INSTITUTE OF MICROELECTRONICS

NSSC2 - Exercise 1

Group ?

Member:
Christian Gollmann, 01435044
Peter Holzner, 01426733
Alexander Leitner, 01525882

Submission: April 19, 2020

Contents

1 Task 1: Questions

(a) Describe the advantages/disadvantages of a two-dimensional decomposition (over a one-dimensional decomposition).

1D	2D
Easier to implement	Less ghost points (same number of sub domains)
Less neighbors per sub domain	More adaptable (later: adaptive meshing)
Simpler MPI-Communication structure	MPI-Communication broken up into 2 stages

Table 1: 1D- vs 2D-decomposition

(b) Using a ghost layer based decomposition, how could multiple independent Jacobi iterations be achieved before communication has to happen?

Idea 1: One could split the iteration space into two different stages, a warm-up and a refinement stage. During the warm-up stage, one proceeds as normal and updates the ghost points after every iteration. The solver is transitioned into the refinement stage once the current approximation $\bar{u}_h$ is either sufficiently close to the analytical solution if known (small total error), solves the system of equations sufficiently well (small residual) or is sufficiently converged, i.e. $||\bar{u}_{h,i+1} - \bar{u}_{h,i}||$ is small. The idea here is to treat the ghost points as inhomogeneous Dirichlet boundary points for the sub domain, at least for a number of iterations, because the small changes will not make much of a difference for the rest of the sub domain. One could even repeat this two stage process more often and switch between warm-up and refinement stage a couple of times.

Idea 2: Instead of using sharp boundaries, i.e. non-overlapping sub domains, in the form of 1-layer ghost points, one could instead define boundary regions that span multiple layers of ghost points, i.e. overlapping sub domains. One then distinguishes between outer ghost points and the "inner" ghost points which represent the overlapping grid points . In our case, since the stencil only uses direct neighbors, the outer ghost points really only are the outermost layer. The Jacobi iterations are then also done on these inner ghost points. After a certain number of iterations, the inter-process communication is initiated. The outer ghost points are communicated as usual while the inner ghost points will have to be treated differently as more than one process will have a suggestion on what their values should be. Here, a consensus has to be reached somehow, e.g. some sort of averaging procedure, and the consensus values have to communicated to all relevant processes/sub domains.

(c) Describe the conceptual differences between an hybrid OpenMP/MPI-parallelization over a pure MPI-parallelization.

In a pure MPI-parallelization, there are no shared variables. Every communication between processes has to be done over messages. OpenMP makes use of shared variables and "local" (N)UMA architecture.

In the pure MPI model, the domain is decompositioned into multiple sub domains which are each assigned to one process. The processes are then assigned to only one thread/core each which works on its task sequentially. The underlying assumptions here is that we have individual UMA (or NUMA) compute units with no form of (or not used here) multiprocessing architecture themselves that are connected via a message passing interface.

The hybrid OMP/MPI-model tries to take advantage of each compute units local multiprocessing architecture by enabling multiple local processing units to work on one of the sub domains via some sort of work sharing construct or such.

On the IUE-Cluster (12 nodes with 2 multicore processors) a pure MPI implementation could, for example, assign each core of a node to work on a sub domain. It thereby completely ignores the multiprocessing capabilities of a NUMA-node and its UMA-processors. In contrast, a hybrid approach might assign each processor of a node with a sub domain and allow each processor to use its multicore architecture via OMP work sharing constructs to do work on the assigned sub domain.

(d) How big is the sum of all L2 caches for 2 nodes of the IUE-cluster?

The IUE-Cluster consists of 12 compute nodes which each consists of two Intel Xeon Gold 6248 multicore processors. We used the information on the Intel Cascade Lake Architecture[1] and Intel Xeon Gold 6248 processor[2]. According to our second source, a Intel Xeon Gold 6248 processor has 20MiB[3] of L2-cache (1 MiB per core). 2 nodes of the IUE-cluster therefore have 80 MiB of L2-cache in total.

