IN4049TU Introduction to High Performance Computing

Lab Report

Philip Soliman (4945255)

February 15, 2024

Contents

0 Intro assignments

The send and receive loop is implemented here.

0.1 Ping Pong

Figures 1 and 2 show the results of the time measurements for several message sizes. The figures also include estimates for the values of α and β. based on the expected relation between message size and communication time

$$T = \alpha + \beta \cdot \text{size} \tag{1}$$

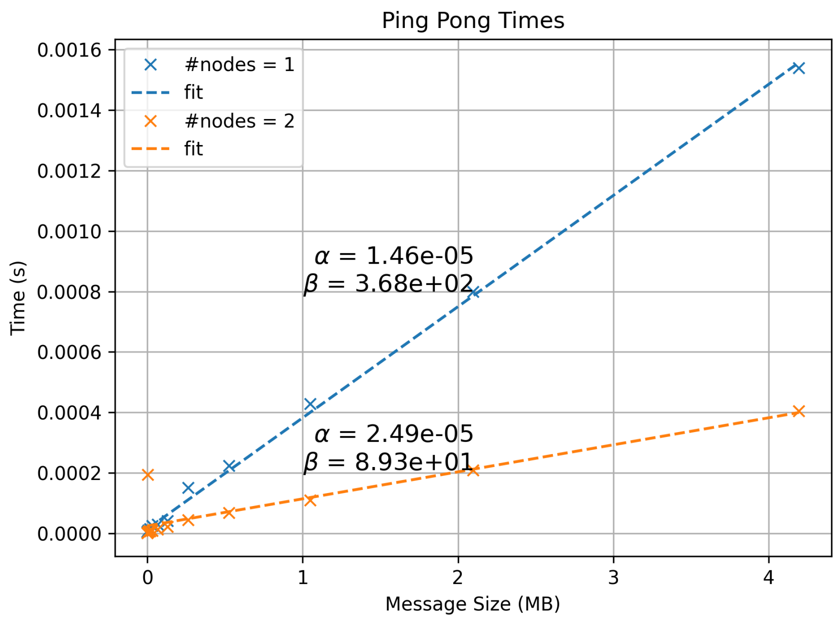

Figure 1: Ping pong results, with the time it took to send a message of size 2^i bytes. Additionally, the values of α and β are shown.

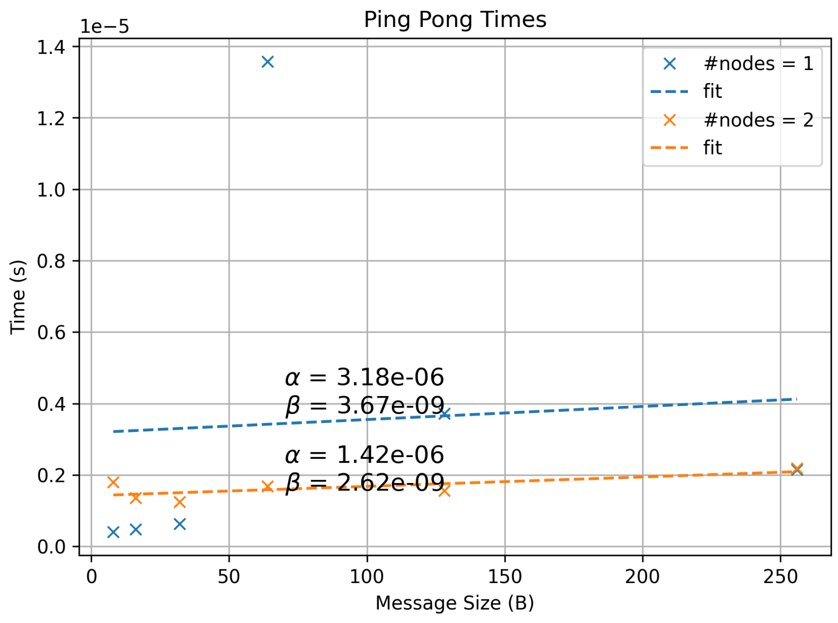

Figure 2: Similar to figure 1, but for small message sizes.

0.2 Matrix multiplication

The matrix matrix multiplication $AB = C$ is implemented in several phases:

0. Matrices A and B are initialized on every process.
1. Portion of matrix A to be multiplied with B by each process is determined.
2. Each process performs the multiplication locally performed.
3. Finally, the results are gathered.

Phase 1 is implemented by dividing the rows of matrix A into (approximately) equal portions for each process. That is to say, each process received $\frac{n}{p}$, where n is the number of rows in A and p is the number of processes. This leads to a balanced workload as long as the number of rows is divisible by the number of processes. If that is not the case, the last process will receive the remaining rows.Next to the phases above, the sequential calculation is performed only on the root process.

matrix size	processes	nodes	execution time (s)	**speedup**
128×128	1	1	8.72×10^{-3}	1.00
	2	1	4.29×10^{-3}	2.07
	4	1	2.03×10^{-3}	4.14
	8	1	1.07×10^{-3}	7.97
	16	1	5.85×10^{-4}	1.49×10^{1}
	32	1	3.23×10^{-4}	2.69×10^{1}
	48	1	2.12×10^{-4}	4.31×10^{1}
	64	2	2.24×10^{-4}	4.00×10^{1}
256×256	1	1	7.45×10^{-2}	1.00
	2	1	3.65×10^{-2}	2.13
	4	1	1.71×10^{-2}	4.17
	8	1	8.63×10^{-3}	8.33
	16	1	4.47×10^{-3}	1.67×10^{1}
	32	1	2.24×10^{-3}	3.26×10^{1}
	48	1	1.56×10^{-3}	4.91×10^{1}
	64	2	1.24×10^{-3}	6.31×10^{1}
512×512	1	1	1.06	1.00
	2	1	5.79×10^{-1}	1.97
	4	1	2.63×10^{-1}	3.86
	8	1	1.38×10^{-1}	7.86
	16	1	6.71×10^{-2}	1.58×10^{1}
	32	1	3.21×10^{-2}	3.11×10^{1}
	48	1	2.07×10^{-2}	4.90×10^{1}
	64	2	1.64×10^{-2}	6.21×10^{1}
1024×1024	1	1	9.49	1.00
	2	1	4.91	1.96
	4	1	2.18	3.89
	8	1	1.30	7.52
	16	1	6.40×10^{-1}	1.48×10^{1}
	32	1	2.61×10^{-1}	3.12×10^{1}
	48	1	1.98×10^{-1}	4.56×10^{1}
	64	2	1.49×10^{-1}	6.06×10^{1}

