Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

High-Performance Computing Lab for CSE **2024**

Student: Noah Gigler Discussed with: Luis Wirth, Felicia Scharitzer, Valentin Vogt

---

Solution for Project 4

Due date: Monday 29 April 2024, 23:59 (midnight).

HPC Lab for CSE 2024 — Submission Instructions
(Please, notice that following instructions are mandatory: submissions that don't comply with, won't be considered)

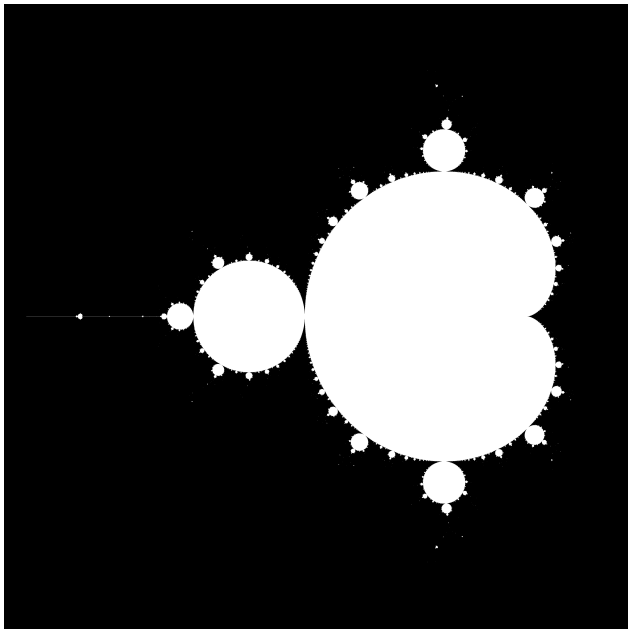
- Assignments must be submitted to Moodle (i.e. in electronic format).
- Provide both executable package and sources (e.g. C/C++ files, Matlab). If you are using libraries, please add them in the file. Sources must be organized in directories called:
  Project_number_lastname_firstname
  and the file 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 sum using MPI [10 Points]

Implementing this programm just requires basic understanding of how the protocols work. After coding the version where every programm sends and receieves at once we can make a small change in order to avoid deadlocks.

```
if(rank % 2 == 0){
    MPI_Send(&p_rank, 1, MPI_INT, next, 0, MPI_COMM_WORLD);
    MPI_Recv(&n_rank, 1, MPI_INT, prev, 0, MPI_COMM_WORLD, &status);
  }
  else {
    MPI_Recv(&n_rank, 1, MPI_INT, prev, 0, MPI_COMM_WORLD, &status);
    MPI_Send(&p_rank, 1, MPI_INT, next, 0, MPI_COMM_WORLD);
  }
```

Here all the processes with even rank send first and receieve later and the ones with odd rank do the opposite. This means all neighbours will do the opposite thing. One will send and the other will wait to receieve and then they do the opposite. No deadlocks can occur this way.



2. Cartesian domain decomposition and ghost cells exchange [20 Points]

In this section we have to create a new communicator with MPI_Cart_create in order to be able to exchange data in our specified domain. Then while exchanging data we use:

```
MPI_Irecv and MPI_Isend
```

As supposed to the last exercise these functions do not lock the process from continuing. This is done in order to avoid a deadlock as we are sending and receiving too chaotically to manually organize this. In order for this not to cause problems we have to:

```
MPI_Waitall(8, requests, MPI_STATUSES_IGNORE);
```

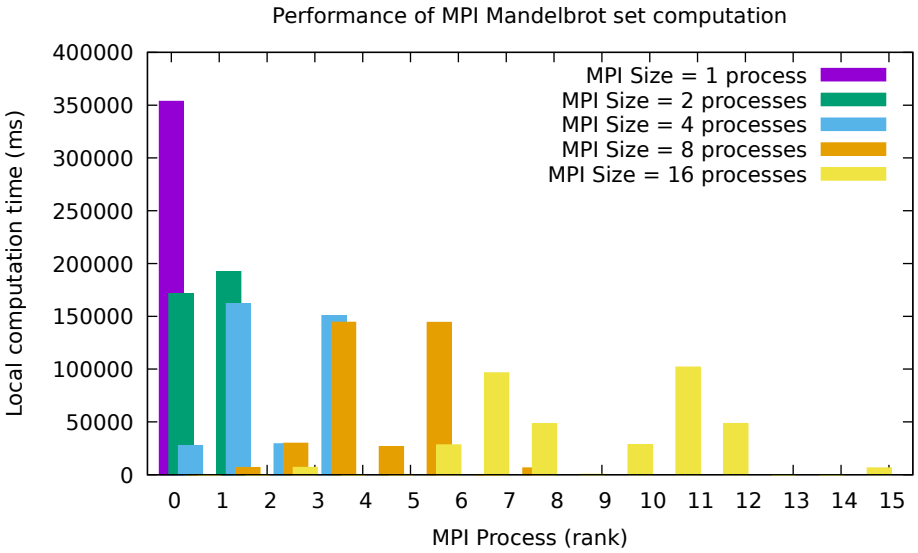
before doing anything with the values. This function insures all the outstanding Isend and Irecv functions are completed before continuing.