2 Task 2: 1D Decomposition

2.1 Description

Your task is to implement a one-dimensional decomposition using ghost layers and MPI-communication to update the ghost layers. Create a program which is callable by

```
mpirun −n NUMMPIPROC ./jacobiMPI resolution iterations
```

e.g.,

```
mpirun −n 4 ./jacobiMPI 250 30,
```

where NUMMPIPROC is the number of MPI-processes to launch, resolution defines the grid spacing as $h = \frac{1}{resolution-1}$, and iterations defines the number of Jacobi iterations to perform. Further and more specifically, your program should

- use $\bar{u}_h = 0$ as initial approximation to u, and (after finishing all iterations)
- print the Euclidean $||\cdot||_2$ and Maximum $||\cdot||_\infty$ norm of the residual $||A_h\bar{u}_h - b_h||$ and of the total error $||\bar{u}_h - u_p||$ to the console, and
- print the average runtime per iteration to the console, and
- produce the same results as a serial run.

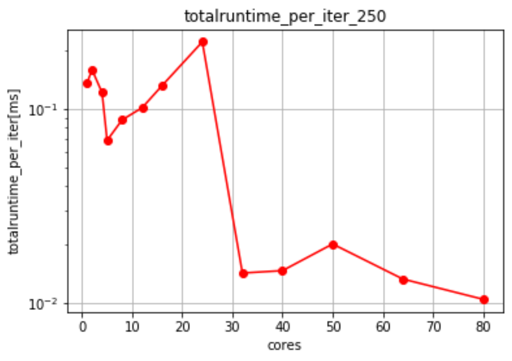
Finally, benchmark the parallel performance of your program jacobiMPI using 2 nodes of the IUE-Cluster for 4 different $resolutions = \{250, 1000, 4000, 8000\}$ using between 1 and 80 MPI-processes (NUMMPIPROC). More specifically, you should

- create a plot of the parallel speed and a plot of the parallel efficiency for each resolution, and
- discuss the results in detail

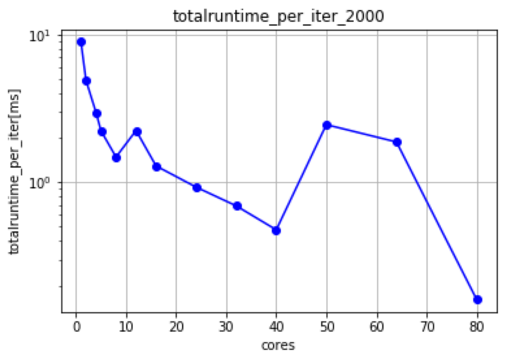
[1]`https://en.wikichip.org/wiki/intel/microarchitectures/cascade_lake#Memory_Hierarchy`
[2]`https://en.wikichip.org/wiki/intel/xeon_gold/6248`
[3]MebiBytes= 2^{20} bytes, see https://en.wikichip.org/wiki/mebibyte

2.2 Results

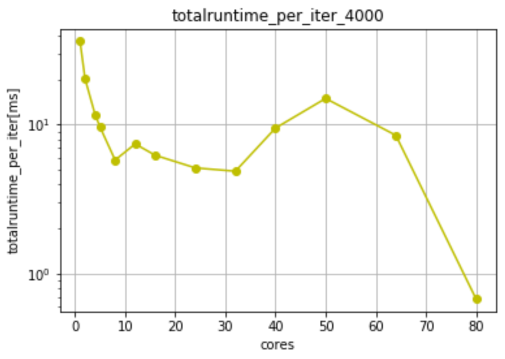
In figure 1 one can see the average runtime per iteration for different grid resolutions in dependence of different number of processes used. Based on the data visualized above we calculated the parallel



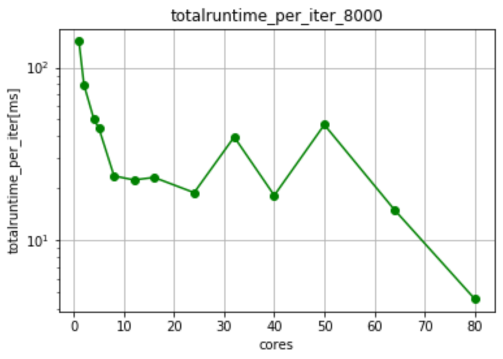
(a)



(b)



(c)



(d)

Figure 1: Runtimes for different grid sizes

speedup according to

$$Speedup\ S_n = \frac{Time\ of\ serial\ run\ T_1}{Time\ of\ parallel\ run\ with\ n\ processes\ T_n} \tag{1}$$

and the parallel efficiency according to

$$Parallel\ efficiency\ E_n = \frac{S_n}{n}. \tag{2}$$

The Speedup and the Efficiency are then also plotted, see 2 and 3 respectively.