Table 1: Execution time and speedup for matrix multiplication

The time measurements for the matrix multiplication are shown in table 1. The general takeaway here is twofold. The time it takes to perform the matrix multiplication:

1. increases with the size of the matrix;
2. decreases almost linearly with the number of processes if the matrix is large enough.

The last conclusion is based on the fact that the speedup for matrix size 125 using 64 processors is only 40 times faster than the sequential version. While the speedup for a matrix of size 256 is 63 times faster. This reason for this occurence is that the 64 processors are located on two different nodes. This leads to a higher communication overhead, which is more pronounced for smaller matrices.

1 Parallel Poisson solver

1.1 Parallellisation of the Poisson solver

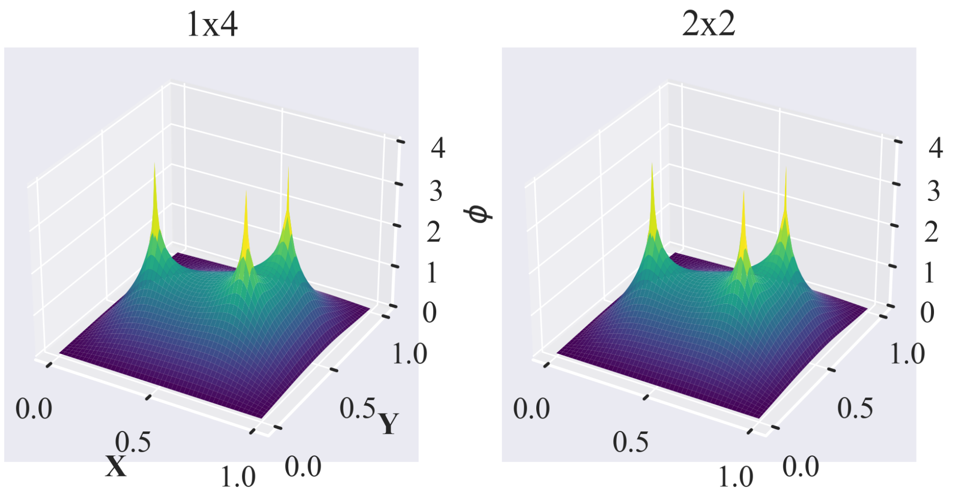

Figure 3: Surface plots of parallel poisson solution various (process) grid shapes.

1.1.1 Steps 1-4

Since these mainly deal with setting up MPI and file I/O I leave out discussion thereof, aside from the fact that I checked that all processes produce identical output, before continuing with following steps.

1.1.2 Steps 5-10

Step 5: altered `Setup_Grid()` to do read grid specifications and sources from the input file on root process and broadcast to other processes.

Step 6: implemented `Setup_Proc_Grid()` defining the process topology.

Step 7: adjusted `Setup_Grid()` to incorporate grid subdomain offsets and sizes.

Step 8: implemented `Setup_MPI_Datatypes()` and `Exchange_Borders()` for border exchange.

Step 9: implemented local error communication through `MPI_Allreduce()` for global error and convergence check.

Step 10: altered the `Write_Grid()` function to write output into a single (binary) file.

1.2 Improvements & performance analysis

Variables:

- n : the number of iterations
- p: number of processes
- g: gridsize
- $h = 1/(g+1)$: grid spacing
- t : time needed in seconds

- pt: processor topology in form pxy, where p: number of processors used x : number of processors in x - direction y : number of processors in y-direction

1.2.1 Algorithmic improvements

To implement the SOR I altered the `Do_Step()`, where now the global variable ω (relaxation parameter) is introduced.

1.2.2 optimal value for ω

Figures 4 and 5 show performance of the parallel poisson solver for various ω, grid sizes and process topologies. The optimal value for ω is found to be $\omega \approx 1.99$ for all cases. This is in keeping with the theoretical value of for the 2D Poisson equation **Yang2009**

$$\omega = \frac{2}{1+\sin(\pi h)},$$

which approaches 2 for small h.

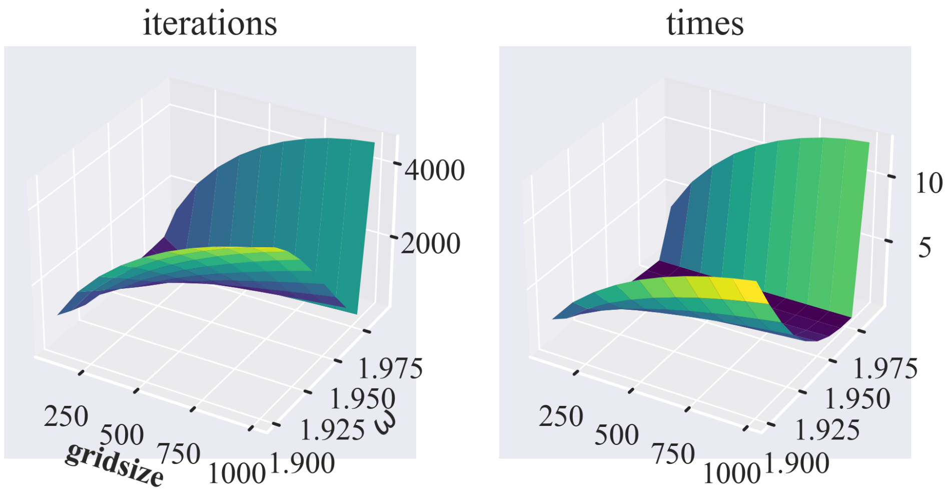

Figure 4: Surface plot of number of iterations (left) and runtime (right) depending on omega and grid size for a 4 × 1 process grid.

