HPC Lab

Assignment 3

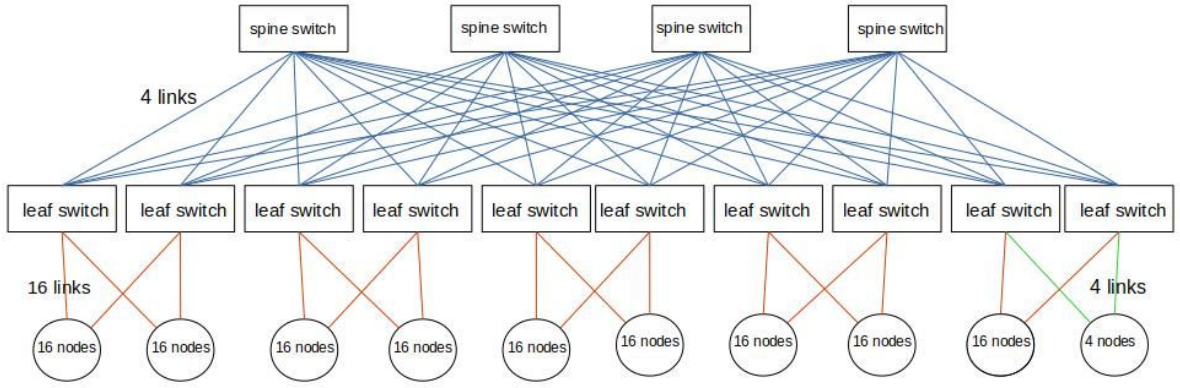
Erkin Kirdan - Manuel Rothenberg - Gabrielle Poerwawinata

1. Amdahl’s Law

1. For an efficiency of 70%, we get Sp = 0.7*p. According the Amdahl’s Law , with 10% of sequential portion, we obtain p around 5.28571. So, the maximum number of processes p is 5 which leads to efficiency of 71%.
2. Gustafson's law addresses to the weaknesses of Amdahl's law, which depends on the assumption of a fixed problem size, that is of an execution workload that does not change as for the change of the resources. Gustafson's law rather suggests that developers tend to set the size of problems to completely exploit the processing power that becomes available as the resources improve. Hence, if faster hardware is available, larger problems can be comprehended within the same time.

2. Probing the Network

1. The bandwidth depends on the message size because Intel Omni Path uses PSM which do high message rates especially on small messages sizes. The bandwidth that we can expect is Intel OPA operates at 100 Gbps, providing 2.5X the bandwidth of 40 Gbps Intel True Scale Fabric solutions. Intel OPA also delivers extremely low latency that stays low at scale (100 to 110 ns per port).
2.



Note:
Each of the leaf switch has 48 ports which 32 ports will make a links to the nodes and the rest of 16 ports goes to the spine switch. The blue line represent the unit of 4 links from leaf switch to each of the spine switch. Each of the leaf switch will be connected to the 16 nodes, where each of these nodes has 2 ports connected to the leaf switch. The red line represent the unit of 16 links from 16 ports from 16 nodes to the leaf switch. There is 148 nodes in total therefore there will be 2 leaf switches that only have 20 nodes instead of 32 connected to them.

3. Bisection width:
To separate the network into 2 equal parts, we can make 2 groups of spine switch. Therefore the number of links that have to be removed is:
8 links * 10 leaf switches = 80 links
16 links * 2 leaf nodes = 32 links
The total of links removed is 112 links

Bisection bandwidth:
The sum of the single bandwidth of all edges that are cut by bisecting the network is:
112 links*100Gbit = 11200Gbit

