Practical Parallel Programming, FIT, BUT

Project: Heat Propagation

David Mihola
xmihol00

26th of April, 2024

Implementation Details

There are two implementations available. First, developed at the beginning of the semester without knowledge about all possibilities of the MPI library and later slightly modified, e.g. by data scatter/gather using user defined MPI data types. Second, developed further into the semester, which better utilizes user defined MPI data types. The main difference between the implementations is that the first uses separate memory location for halo zones, while the second stores them within the temperature tiles. Although the second implementation makes the code shorter and easier to read, it is unsafe to use, because there are concurrent writes to a memory window by the owning process and other processes via RMA[1]. Fixing this issue, i.e. writing with RMA into a separate buffer and then unpacking it to a temperature tile by the owning process, would cause quite a significant communication overhead. Both implementations perform almost identically, therefore, given the aforementioned issue of the second implementation, the first one is chosen as the primary one[2].

Performance Analysis

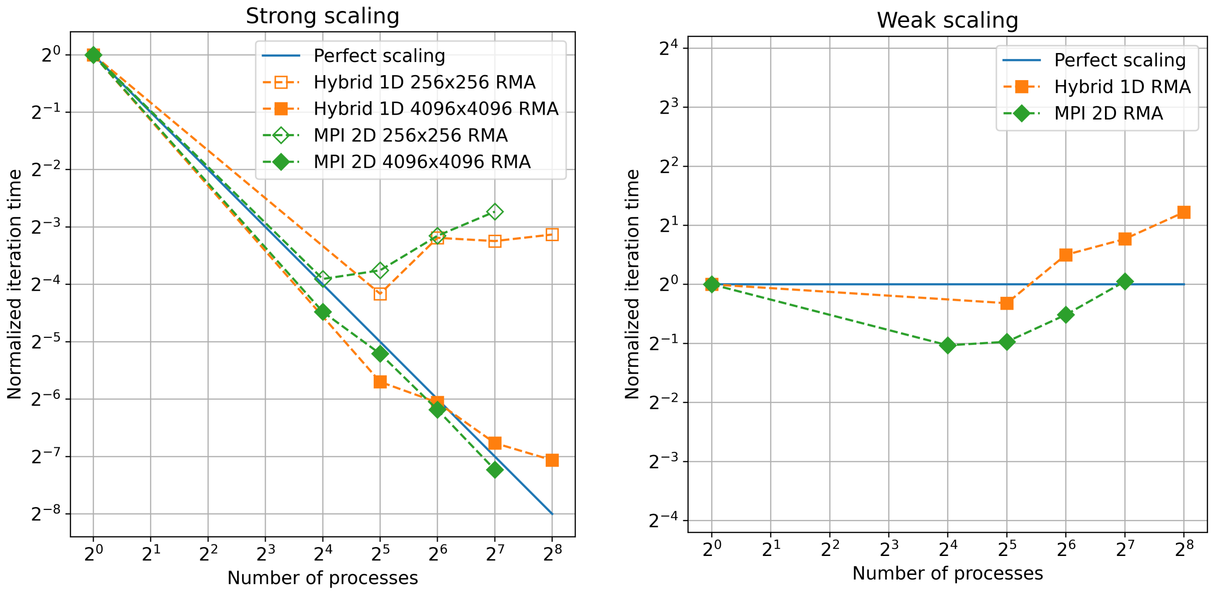


Figure 1: Graphs of strong and weak scaling

The graph of strong scaling in Figure 1 depicts the measured best and worst case scenarios for the 1D and 2D domain decompositions. Scaling on the large domain is often better than what is the theoretical limit[3], especially for the 2D decomposition, while scaling on the small domain quickly gets into diminishing returns because of a disproportional amount of communication to computation. We can also determine the speedup and efficiency on the two chosen domains from this graph. Better than theoretical strong scaling leads to super-linear speed up and above 100 % efficiency. On the

[1]However, all tests pass, because the writes are not overlapping and to different cache lines.

[2]See the `DATA_TYPE_EXCHANGE` macro in the `ParallelHeatSolver.hpp` and `ParallelHeatSolver.cpp`.