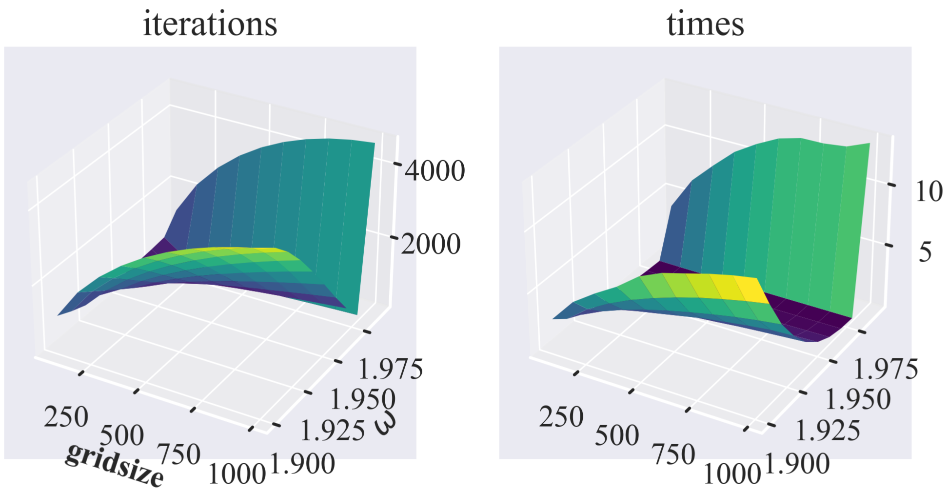

Figure 5: Similar to figure 4 but for a 2 × 2 process grid.

1.2.3 time versus iterations

Figure 6 shows the elapsed versus iteration number for different grid sizes and process topologies.

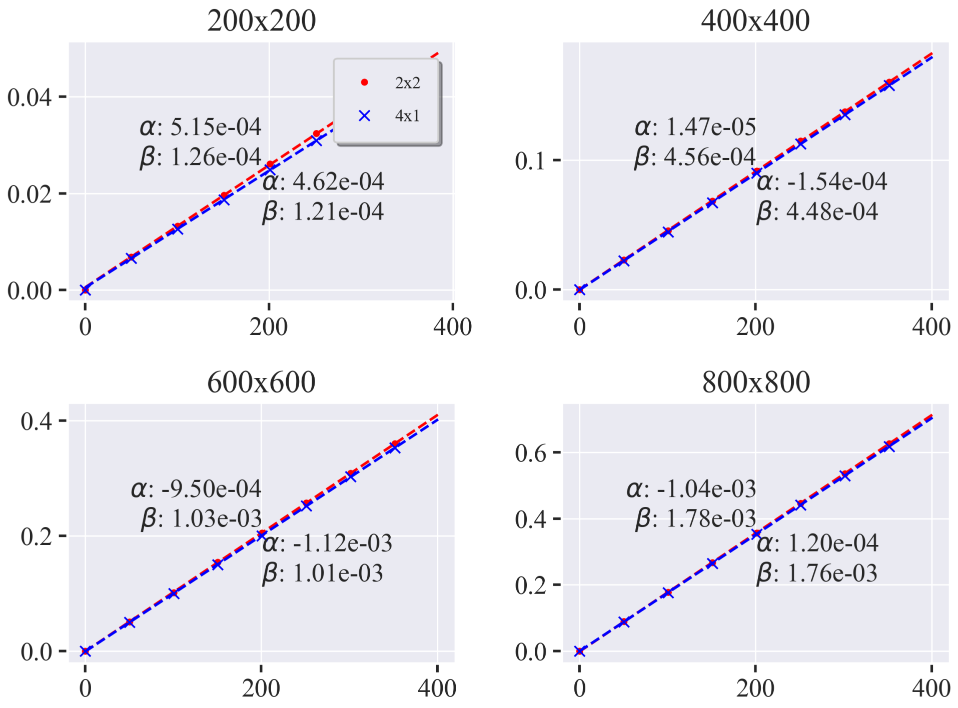

Figure 6: Time versus iterations for different grid sizes and process topologies. Estimated values for α and β are shown as well.

Note that some values of α turn out to be negative, which is not possible. This is a result of the linear regression model not being constrained to give zero or positive values for α. More interesting are the values for β, which increase for larger grid sizes. Additionally, the values for β are smaller for the 4×1 process grid, which is expected since the communication overhead is smaller. This is because the number of border exchanges is smaller for the 4×1 process grid.

1.2.4 domain partitioning

From the discussion above it is clear that in choosing between a 2×2 and 4×1 process grid, the 4×1 process grid is the better choice. This means that for 16 processors the 16×1 or 1×16 process grids are the better choice as well, as these minimize the number of border exchanges. In these cases the value of α is expectd to be four times as big, since the amount of messages quadruples. As for the value of β, this is expected to be fou times as big, since the amount of communicated data quadruples as well.

1.2.5 iterations versus problem size

See, again, figures 4 and 5. Figure 7 shows the number of iterations versus grid size for the optimal $\omega = 1.99$. We see that for larger grid sizes the number of iterations increases linearly.