As can be seen in 1, our implementation did achieve a speedup compared to the serial/1-process implementation for all four resolutions. There are however a few kinks and irregularities.

In 1 (a), for two core counts, the MPI-parallel implementation was actually slower than the serial reference. Since we did not observe a similar behaviour for different resolutions, we think this might have been caused by problems/irregularities in the node of the cluster.

In 1 (b) to (d), the runtimes do not (strictly) scale with increasing core/process counts. The jump, that is present in every plot when going from 40 to 50 processes, is most likely due to the fact that one node of the cluster consists of 40 cores in total. Therefore, a second node has to be used to be able to use 50 processes or more. The inter-node communication network is most likely much slower than the intra-node communication which is then reflected in the runtime plots. Across the board, we witnessed a significant speedup when engaging both nodes fully (80 processes/cores).

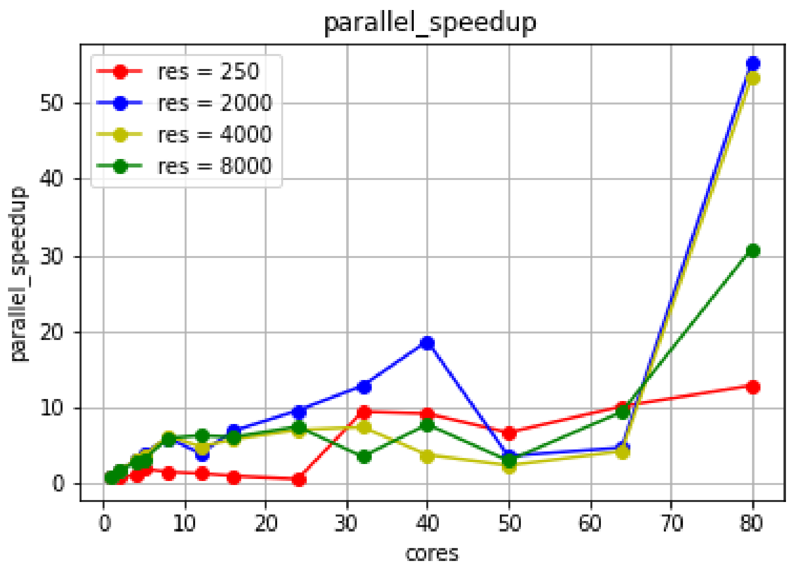


Figure 2: Parallel Speedup S_n plotted for 4 different resolutions

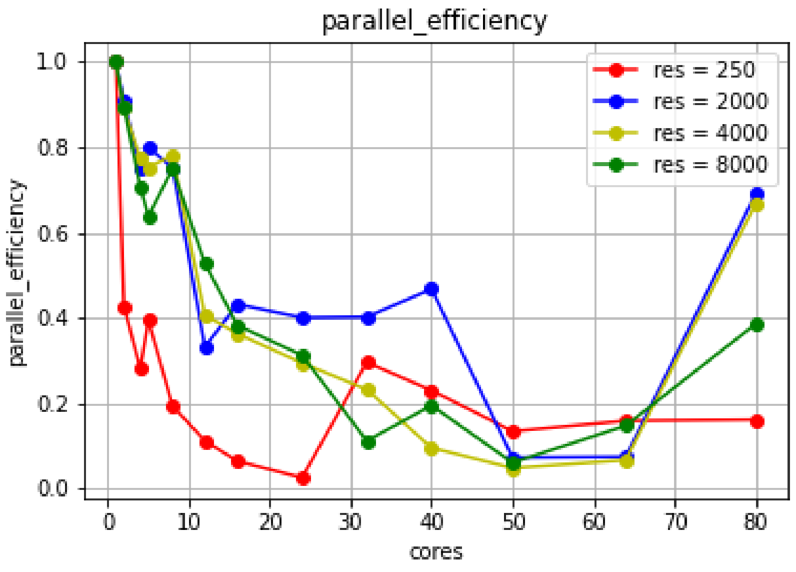


Figure 3: Parallel Efficiency E_n plotted for 4 different resolutions

3 Task 3: Single-Precision Data Representation

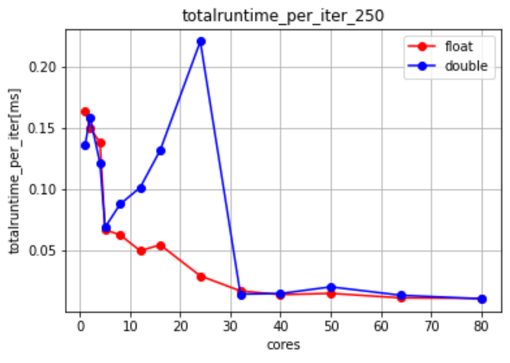
3.1 Description

Adopt your program jacobiMPI from above by changing the underlying floating point data type of the discretization from double to float. Specifically,

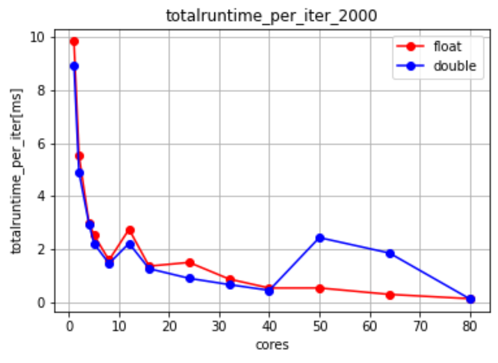
- ensure a correct implementation (by comparing results to a serial run), and