[3]In both cases most likely enabled by better cache access pattern.

other hand, getting into the state of diminishing returns causes a significant drop in efficiency from almost 100 % with 32/16 processes to below 50 % with 64/32 and more processes respectively for the 1D and 2D domain decompositions. The speedup is therefore sub-linear and one could even say sub-constant, i.e. adding processes slows down the computation.
The graph of weak scaling in Figure 1 again compares the two possible domain decompositions. Here, the measured values are not only normalized by the iteration time of the sequential solution, but also to reflect the quadratic computational complexity determined by the domain edge size, as well as the inconsistent increase between the domain size and number of processes[4]. Both decompositions initially scale better than the theoretical limit, but deteriorate with increasing domain size and number of processes.
The reasons why there are differences in scaling and therefore speedup/efficiency between the 1D and 2D decompositions are the following:

1. Given P processes and a domain with edge size of N, the number of values computed and exchanged by each process in the halo zones is proportional to:

   - $2N$ for the 1D decomposition,
   - $4N/\sqrt{P}$ for the 2D decomposition.

   Meaning that with increasing number of processes the amount of exchanged data decreases for the 2D decomposition, while it stays the same for the 1D decomposition, possibly causing larger communication overheads.

2. The 1D decomposition is run in a hybrid setting, where some processes are actually just threads only active for the computation of the temperature tile. Meaning that the computation of halo zones is not fully parallelized which causes worse scaling, see Figure 3, and furthermore, it is more likely that the communication will take longer than the fully parallelized computation of the tile in this setting, i.e. communication will not be fully overlapped by computation producing further otherwise hidden delays.

Consequently, the 2D decomposition should be preferred if unequal computation loads, see Figure 4, are not an issue, i.e. all edge nodes in the 2D decomposition do not have at least one neighbour, whereas only two nodes are without their second neighbour in the 1D decomposition.

I/O

Naive use of parallel I/O can very easily, as is shown in Figure 2, cause diminishing returns. The written amount of data, HDF5 file alignment and Lustre file system stripe size must all be in sync to outperform the sequential I/O. And even if that is the case, the gains are not very significant in the case of our example with only relatively small amounts of written data (1 MiB per process). In order to really take advantage of the parallel I/O the amount of written data would have to be in orders of magnitude larger.

[4]See `plot_weak_scaling.py`.

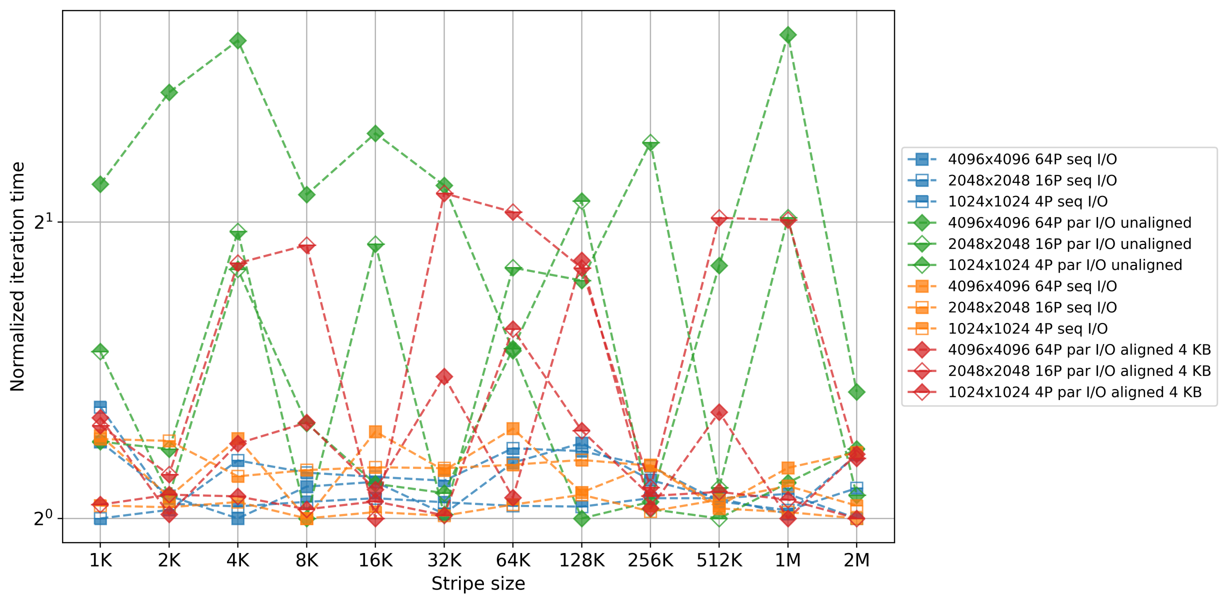


Figure 2: Comparison of different domain sizes, HDF5 alignments and Lustre file system stripe sizes with stripe count always set to 16