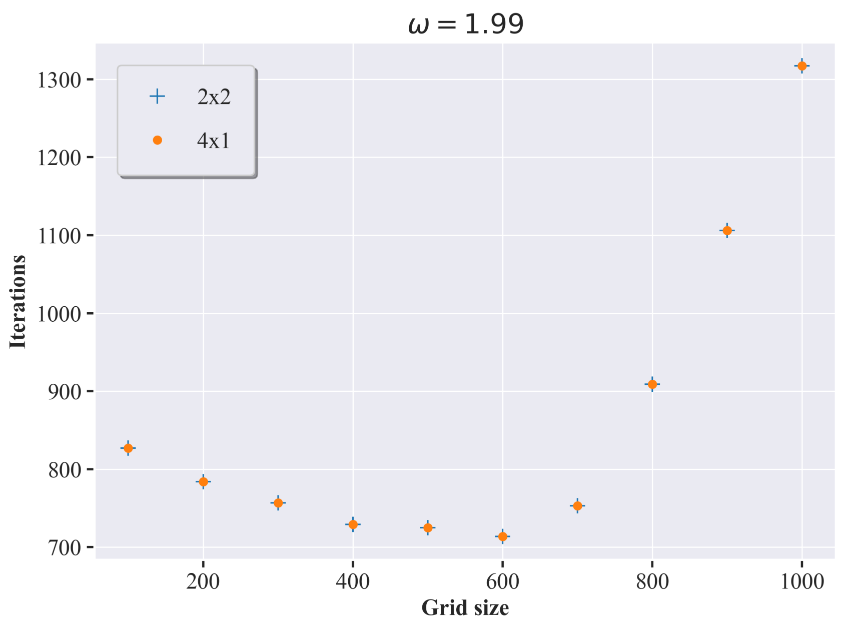

Figure 7: Iterations versus grid size for $\omega = 1.99$.

1.2.6 error of 800×800 grid

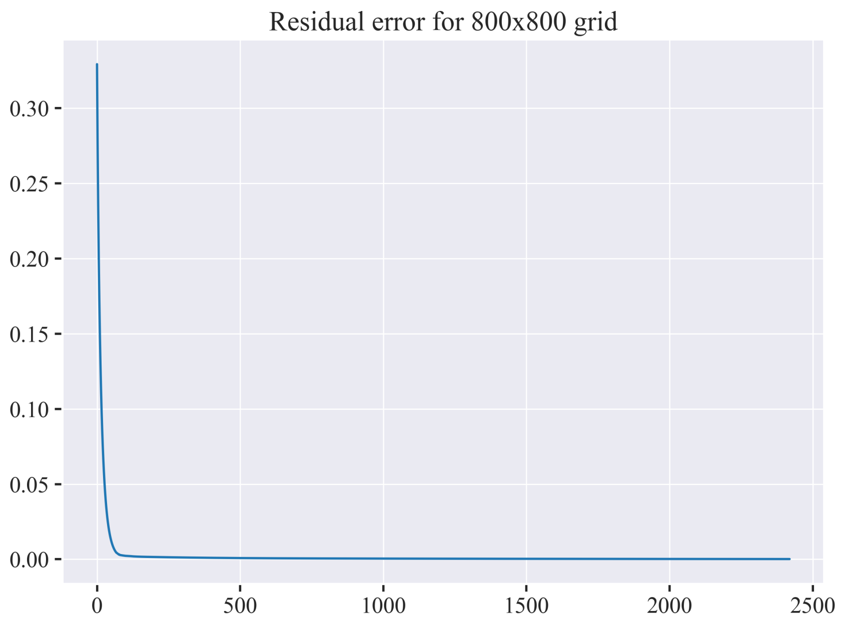

Figure 8: Error evolution versus iteration.

1.2.7 Global error communication reduction(optional)

To do ▶*implement sweeps over allreduce and compare to standard implementation.*◀

1.2.8 Border communication reduction

To reduce communication I altered the `Exchange_Borders()` function to only perform communication for iterations that are multiples of the number of sweeps.

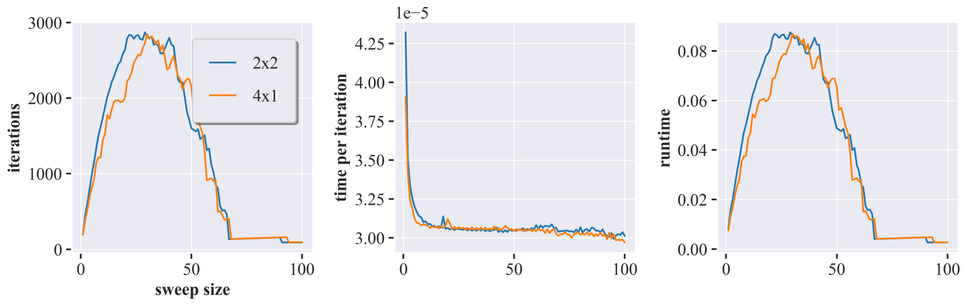

Figure 9: Time versus number of sweeps (left) and iterations versus number of sweeps (middle) and total runtime (right).

Figure 9 shows that the number of sweeps has a significant impact on the number of iterations. For a small sweep size the number of iterations and runtime increase, but the time per iteration sharply decreases. This is because by increasing the sweep size from 1 to 2, the number of border exchanges is halved. Then from 2 to 3 the number of border exchanges is halved again and so on.

For slightly larger sweep sizes the number of iterations increases, while the time per iteration stays approximately constant. This results in an overall increase in runtime.

Finally, for larger sweep sizes the number of iterations and runtime decrease again. This is because the communication between the processes is reduced to such an extent that they essentielly become independent. The solver still converges, because the stopping criterion is simply the residual between subsequent runs. This gives the illusion of better performance, but in reality the solver is not converging to a global solution. Instead, the solution is converging to a local solution on each process.

All in all, sweeps do not seem to be a good way to reduce communication while maintaining correctness.