- compare the results to the results from Task 2 in a suitable plot and discuss your results.

3.2 Results

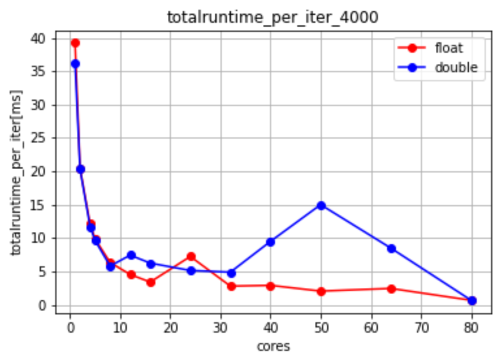
We looked at Task 3 from two different points of view; difference in runtime and difference in precision. Let's first discuss the difference in runtime. In figure 4 (a)-(d) one can see the runtimes for double- and single-precision implementations. One notices that there seem to be some irregularities in the double graphs. The possible reason for that was already pointed out in the discussion of Task 2. Apart from that we did not encounter much difference between the two curves even though we expected the runtime of the float implementation to be lower due to reduced precision.



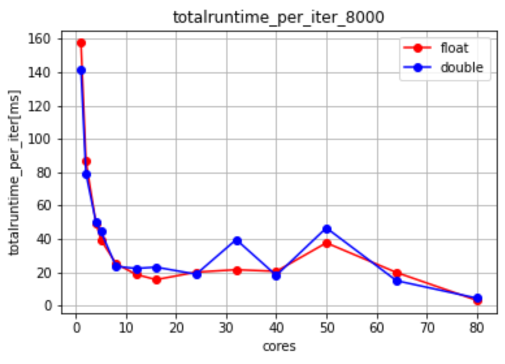
(a)



(b)



(c)



(d)

Figure 4: Comparison between runtimes of double and float numbers

However, this difference only becomes noticeable for a sufficiently big resolution. When looking at 5 one notices that the errors for float and double precision overlap exactly. As we know from the NSSC1 lecture, the total error should be bounded by $||\bar{u}_h - u_p|| \leq Ch^2$. The same bounding holds for the residual. For this resolution, the single-precision data representation is accurate enough.

The smaller residual for the double implementation simply means that it can solve the system of equations more precisely which doesn't necessarily translate to a smaller total error.

The total errors (both for the 2- and the max-norm) become noticeably different for bigger resolutions, e.g. in 7 and 8. Here, the double representation clearly outperforms the float implementation.

