` slurm automatically assign a node with at least 4 cores, since by default:

- 1 Slurm task = 1 MPI process
- 1 MPI process = 1 CPU core

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

I first create a 4×4 Cartesian communicator with periodic boundaries on both axes. Then, by using `MPI_Cart_shift`, I get the ranks of the neighboring processesnorth, south, east, and west, making sure the first and last ranks stay connected. To handle the column ghost layer, I create a derived datatype using `MPI_Type_vector`; this datatype covers one element for every `DOMAINSIZE` entries and goes all the way through the interior height of the tile.

I then keep using `MPI_Sendrecv` calls, one for each direction, to avoid any deadlock. Each call sends the interior boundary (the rows or columns that leave out the corner points) and directly receives the data into the ghost layer.

```
105
106     //  to the top
107     MPI_Sendrecv(&data[1 * DOMAINSIZE + 1], SUBDOMAIN, MPI_DOUBLE, rank_top, 0,
108                  &data[(DOMAINSIZE - 1) * DOMAINSIZE + 1], SUBDOMAIN, MPI_DOUBLE, rank_bottom, 0,
109                  comm_cart, &status);
110
111     //  to the bottom
112     MPI_Sendrecv(&data[(DOMAINSIZE - 2) * DOMAINSIZE + 1], SUBDOMAIN, MPI_DOUBLE,
113                  rank_bottom, 1, &data[0 * DOMAINSIZE + 1],
```

Listing 4: Two example from top and bottom ghost cells exchange from the file ghost.c with `SUBDOMAIN = 6`

Here the output of the program:

```
data of rank 9 after communication
 9.0  5.0  5.0  5.0  5.0  5.0  5.0  9.0
 8.0  9.0  9.0  9.0  9.0  9.0  9.0 10.0
 8.0  9.0  9.0  9.0  9.0  9.0  9.0 10.0
 8.0  9.0  9.0  9.0  9.0  9.0  9.0 10.0
 8.0  9.0  9.0  9.0  9.0  9.0  9.0 10.0
 8.0  9.0  9.0  9.0  9.0  9.0  9.0 10.0
 8.0  9.0  9.0  9.0  9.0  9.0  9.0 10.0
 9.0 13.0 13.0 13.0 13.0 13.0 13.0  9.0
```

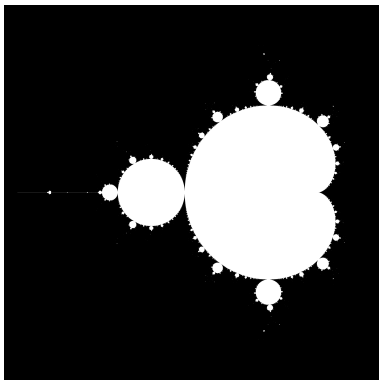
Listing 5: Output from file ghost_57333.out

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

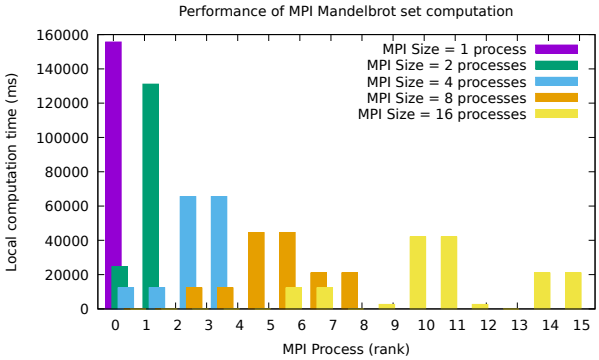
The function `createPartition` configure the process grid with `MPI_Dims_create`, then builds a Cartesian topology using the function `MPI_Cart_create`, and gets the coordinates for each process with `MPI_Cart_coords`. With this information, `createDomain` gives each process a rectangular subdomain, figuring out their global boundaries (`startx`, `endx`, `starty`, `endy`).

Next, each process calculates its own part of the Mandelbrot image independently. When they're done, the non-root processes send their sub-partition to the master using `MPI_Send`, while the master collects all the blocks with `MPI_Recv`, updates the partition through `updatePartition`, and places each block in the right spot in the final PNG.

The finished image is displayed in Figure 1a.



(a) Final Mandelbrot image computed in parallel (MPI).



(b) Per-process computation time for varying MPI process counts.