1.2.9 time spent in border exchange (optional)

To do ►*perform latency analysis for uniform (proces) grids sizes*◄

1.2.10 latency in test problem

Timing of the communication overhead is implemented in the `Exchange_Borders()` function

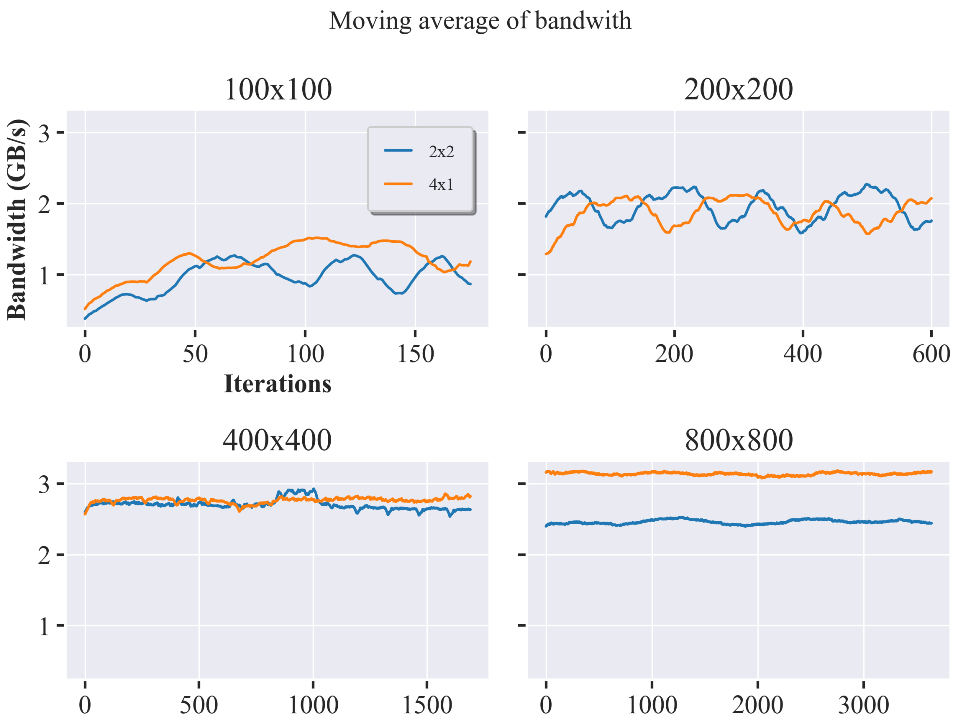

Figure 10: Latency analysis for different (proces) grid sizes. Shown is the moving average of with a window size of 10% of the data.

1.2.11 On optimization of border exchange

- The adress of the first point to exchange depends on:
  - the phase, because it is either (red- or black-sweep).
  - the color of the gridpoint (red or black)
- The number of points to exchange depends on:
  - the phase, as we one needs to group these points in a single message,
  - the size of the subdomain (even or odd).
- The number of points in between to be exchanged points depends on:
  - the phase, as we one needs to group these points in a single message,
  - the size of the subdomain (even or odd).

It ultimately depends on the amount of time spent on extra operations compared to the communcition time saved whether or not this optimization is worth it. For relatively large $\frac{g}{p}$ the time spent on communication is not significant, so the optimization is not worth it. However, for relatively small $\frac{g}{p}$ such an optimization might be worth it.

2 Parallel FEM Poisson solver

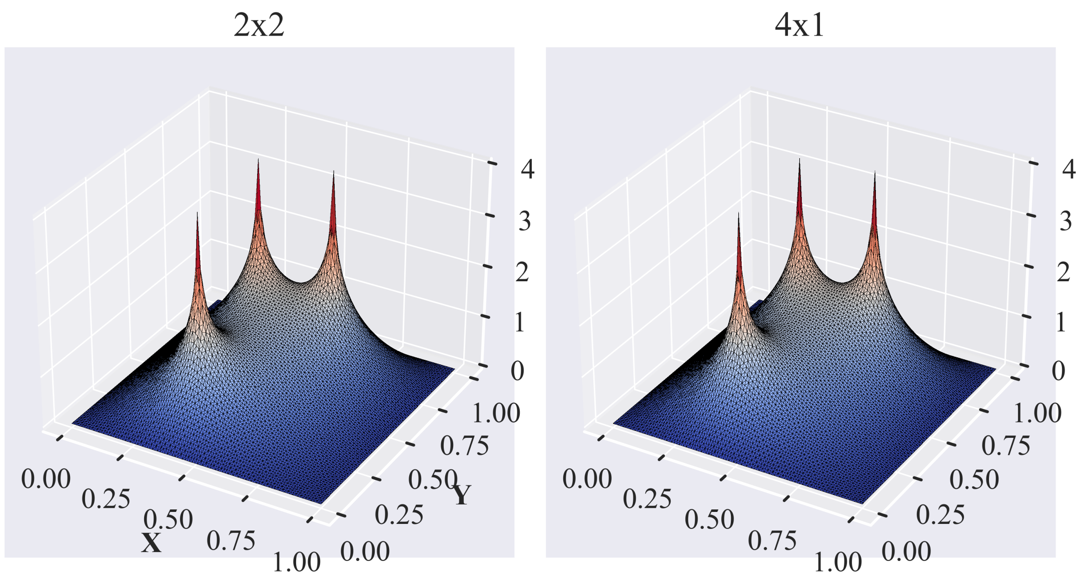

Figure 11: Solution of the parallel FEM Poisson solver.

2.1 Data communication between neighbouring processes

The data communication between neighbouring processes is implemented in the `Exchange_Borders()`

2.2 Benchmarking

Figure 12 shows the timing results of the parallel FEM Poisson solver. Note that I added one extra category for file I/O.

From the figure we see that the computation time is approximately the same for all processes for any grid size and a given process topology. This is to be expected, as processes keep performing computations untill the global error criterion is met.

Additionally, assymetry in exchange is evident from the figure. I explain why this is so in section 2.4.