4. Key features of Intel MPI Benchmark consists of the following components

- **IMB-MPI1** - benchmarks for MPI-1 functions, has class of:
    - Single Transfer Benchmarks : involve 2 active processes into communication. Other processes wait for the communication completion. Throughput values are measured in MBps and calculated as:
      throughput = X/time
      where,
        - time is measured in μ sec.
        - X is the length of a message, in bytes
    - Parallel transfer benchmarks involve more than 2 active processes into communication.
      Throughput values are measured in MBps and can be calculated as follows:
      throughput = nmsg*X/time
      where,
        - time is measured in μsec.
        - X is the length of a message, in bytes.
        - Nmsg is number of messages outgoing from or incoming to a particular process
    - Collective Benchmarks measure MPI collective operations. Each benchmark is run with varying message lengths. The timing is averaged over multiple samples.
- Two components covering MPI-2 functionality:
    - **IMB-EXT** - one-sided communications benchmarks
    - **IMB-IO** - input/output (I/O) benchmarks
- Two components covering MPI-3 functionality:
    - **IMB-NBC** - non-blocking collectives benchmarks that provide measurements of the computation/communication overlap and of the pure communication time
    - **IMB-RMA** - Remote Memory Access (RMA) benchmarks that use passive target communication to measure one-sided communication

5. Running the benchmark o linux:

host$ mpirun -np <P> IMB-<component> [arguments]

where

- P is the number of processes. P=1 is recommended for all I/O and message passing benchmarks except the single transfer ones.
- <component> is the component-specific suffix that can take **MPI1**, **EXT**, **IO**, **NBC**, and **RMA** values

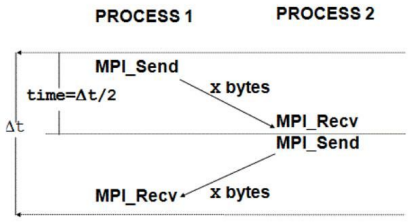
Performance:

- Single Transfer Benchmark, in this case we tried PingPong single transfer class. For single transfer, we need 2 process available therefore we defined -np 2. Here is the following script for single transfer:

```
#!/bin/bash
#SBATCH -o /home/hpc/t1221/lu26xum/%j_%N.txt
#SBATCH -D /home/hpc/t1221/lu26xum
#SBATCH --nodes=32
#SBATCH --mail-user=gw.poerwawinata@tum.de
```

srun mpirun -np 2 IMB-MPI1 PingPong

The figure below shows how process in PingPong class interacting.



The achieved PingPong bandwidth is shown as figure below.

It shows that messages with bigger size does not necessarily resulted in high bandwidth even it shows a rising trend.

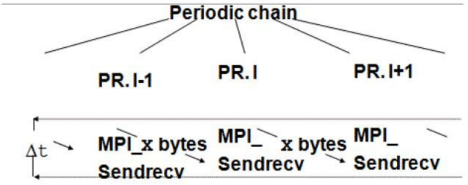
- Parallel and Collective Benchmark, in this case we tried SendRecv parallel transfer and Reduce collective transfer class together. For these class, we need more than 2 process available. It could run with 2 process too, however to test the sending and receiving message from left and right neighbor then it would be better to use more than 2 process. The script is modified as:

  srun mpirun -np 8 IMB-MPI1 SendRecv Reduce

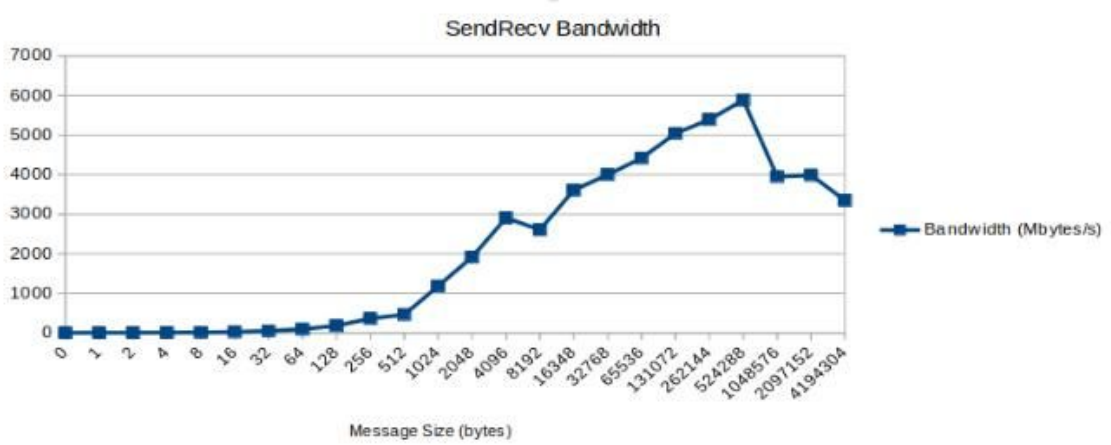
  We choose 8 number of process

  - Parallel Benchmark using SendRecv

  The figure below shows how process in SendRecv class interacting.