Supporting materials

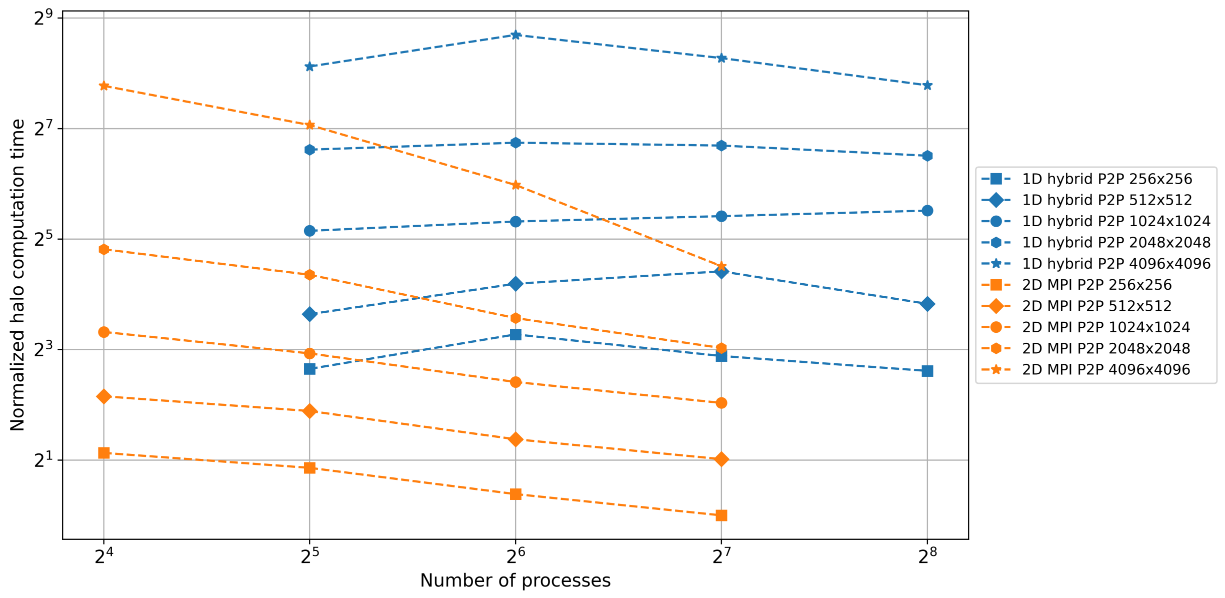
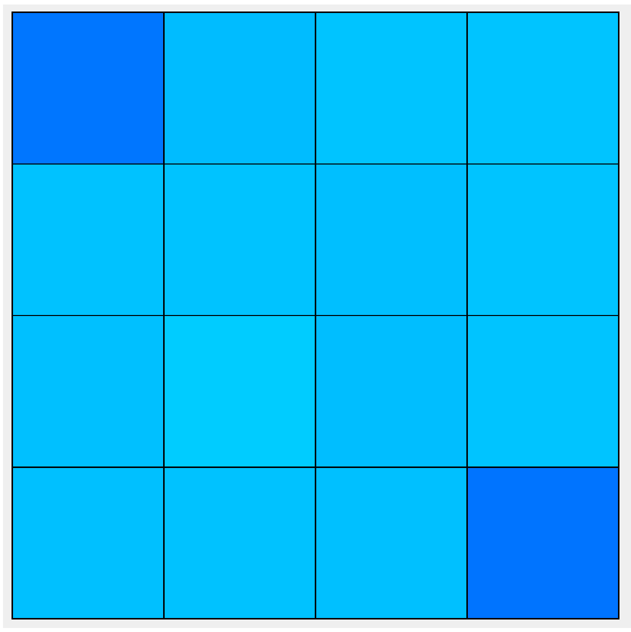
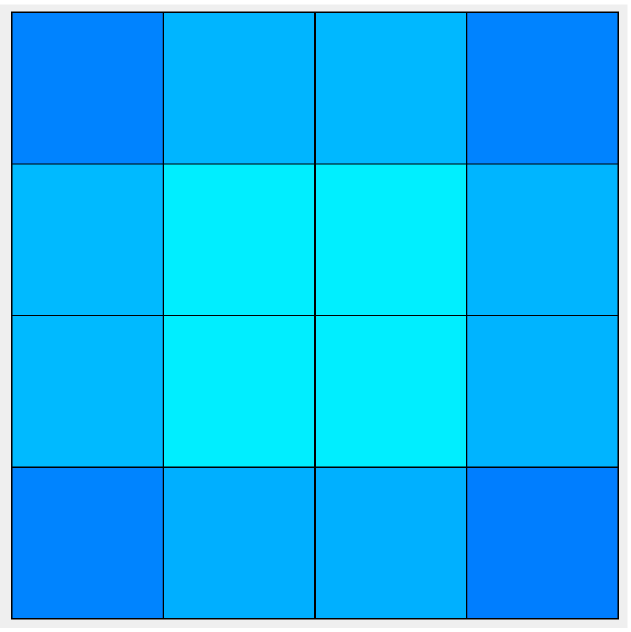