Lastly, we observe that a good portion of the root process' time is spent in file I/O. This is due to the fact that the root process is responsible for reading from the input and writing to final the solution to file. While it performs these tasks the other processes remain idle.

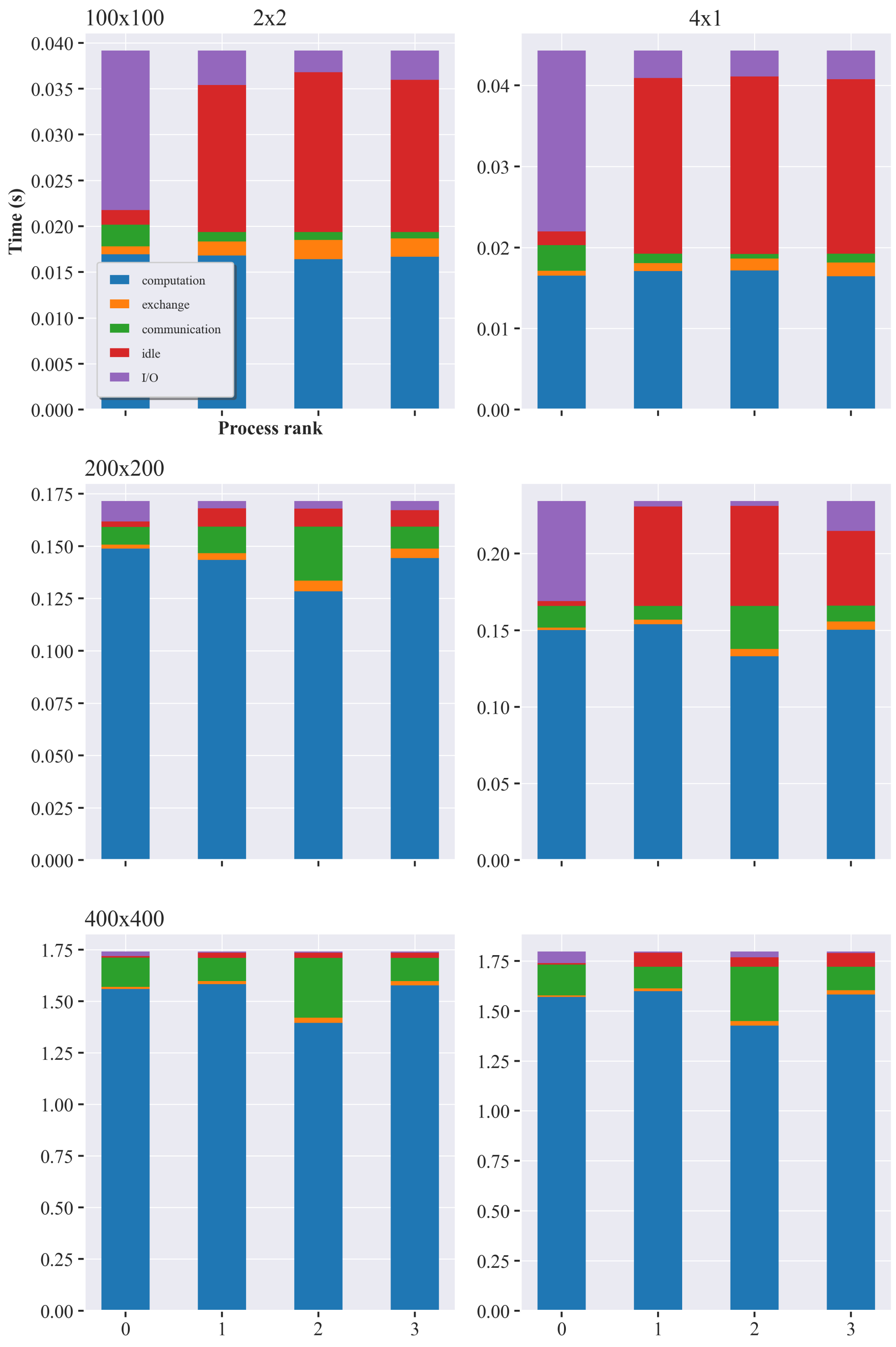

Figure 12: Benchmarking of the parallel FEM Poisson solver. Each bar corresponds to a process, rows correspond to grid sizes, columns correspond to the process topology and colours to thr different categeories of time; computation (blue), exchange/local communication (orange), global communication (green), idle (red) and file I/O (purple).

2.3 Data communication estimates

Let n and p be the number of grid points and processes in each direction respectively. Any two neighbouring processes share $\frac{n}{p}$ vertices. Inner processes have 4 neighbours, edge processes have 3 neighbours, and corner processes have 2 neighbours. There are $(p-2)^2$ inner processes, $4(p-1)$ edge processes, and 4 corner processes. Hence the total number of shared vertices is given by

$$\begin{aligned}
|V_{\text{shared}}| &= \frac{n}{p}\left(4(p-2)^2 + 12(p-1) + 8\right) \\
&= \frac{n}{p}\left(4p^2 - 16p + 16 + 12p - 12 + 8\right) \\
&= \frac{n}{p}\left(4p^2 - 4p + 12\right) \\
&= \frac{4n}{p}\left(p^2 - p + 3\right),
\end{aligned}$$

which holds for $p > 3$ and $n \geq p$. Since a double is 8 bytes and data is communicated in both directions, the total amount of communicated data per iteration is approximately given by

$$\begin{aligned}
B &= 16|V_{\text{shared}}|[B] \\
&= \frac{64n}{p}\left(p^2 - p + 3\right)[B].
\end{aligned}$$

2.4 Communication assymetry

From the function `Write_Datafiles()` I deduce the communication pattern shown in Figure 13, which shows a sketch of the grid for $n = 6$ and $p = 3$. From this figure we see that vertex exchange for the inner process p_{11} takes place in 6 distinct locations and directions. This means that the inner process has 6 neighbours. In the figure we also see the communication of the corner process p_{22} which only has 2 neighbours.