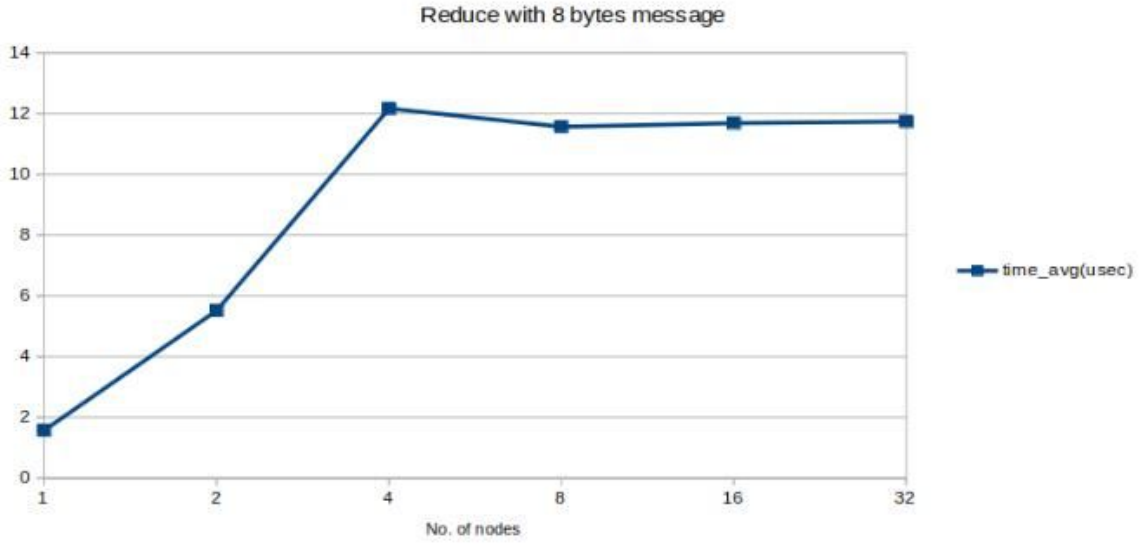


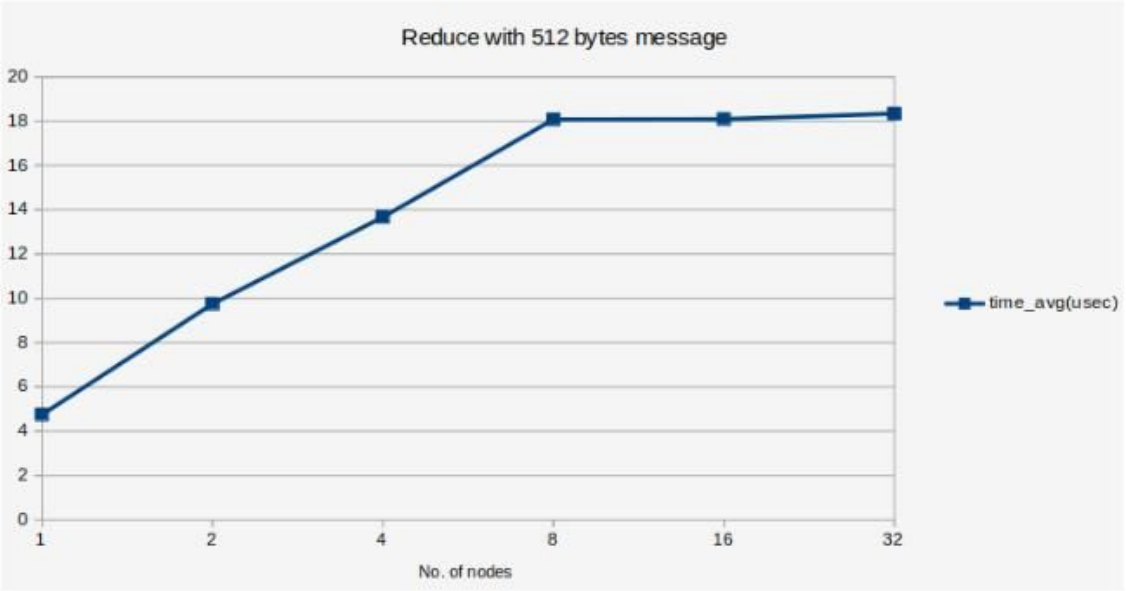
The achieved bandwidth of SendRecv is shown as figure below:



- Collective Benchmark using Reduce

The following figures show the Reduce benchmark performance of 8 bytes and 512 bytes respectively.





From 2 figures above, all the performance curves rise up with the increase of nodes. The more nodes involved in collective communication, the longer time is needed.

3. Broadcast

Comparison of achieved bandwidth our own implementation broadcast with MPI_Bcast, with number of element = 4. Therefore the total number of bytes is 32.

Processor	Our own mpibcast (bytes/s)	MPI_Bcast (bytes/s)
2	17611.45	64777.32
4	53601.34	28906.95
8	10578.51	40712.47
16	13245.03	37514.65
32	13333.33	42895.44

4. Parallel CG

Introduction

Our task was, to parallelize the given iterative conjugate gradient method (CG-method) to solve the Laplacian. The program takes the mesh width as argument and generates a `grid` (initial condition) as well as the right hand side `b` with each the width and height of `grid_points_1d = (1 / mesh_width) + 1`.
In our program the right hand side `b` is initialized with all zeros, because we want to just calculate the Laplacian instead of Poisson.
To calculate the Laplacian, our algorithm uses the following functions:

- `g_product_operator(grid, result)`
- `g_scale_add(dest, src, scalar)`
- `g_scale(grid, scalar)`
- `g_dot_product(grid1, grid2)`

All of the functions operate on the whole grid, but during one call of the functions the calculations are only dependent on one value of the grid at once. That means you can calculate one value completely independent of another value within the grid. An exception of this is the function `g_product_operator(grid, result)` which depends on the neighbor cells (top, bottom, left and right neighbor).
All of the functions also only modify the inner cells, the border cells are not used or modified. The function `g_product_operator(grid, result)` is also exception here, since it doesn't modify the border cells, but uses them for calculations.

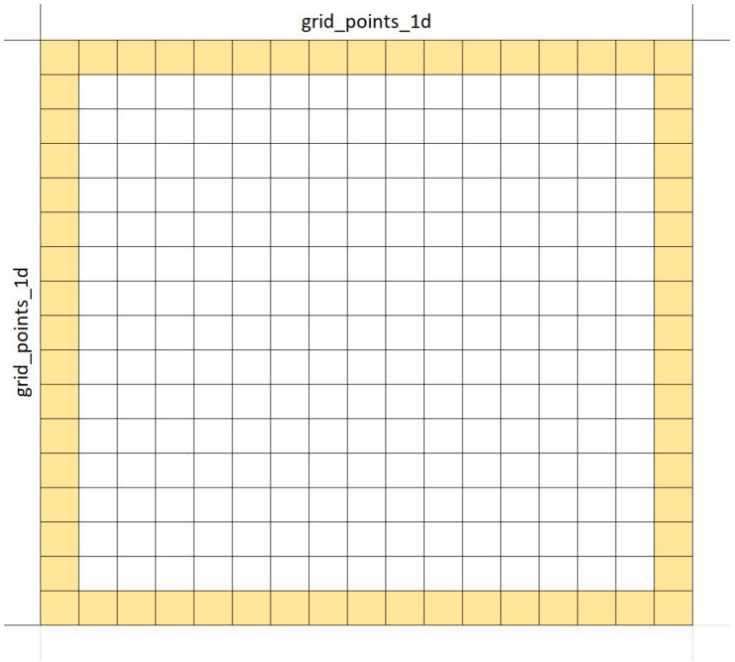


Figure 4.1 (mesh_width = 0.0625 / grid_points_1d = 17)

Figure 4.1 shows an example grid with the mesh_width 0.0625 which results in 17 cells per dimension. The white cells are the inner cells which get modified and the yellow cells are the border cells which are just used for calculations (only in function `g_product_operator`).

