Università della Svizzera italiana

Institute of Computing CI

High-Performance Computing **2021**

Student: Pratyai Mazumder Discussed with: Zenin Easa Panthakkalakath

Solution for Project 5

Due date: 21.11.2021, 23:59

HPC 2021 — 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 both executable package and 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.

1. Ring maximum using MPI [10 Points]

Please check the attached file `ring/max_ring.c` for complete source code.

The implementation follows the iterative idea from the assignment, where each node starts with its own rank as the payload, then in each of the next *size* iterations **asynchronously sends** its current payload to its immediately next neighbor on the ring, and then **synchronously waits** for the payload from its immediately previous neighbor on the ring to replace the current payload. Finally, it waits to make sure that the asynchronous send indeed went through.

This blocks the algorithm at each iteration until each process is done with their part in the iteration. The asynchronous message-passing helps to avoid the deadlock where each process is trying to send to a process who is not receiving, because in turn that process is also trying to send to another process is not receiving, and so on – essentially, the Dining Philosophers problem.

We could also consider making the receiving synchronous if the subproblems that are distrbuted among the processes were of uneven complexity and we wanted to unblock any process that finishes early regardless its rank. For our simple program it was rather unnecessary.

Output:

```
[stud24@icsnode18 ring]$ mpirun -np 4 ./max_ring
Process 0:      Max = 6
Process 1:      Max = 6
Process 2:      Max = 6
Process 3:      Max = 6
```

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

Please check the attached file `ghost/ghost.c` for complete source code.

The implementation follows the instructions in the comments of the provided boilerplate. I.e. each process

- Creates a cartesian communicator to establish the topology of the connected processes.
- Finds the size of that topology and its own rank in it.
- Establishes the data types (i.e. row and column vectors) that will be used for communication.
- **Asynchronously sends** the appropriate row or column from its own boundary to each of its neighbors, who **synchronously receives** it and places it in the appropriate ghost boundary region.
- Finally, waits to make sure that the asynchronous sends indeed went through.

Again, in this case the receives were left synchronous to avoid unnecessary complication for this simple program.

Output:

```
[stud24@icsnode18 ghost]$ mpirun -np 16 ./ghost
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
```

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

3.1. Task 1, 2: Implement 'createPartition()', 'updatePartition()', 'createDomain()'.

Please check the attached file `mandel/consts.h` for complete source code.

There isn't much of note here – it just establishes the topology and defines the problem the algorithm will be solving.

3.2. Task 3: Synchronize processes.

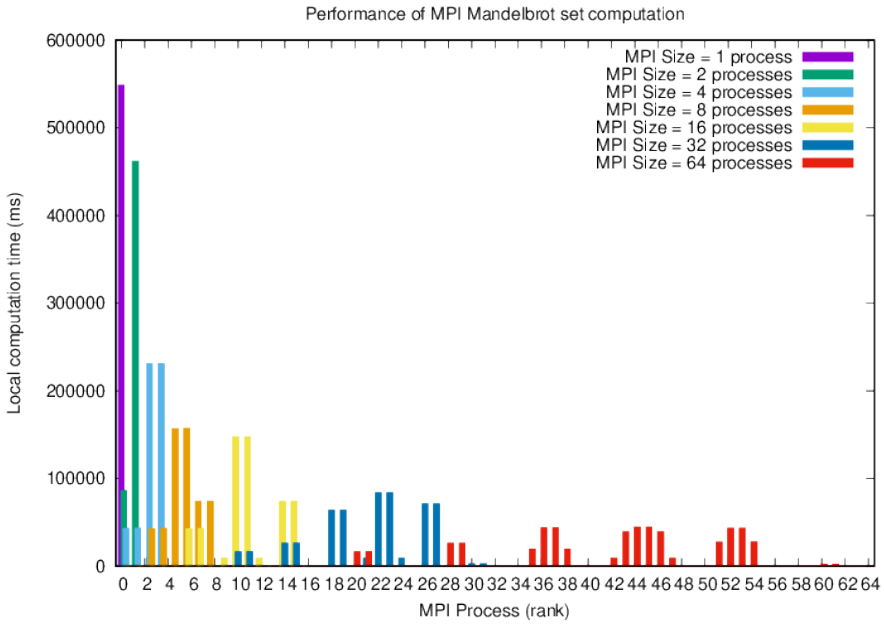
Please check the attached file `mandel/mandel_mpi.c` for complete source code.

The Mandelbrot process was already formulated in the boilerplate. The implementation of the parallelism followed the instructions in the assignment. I.e. subdivide the problem into a fixed number of subproblems of equal size, let each process solve exactly one problem completely, then collect the generated part of the image to produce the full image.
Things of note–

- The problem size (i.e. the image size) was not defined in compile-time. So, the arrays were heap-allocated (and appropriately freed at the end).

- Only the master process needs enough memory to store the full image. Every other process needs to store only the part of the image it works with. So, the heap-allocated arrays were of different sizes based on the rank of the process.
- All the sends and receives this time were **synchronous**, since this is essentially a "star network" where the other processes do not talk to each other and there is no opportunity for a cyclical dependency. Again, we could make the receives asynchronous to let the processes that finish early do other tasks – but our partitioning is too rigid for that anyway.
- Indeed, the computational complexity of the different partitions are uneven. We could possibly benefit from more but smaller partitions and dynamically allocating them.

3.3. Task 4: Analyze 'perf.ps'.



The chart shows the totall processing time of the individual MPI processes for a given parallelism (represented by the colour). Solving the whole problem will take at least the maximum of the individual subproblem-solving times for any given parallelism.