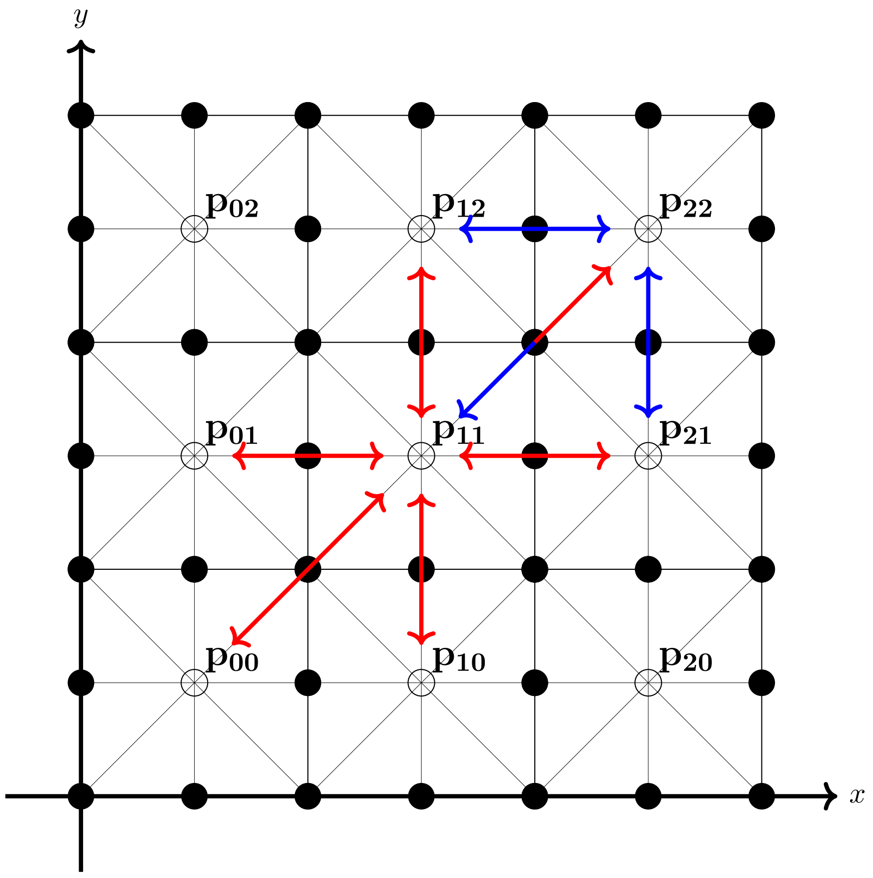

Figure 13: Sketch of gird for $n = 6$ and $p = 3$. Filled black circles represent shared vertices and black circles represent the vertices of the internal vertices. In this example each process only contains one inner vertex. The red lines indicate vertex exchange between the inner process $p_1 1$ and its neighbours.

Assymetry in communication stems from the fact that vertices located on the boundary of 4 processes are ambiguous w.r.t. which of those process they belong to. Hence we choose to exclusively link bottom-left and/or top-right neighbours.

2.5 Communication and computation overlap

Figures 14 and 15 show the communication and computation overlap in the parallel FEM Poisson solver defined as

$$\text{overlap} = \frac{\text{computation time}}{\text{communication time}}. \tag{2}$$

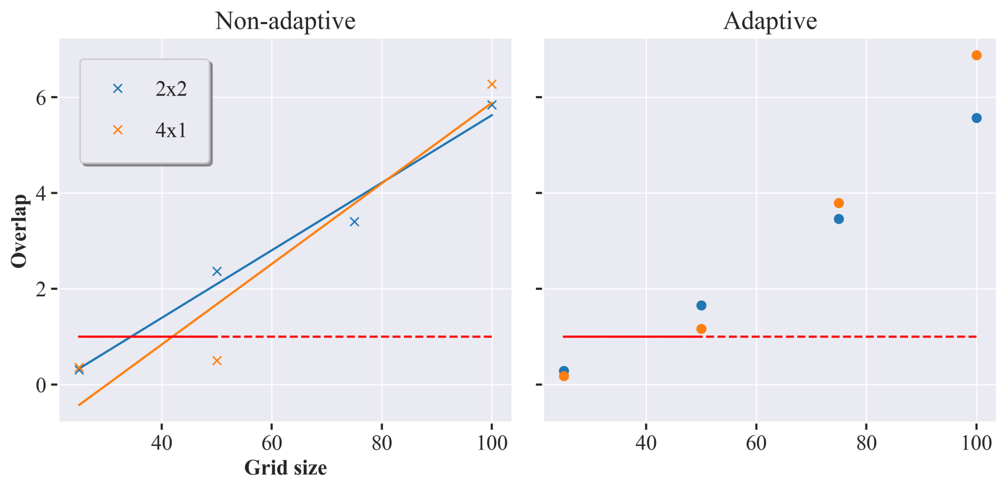

Figure 14: Communication (global + local) and computation overlap in the parallel FEM Poisson solver for 2×2 and 4×1 processes plotted against gridsize.