Parallelization

To parallelize the given iterative method, we divide the grid into equal-sized patches of the grid. For this, we don't distribute the first and the last column (not modified border cells), we just share `grid_points_1d - 2` columns. Each calculation-task in the parallel implementation then works on one patch of the grid. If we would use for example 5 calculation-tasks for the example grid shown in Figure 4.1 each task would work on `(17-2) / 5 = 3` columns (this is the value `grid_points_x`).
This is done, because during one function call of the used functions (see Introduction) the calculations happen independent and we can distribute the work to several tasks.
It is possible to distribute the work within the function calls, but before and after the function calls it is necessary to use the whole grid and not only a small patch of it. That is the reason, why we use one of the tasks as a controller-task (task with rank 0). This tasks is used, to split the grid and distribute the work to the calculation tasks (rank > 0). It also collects the results of the calculation-tasks, combines the partial results and sends out the final result to all calculation-tasks.

The controller-tasks is the only task which generates the complete grid. The calculation-tasks just allocate storage to store their part of the grid. In our example the calculation-tasks would allocate storage for `(3 + 2) * grid_points_1d` cells. The two extra columns are stored, because within the function `g_product_operator` the neighbor cells are needed for calculations, even if they are not modified. After that, the controller-task as well as the calculation tasks, jump into the `solve()` function. The first step for the controller-tasks is to split the grid and send out the sub-grids to the other tasks. For the calculation-tasks, the first step is to receive their patch of the grid. The controller-tasks sends out the partial grids, and includes the neighboring columns as well. This communication happens with the help of `MPI_send` and `MPI_receive`. As data type we use the help of `MPI_type_vector`, which we configured to hold exactly `(grid_points_x + 2) * grid_points_1d` double values. This new data type is called `myVectorTypeGridWide`.
After the grid is divided and every calculation-task has it's patch, the algorithm works like in the serial implementation, because every task can operate on its own sub-grid. As soon, as all tasks reach a function call of `g_dot_product`, they calculate their part of the dot product and send it to the controller-tasks. The controller-task then sums up all received sub-results and calculates the real and complete result. It then sends out the complete result again to the calculation-tasks, because they need the complete result and not only their own calculated partial result. For the function calls of `g_scale` and `g_scale_add` there is no communication needed, since every calculation task can work on it's own patch, there is no need to distribute these results.
This is not the case for function calls to `g_product_operator`. As soon as all calculation tasks have reached one of these function calls, they send out their first column to their left neighbor as well as their last column to their right neighbor. After a task sends out these

columns it receives the corresponding columns. This is necessary because each calculation-tasks needs the latest calculations of the corresponding border-cells. An exception here are the first and the last patch, because the most-left column as well as the most-right column are not updated by any task. That's why the first and last calculation-task-rank only send out one column instead of two. They also each receive only one column. For this communication we used a new `MPI_type_vector` generated data type which holds exactly one column. This new data type is called `myVectorType1Col`. Since the algorithm works the same for all tasks, it's always ensured, that the communication happens within the same function call. This means when communication gets necessary, all tasks "meet" at this point, share data and continue the work after that.
So far the whole calculation can happen distributed on several tasks. After all of the calculation is done, every calculation-task sends their part of the grid to the controller-task (without border-columns). The controller-task then combines the results into the final and complete grid. This happens with the `MPI_vector_type` generated data type `myVectorTypeGrid`.