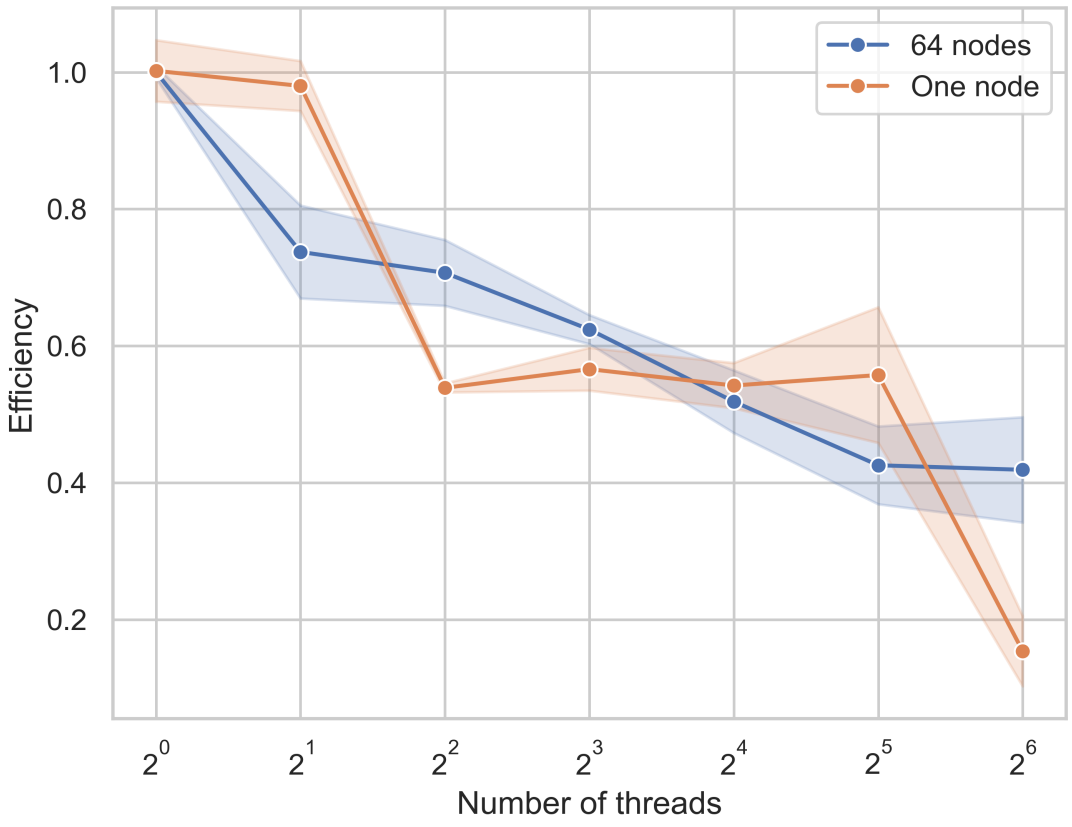
3. Parallelizing the Mandelbrot set using MPI [30 Points]