One reason, why there is such a big difference between float and double with respect to the residual could be that to calculate the residual, one essentially applies the stencil in a similar way as during one Jacobi iteration. This leaves room for round off errors which are more severe for the float representation due to the lower precision.

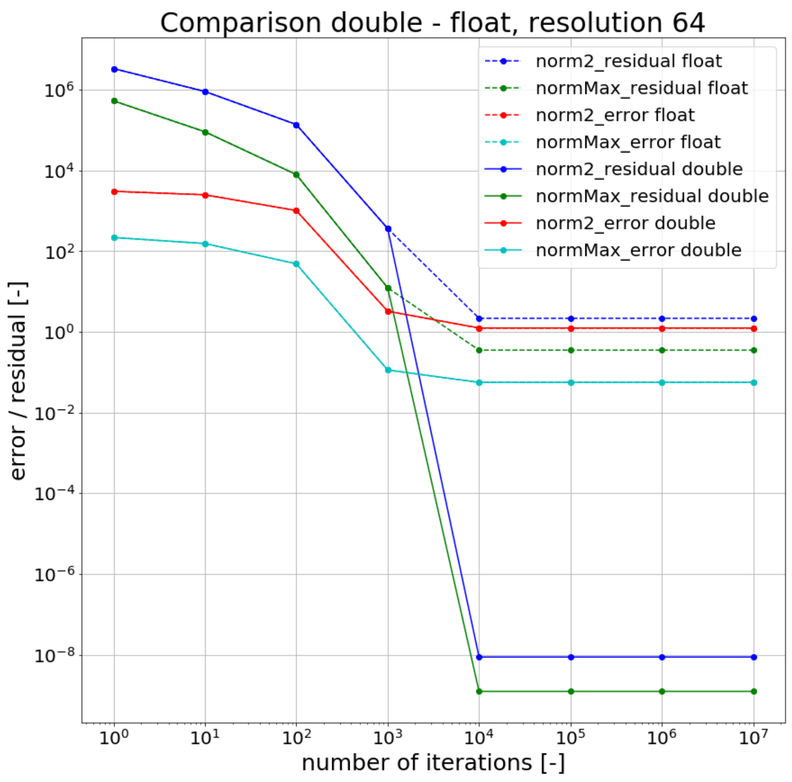


Figure 5: Comparison between double and float for resolution 64

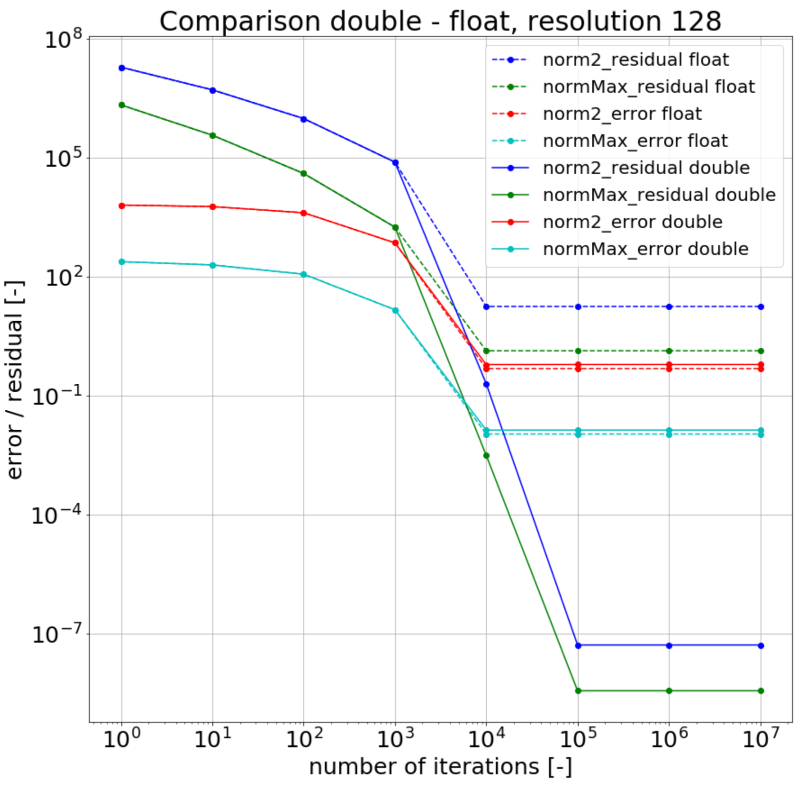


Figure 6: Comparison between double and float for resolution 128

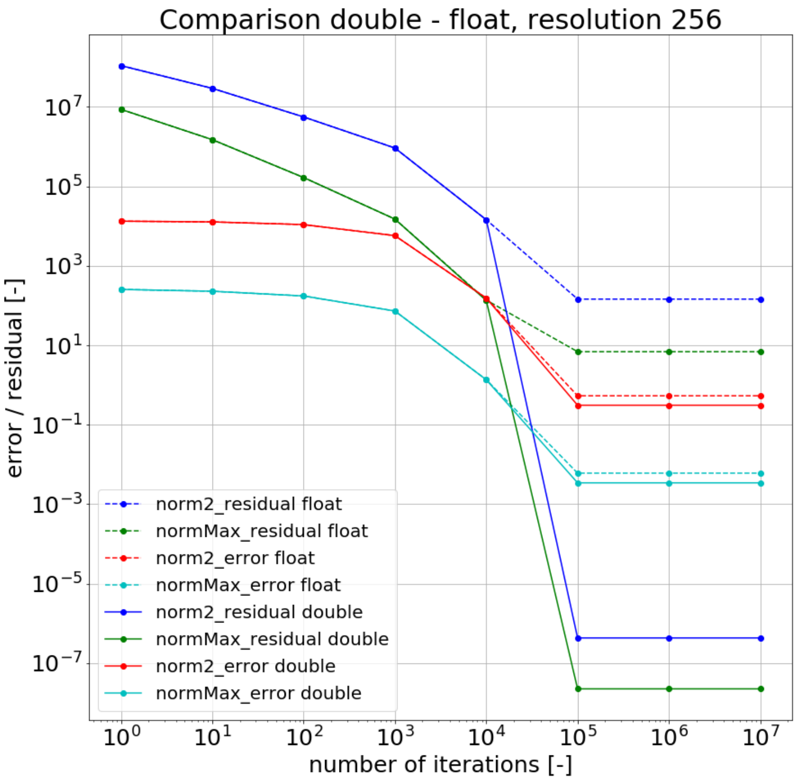


Figure 7: Comparison between double and float for resolution 256

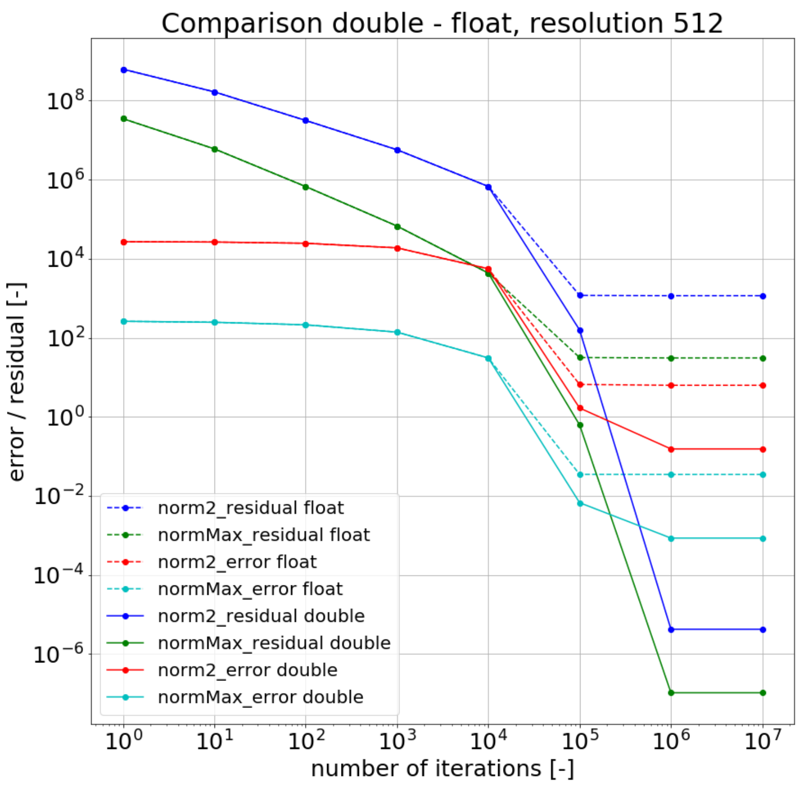


Figure 8: Comparison between double and float for resolution 512

4 Task 4: 2D Decomposition

4.1 Description

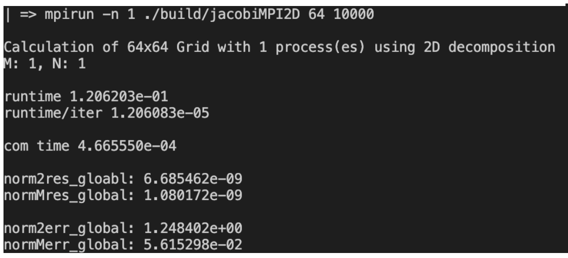