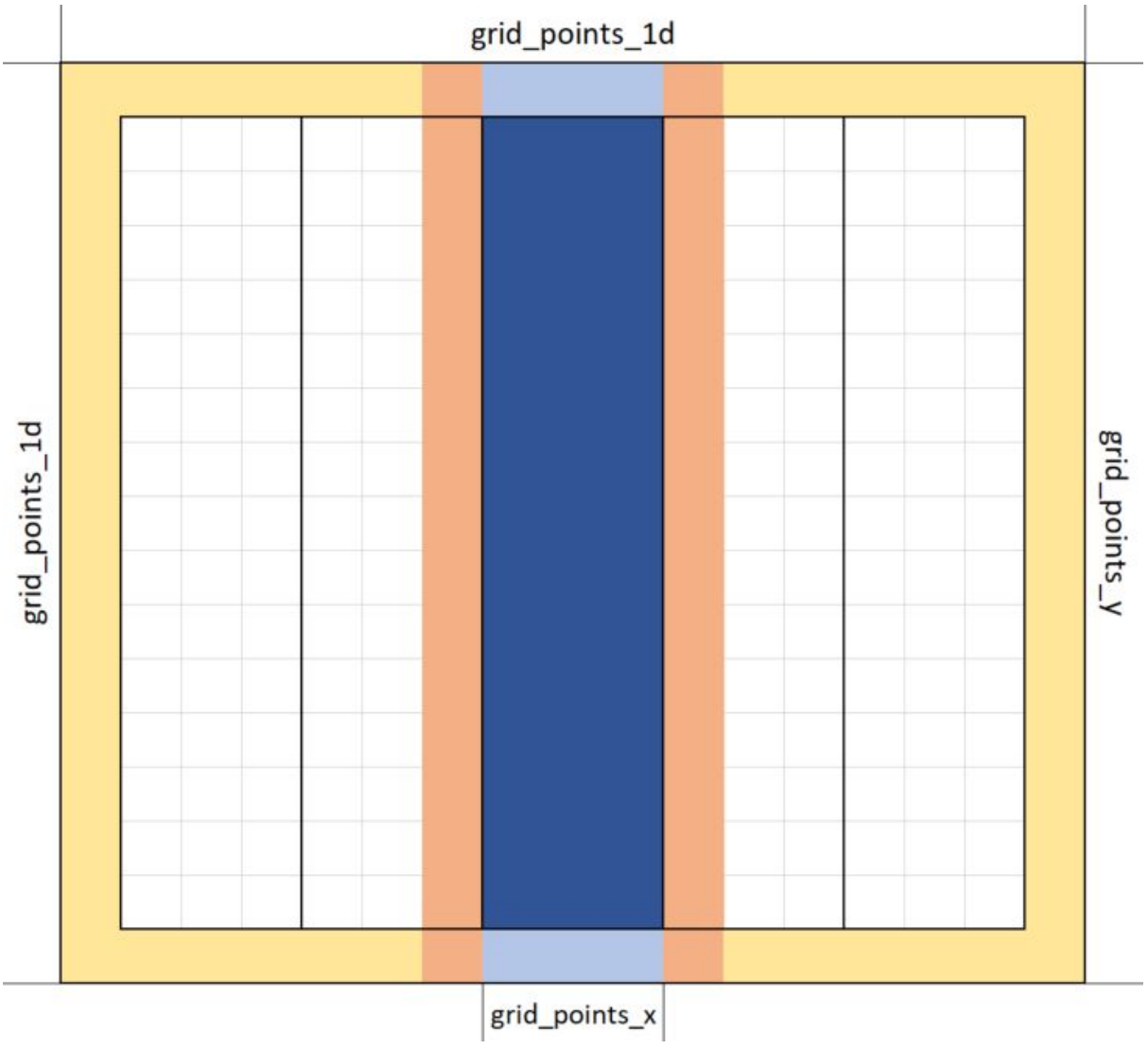


Figure 4.2

Figure 4.2 shows how the example grid is divided into 5 sub grids for one of the calculation-tasks (rank 3). The dark blue part holds the cells which get modified. The orange columns don't get modified by this task, but are received from the neighbor calculation-tasks, to execute the function `g_product_operator` with the latest results.

We tested our program for valid results several times and with a lot of different configurations. A couple of these tests can be seen in the GIT repository in the folder "assignment_3\our_stuff\4\tests". There's always a file init_solution.gnuplot with the results of the iterative implementation as well as a file solution.gnuplot with the results of our implementation. The folder name is the configuration of the execution. nxx shows the number of used tasks.