Clearly, the maximum subproblem-solving time noticeably drops with the increased parallelism (although it's not a perfect scaling) at the begining. However, this doesn't hold anymore after 8 MPI processes.

We already know from the previous version of this problem that the subproblems are not equally computationally-expensive. Computing the white pixels (which tend to appear close to each other) takes much longer than the black pixels due to the divergence condition. This explains the drastic difference between the processing time of different MPI processes.

This also somewhat explains why scaling is not perfect. One explanation is that during the MPI Size = 8 run, the processes ranked 4 and 5 solved a mixture of black and white pixels, and during the MPI Size = 16 run, the processes 9 and 10 got those white pixels but not the black ones due to the aforementioned non-uniform distribution of the white pixels. However, this was not rigourously verified.

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

4.1. Implementation

Please check the attached files `powermethod/powermethod.c` for complete source code.

Beyond the almost trivial linear-algebra implementation, the things of note are –

- The problem spec (i.e. the matrix size and the number of iterations) were taken from the commandline argument. So, the matrix had to be heap-allocated.
- The powermethod iterations are themselves sequential – i.e. we must complete one iteration before proceeding to the next one.
- Within one iteration, the parallelization opportunity comes from splitting the matrix into blocks such that each block contains only full columns. The vector x is multiplied by these blocks (each a submatrix itself) to fully compute the components of x associated with the block for the next iteration. Since each process computes a different, disjoint subset of the components of x, there is not much synchronization to be done beyond just sending the results to the master and receving the vector for the next iteration.
- Since each process works with only a subset of the contiguous columns of the matrix, they did not need to allocate the whole n^2 memory – only the master processed needs that.
- The whole iteration-step is extremely simple – one `MPI_Bcast()` call to distribute the entire current x from the master to all the processes[1], one matrix multiplication to compute the appropriate components of x for the next iteration[2], and one `MPI_Gather()` call that places all the components of the next x to the appropriate offset in the memory of the master.

4.2. Analysis

Evidently the algorithm is highly parallelizable. The parallel efficiency for a fixed problem size stays almost the ideal until the number of processes goes past 12, and drops significantly after 16. On ICS there are only 20 cores available per node, so it makes sense that there will be some overhead for inter-node communication.

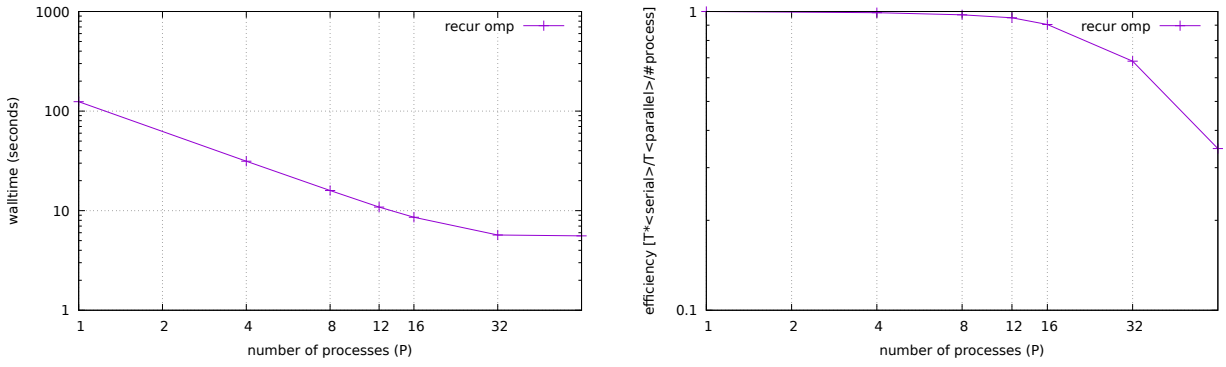


Figure 1: Strong-scaling walltime and parallel efficiency for $N = 10176$

Similarly, the parallel efficiency for a fixed problem size per processs too stays almost the ideal until the number of processes goes past 16, due to inter-node communication overhead.

[1]Stored in the variable "x" in every process including the master.
[2]Stored in the variable "y" in every process including the master.

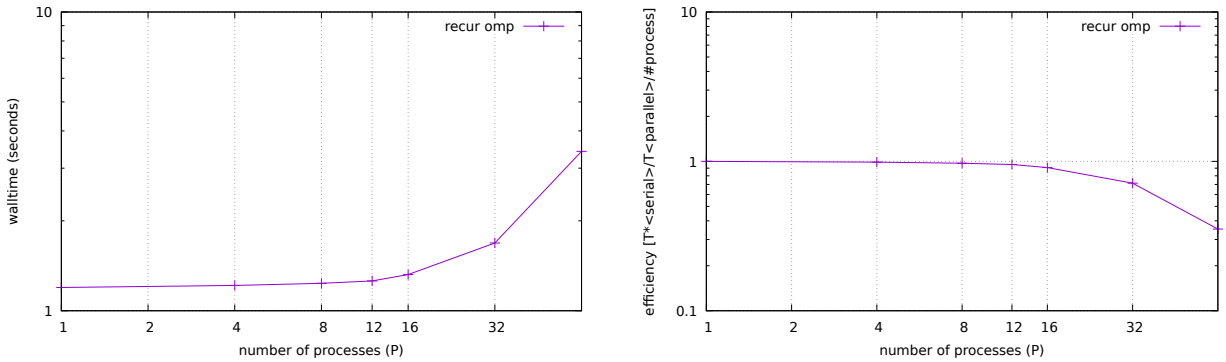


Figure 2: Weak-scaling walltime and parallel efficiency for $\frac{N^2}{process} \approx 3162$

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

Not done.

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

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.