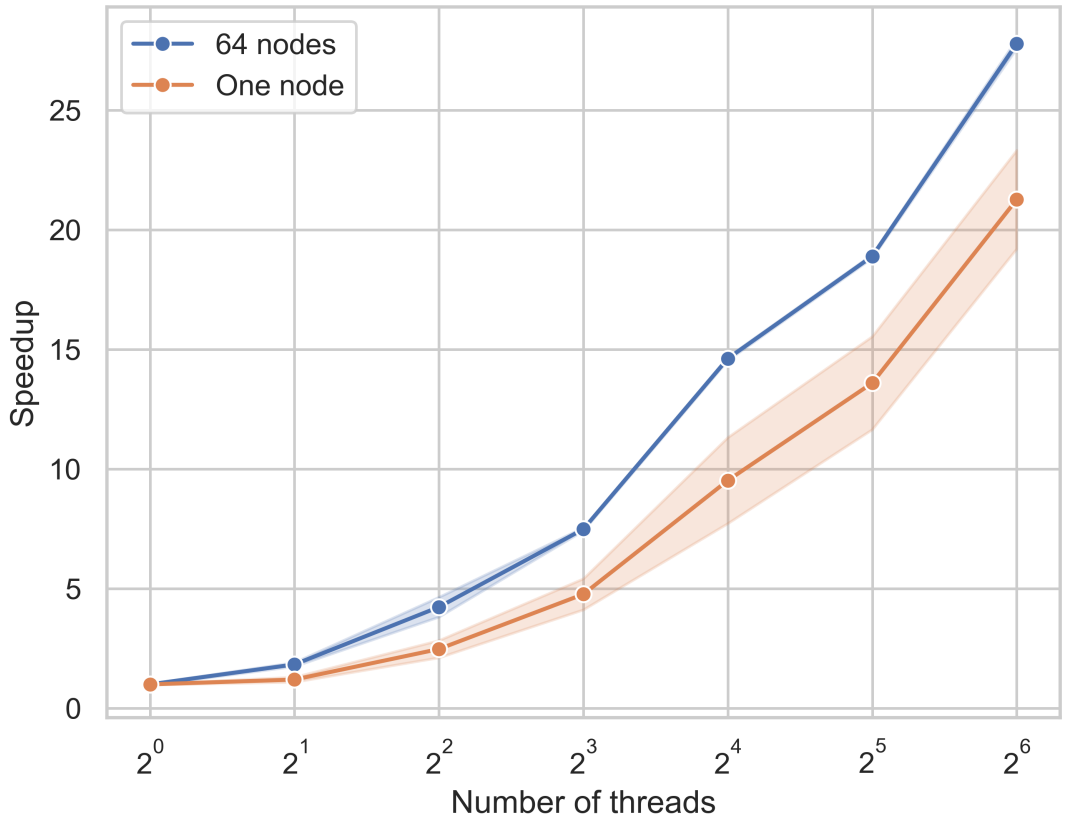
As we can see in the graph the slowest performance is the single core performance. When splitting the domain in the middle we are dividing the work about equally which leads too the version with two processes being about twice as fast. However if we look at the mandel set we see the problem with the 4 threaded version. Two of the four quadrant contain mostly black pixels which require no computational effort. This leads to it being almost the same speed as the one with 2 processes, as the the additional processes are doing almost none of the work. This issue continues with the high processes as well but gets a bit better for 16 as there we finally divide some of the big chunks of work. With smarter allocation which requires prior knowledge of the shape of the set we could code this to be much more efficient.

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

The weak scaling shows what one expects. The more processes you add the bigger the overheads are that come with it and the lower the efficiency goes. When each processor has its own node


Performance of MPI Mandelbrot set computation
Local computation time (ms)
MPI Process (rank)
MPI Size = 1 process
MPI Size = 2 processes
MPI Size = 4 processes
MPI Size = 8 processes
MPI Size = 16 processes
400000
350000
300000
250000
200000
150000
100000
50000
0
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15


64 nodes
One node
Efficiency
Number of threads
1.0
0.8
0.6
0.4
0.2
2^0 2^1 2^2 2^3 2^4 2^5 2^6



the graph linearly decreases much as we would expect. However when we put all the processors on one node the efficiency plateaus for a bit. One node managing two processors is quite efficient but the moment it gets four the communication overhead is so huge the efficiency tanks by a lot. Adding more processors now does not reduce the efficiency as it was already so bad before and the additional work done by the processor offsets its added communication overhead. This is why it plateaus. After 32 processes the trend continues as expected and the efficiency starts getting lower again.

The strong scaling for the multi processor architecture looks as expected. The speedup always increases by around 1.5x which makes sense as it has to be lower than 2 due to overheads. When running on a single node the overheads are higher which means the speedup is just a bit less for every value. It follows the same trend though. In this case I only had time to run it with 1000 as the matrix size as my program crashed when running with 10k. The scaling is probably a lot worse when the problem size is higher. This would mean that at some point it is no longer worth it to add processors and the speedup would get worse. This should happen to One node first as it should scale worse with its bigger overheads.