Configuration

Since for our implementation it is necessary to equally distribute the grid to the calculation-tasks and we also have one task which is used as controller, the following restriction applies:

```
(grid_points_1d - 2) % (size - 1) == 0
```

Size is the total number of tasks.

When we execute the program with mpiexec, we can specify the number of tasks which should work on the problem, as well as the mesh width, which determines also grid_points_1d.
To filter out all valid configurations, which meet the above restriction, we wrote a small script, which shows us just valid configuration of n and mesh width depending on grid_points_1d. This was helpful for the following performance analysis. The valid configurations can be found in the GIT repository in the file "assignment_3\our_stuff\4\config\possible_configs.xlsx" The restriction is a reason for the sometimes odd numbers during the performance analysis.

Performance Analysis

For the performance analysis, we performed several tests with increasing problem size (grid size) as well as increasing tasks on the same problem size. The results can be found in the GIT repository. We started with a problem size of grid_points_1d = 26 and went up to grid_points_1d = 8186. The task range is between 2 and 1024 on up to 16 nodes.

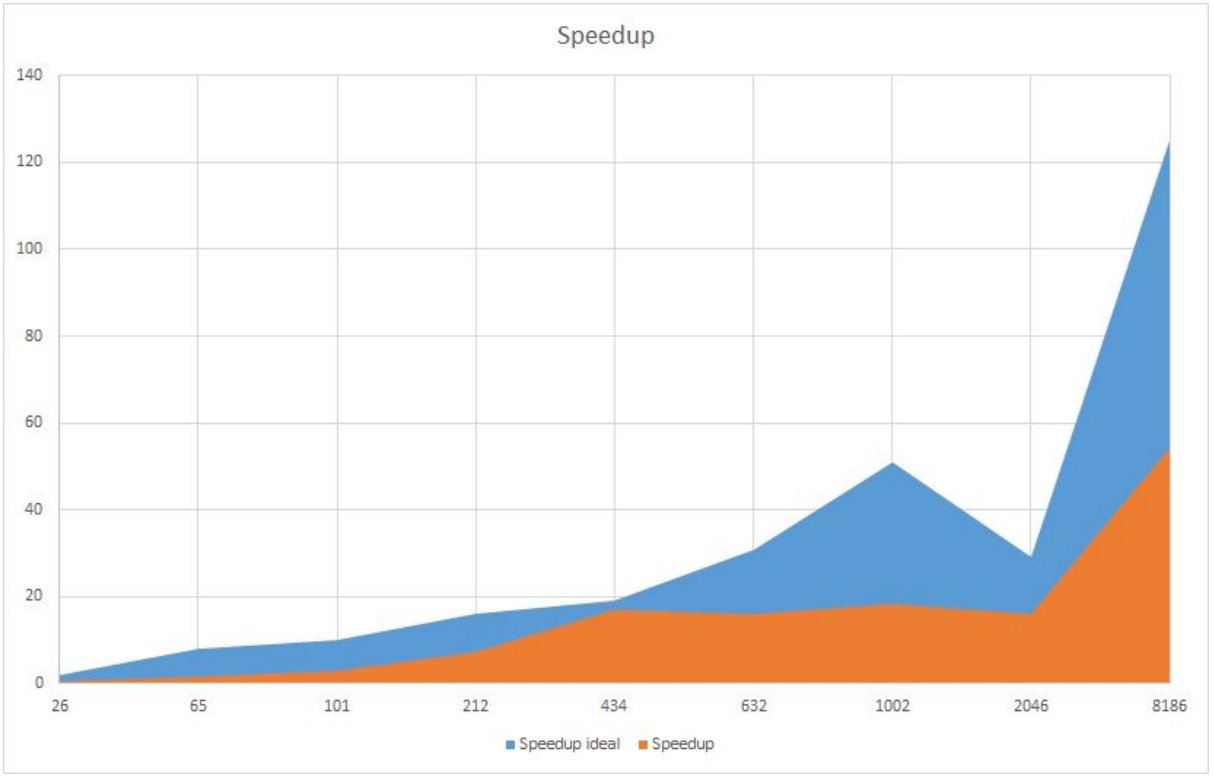


Figure 4.3 - Speedup

For every tested problem size we determined the fastest configuration. Figure 4.3 shows these fastest results. The x-axis shows the increasing problem size, the y-axis shows the speedup regarding the used tasks. In all cases, we couldn’t reach ideal speedup, but that was predictable, since we need both processing power as well as time for the MPI communication. Peak Speedup during our tests was 54.28.