Figure 3: Comparison of halo zone computation times of the 1D and 2D decompositions
We can see a clear trend, where the computation times of the 2D decomposition are decreasing with additional processes, while they remain rather constant with the 1D decomposition. (See the `MEASURE_HALO_ZONE_COMPUTATION_TIME` macro in the `ParallelHeatSolver.hpp` and `ParallelHeatSolver.cpp` how the measurement was performed.)

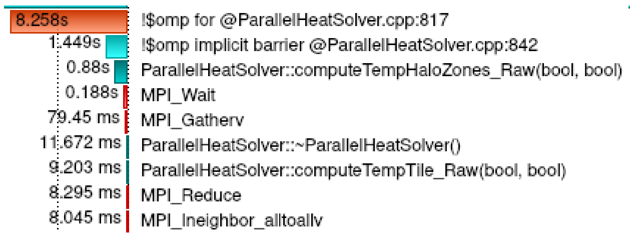


(a) 1D decomposition (wrapped)



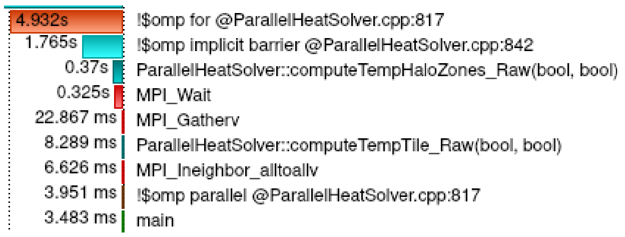
(b) 2D decomposition

Figure 4: Comparison of process loads during halo zone computation visualized in Cube
Darker color means less loaded. Expectedly, the computation load of the 1D decomposition is more balanced.



```
8.258s     !$omp for @ParallelHeatSolver.cpp:817
1.449s     !$omp implicit barrier @ParallelHeatSolver.cpp:842
0.88s      ParallelHeatSolver::computeTempHaloZones_Raw(bool, bool)
0.188s     MPI_Wait
79.45 ms   MPI_Gatherv
11.672 ms  ParallelHeatSolver::~ParallelHeatSolver()
9.203 ms   ParallelHeatSolver::computeTempTile_Raw(bool, bool)
8.295 ms   MPI_Reduce
8.045 ms   MPI_Ineighbor_alltoallv
```

(a) 1D decomposition



```
4.932s     !$omp for @ParallelHeatSolver.cpp:817
1.765s     !$omp implicit barrier @ParallelHeatSolver.cpp:842
0.37s      ParallelHeatSolver::computeTempHaloZones_Raw(bool, bool)
0.325s     MPI_Wait
22.867 ms  MPI_Gatherv
8.289 ms   ParallelHeatSolver::computeTempTile_Raw(bool, bool)
6.626 ms   MPI_Ineighbor_alltoallv
3.951 ms   !$omp parallel @ParallelHeatSolver.cpp:817
3.483 ms   main
```

(b) 2D decomposition

Figure 5: Comparison of computation times visualized in Vampir