Extend your program from Task 2 by implementing a two-dimensional decomposition using a ghost layer and MPI-communication to update the ghost layers. Create a program which is callable by

```
mpirun −n NUMMPIPROC . / jacobiMPI2D resolution iterations ,
```

where the command parameters have the same meaning as above. Ensure a correct implementation by comparing your results to a serial run. Benchmarking on the cluster is not required.

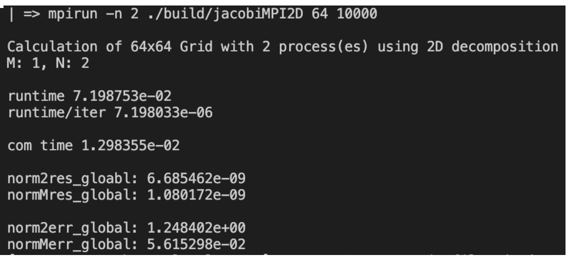
4.2 Results

We tested our implementation using 2D decompositions on our local machines and concluded that our implementation works since it produces the same errors and residuals as a serial reference when using the same resolution and number of iterations (see figure 9).



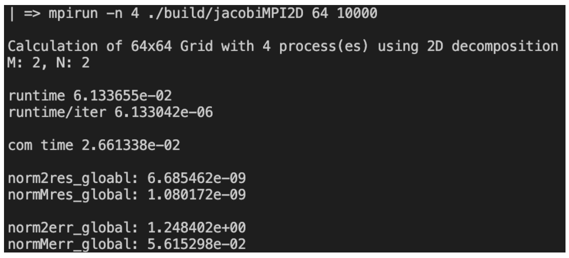
```
| => mpirun -n 1 ./build/jacobiMPI2D 64 10000

Calculation of 64x64 Grid with 1 process(es) using 2D decomposition
M: 1, N: 1

runtime 1.206203e-01
runtime/iter 1.206083e-05

com time 4.665550e-04

norm2res_gloabl: 6.685462e-09
normMres_global: 1.080172e-09

norm2err_global: 1.248402e+00
normMerr_global: 5.615298e-02
```

(a) Serial reference



```
| => mpirun -n 2 ./build/jacobiMPI2D 64 10000

Calculation of 64x64 Grid with 2 process(es) using 2D decomposition
M: 1, N: 2

runtime 7.198753e-02
runtime/iter 7.198033e-06

com time 1.298355e-02

norm2res_gloabl: 6.685462e-09
normMres_global: 1.080172e-09

norm2err_global: 1.248402e+00
normMerr_global: 5.615298e-02
```

(b) 2 processes (=1D decomposition)



```
| => mpirun -n 4 ./build/jacobiMPI2D 64 10000

Calculation of 64x64 Grid with 4 process(es) using 2D decomposition
M: 2, N: 2

runtime 6.133655e-02
runtime/iter 6.133042e-06

com time 2.661338e-02

norm2res_gloabl: 6.685462e-09
normMres_global: 1.080172e-09

norm2err_global: 1.248402e+00
normMerr_global: 5.615298e-02
```

(c) 4 processes (true 2D decomposition)

Figure 9: Output of jacobiMPI2D vs serial reference