Figure 1: Mandelbrot set computed in parallel using MPI.

In Figure 1b, we can see that as we increase the number of MPI processes, the computation time for each process goes down quite a bit. It scales very good up to `8` processes. But when you push to `16` processes, the workload is light enough that you start to notice some overhead and imbalance, which is pretty typical for domain-splitting algorithms. The imbalance in workload distribution among precesses might be due to the static division of the image so the central processes end up with more complex calculations. This should be possible to resolve with a dynamic task allocation strategy but in OpenMP.

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

The given skeleton file, `powermethod_rows.c`, already took care of the serial part of the power iteration. I implemented spreading the matrix among the MPI processes, sending out the iterate vector, and combining the partial results to refine the eigenvalue estimate.

The MPI routines used were:

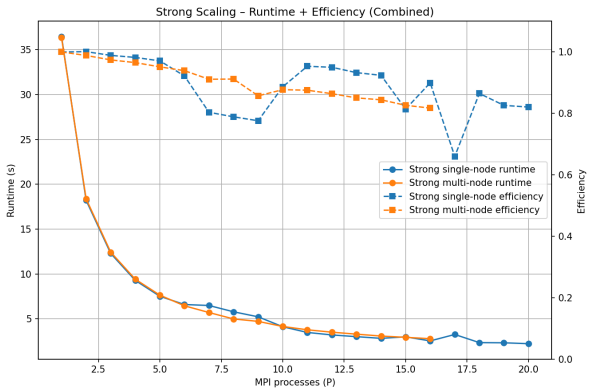
- `MPI_Scatterv`: distribute each block of matrix rows to its process
- `MPI_Bcast`: broadcast the current iterate vector x to all processes
- `MPI_Reduce`: compute global inner products required by the power iteration
- `MPI_Gather`: collect the updated vector entries on the master process

To run all required experiments, I used four automated Bash:

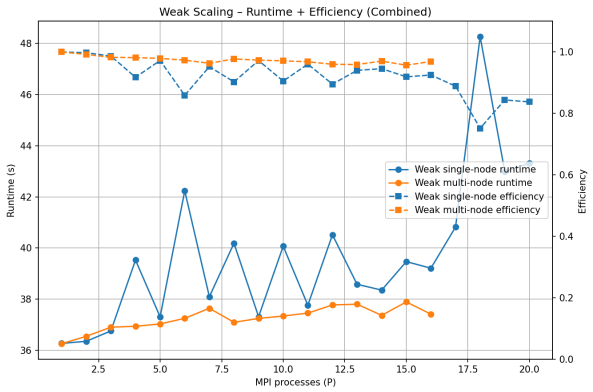
```
#!/bin/bash

sbatch run_strong_single.sh
sbatch run_strong_multi.sh
sbatch run_weak_single.sh
sbatch run_weak_multi.sh
```

Listing 6: Bash script to run all experiments run_all.sh

The resulting strong- and weak-scaling plots are shown in Figure 2. Each of these plots shows the runtime and efficiency curves for both single-node and multi-node setups.



(a) Strong scaling: runtime and efficiency.



(b) Weak scaling: runtime and efficiency.

Figure 2: Strong and weak scaling results for the power method.

Strong Scaling:
Analysis The strong scaling results show that we've done a good job with parallelization—runtimes keep dropping as we increase the number of processes. The multi-node setup (in orange) is really holding up well, keeping over 80% efficiency throughout the tests. On the other hand, the single-node setup (in blue) has been pretty unstable, especially when dealing with 9 and 17 processes, where we see a significant drop in efficiency. This difference points out that, while the algorithm logically scales well, cramming too many processes onto a single node hits some hardware limits—probably due to memory bandwidth issues—that are helped out when we spread the load over multiple nodes.

Weak Scaling:
After correcting the efficiency formula, the weak scaling graph shows we're close to ideal performance for the multi-node setup. The runtime for the multi-node system (orange solid line) stays nearly flat, and the efficiency (orange dashed line) stays pretty close to 1.0, which shows that we're not adding much overhead as we scale both the problem size and the number of processors. In contrast, the single-node performance (in blue) is all over the place, with big spikes in runtime and drops

in efficiency. This really confirms that a single machine can't handle the total memory bandwidth needed as the problem size grows, underscoring why a multi-node approach is necessary.

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