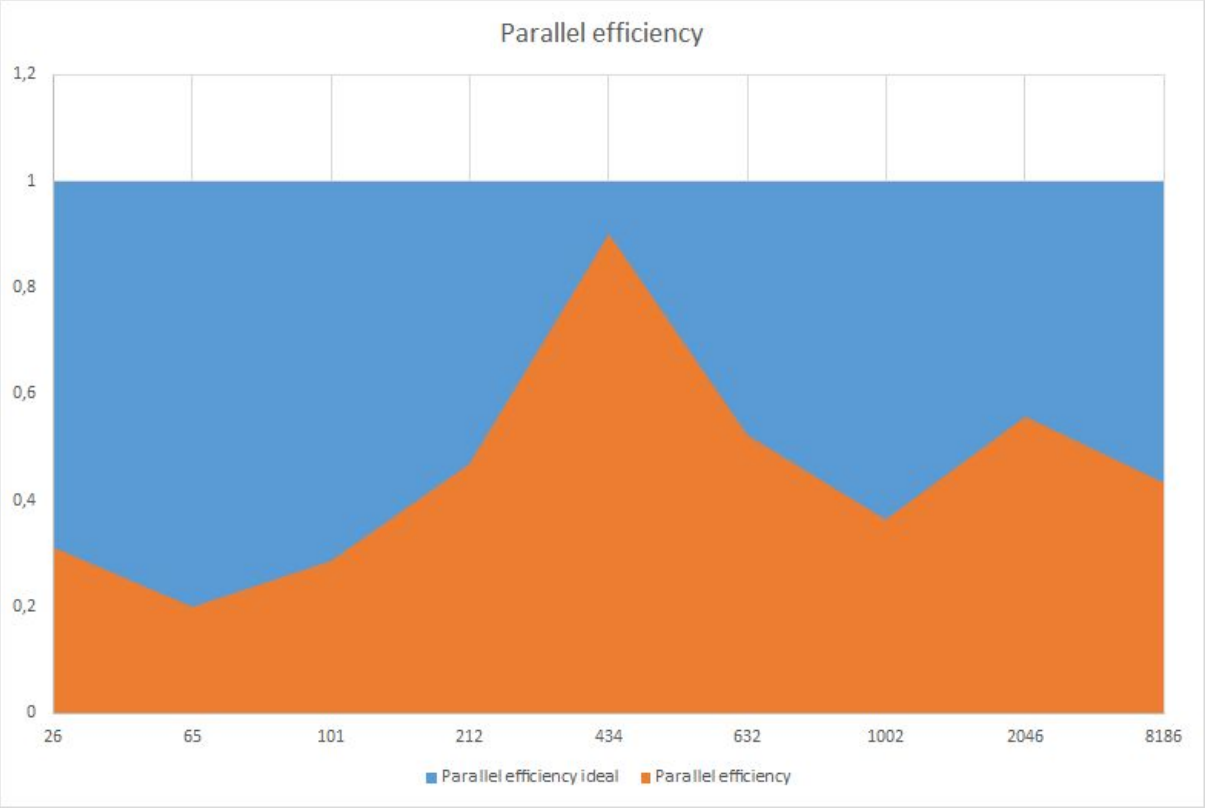


Figure 4.4 - Parallel efficiency

Also for the parallel efficiency we took the best test results for each problem size. Figure 4.4 shows these results. The x-axis shows the growing problem size. The y-axis shows the efficiency. Peak Parallel efficiency during our tests was 0.9.

For all of our tests, in general we can say, that the peak speedup increases with the problem size, this was observable for all tests. We can as well say, that there need to be a balance between the used task number and the necessary communication between these tasks. The more tasks were used, the smaller is the sub-problem of each task which results in more communication overhead.
Usually, the best factor problem_size/tasks was between 0.05 ~ 0.1.