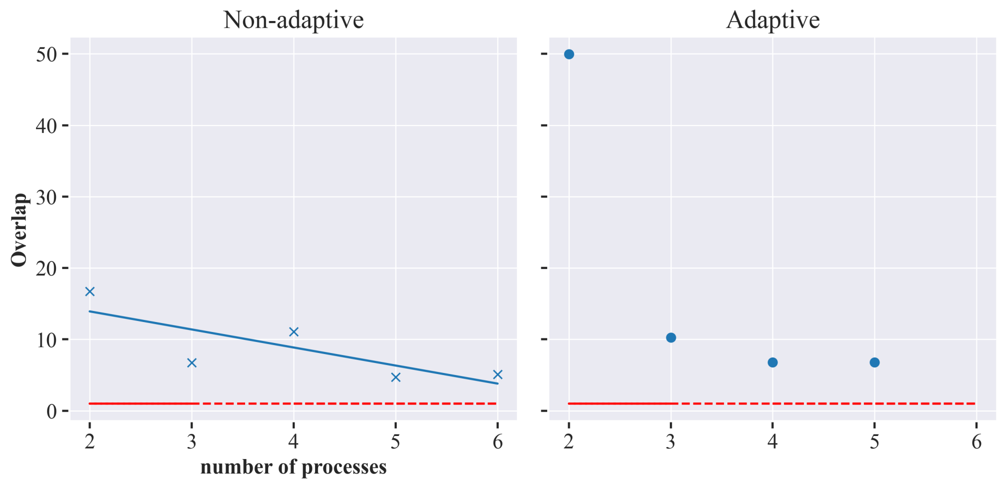

Figure 15: Communication (global + local) and computation overlap in the parallel FEM Poisson solver plotted against number of processes.

From the above figures we see that for fixed $p = 4$ and a regular grid, the communication and computation overlap equals 1 for grid sizes $g = 35$ to 40 depending on process topology. This means that the computation time is approximately equal to the communication time. For the adaptive grid, the overlap is approximately 1 for $g = 50$.

Similarly, for a fixed $g = 100$ and a regular grid, the overlap (approximately) equals 1 for $p = 6$ and is predicted to be below 1 for $p > 7$. The experiment is inconclusive for the adaptive grid, but the overlap is close to onr for $p = 4$ and 5.

2.6 Adaptive grid refinement

Figure 16 shows the Effect of adaptive grid refinement on error evolution

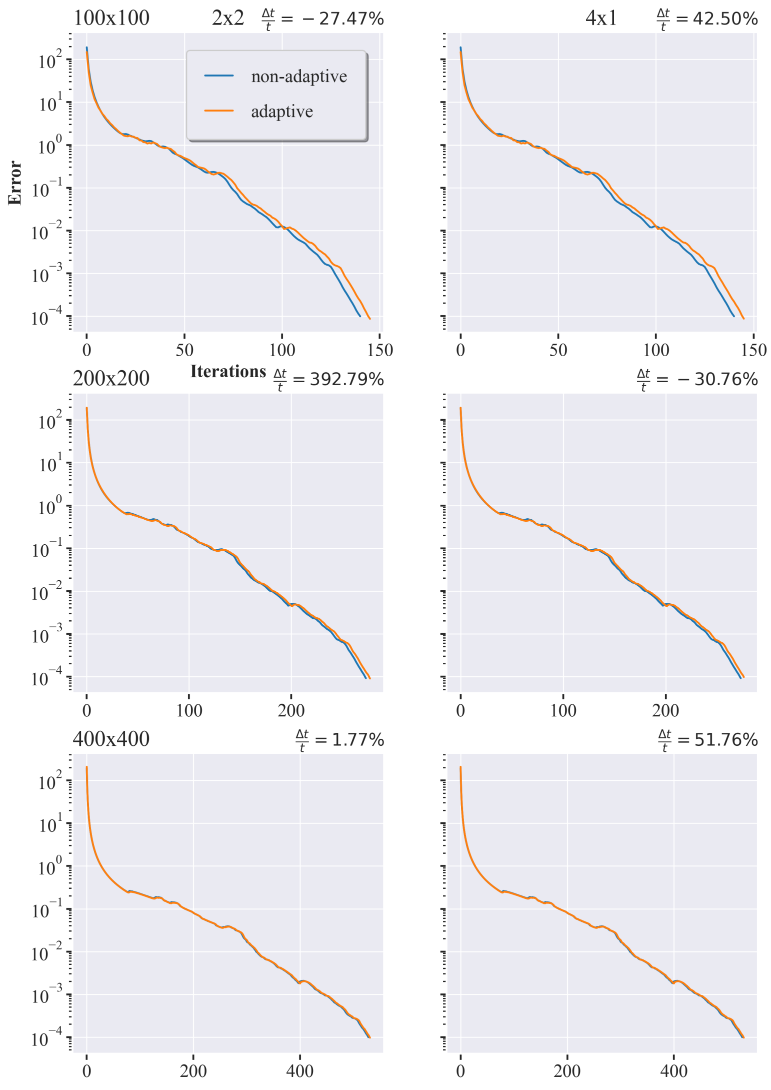

Figure 16: Error evolution for $g = 100, 200, 400$ for both process topologies and on (non-)adaptive grids. Rows correspond to grid sizes, columns correspond to the process topology, blue lines to non-adaptive grids and orange lines to adaptive grids. On the top right of each plot the relative percentual speedup is shown, where negative values indicate a faster computation time for the adaptive grid.

3 Power method on the GPU

Next to the provided implementations of the GPU util functions using shared memoty I implemented global memory versions of these same functions

- `Av_Product_Global()`
- `FindNormW_Global()`
- `NormalizeW_Global()`
- `ComputeLamda_Global()`

I then wrapped these functions in a more general GPU power method function RunGPUPowerMethod, which allows to dynamically choose between the global and shared memory versions of the util functions. It also performs all necessary time measurements.

3.1 Performance analysis

All time results below are averaged over 5 runs. In case of two entries, the left one is the global memory version and the right one is the shared memory version.

matrix size	cpu time	32 threads		64 threads		100 threads	
50	8.30×10^{-5}	4.14×10^{-4}	5.19×10^{-4}	4.84×10^{-4}	5.94×10^{-4}	4.76×10^{-4}	N.A.
500	1.49×10^{-2}	7.19×10^{-4}	1.48×10^{-3}	1.75×10^{-3}	1.03×10^{-3}	1.09×10^{-3}	N.A.
2000	1.16×10^{-1}	4.34×10^{-3}	2.18×10^{-3}	6.55×10^{-3}	2.66×10^{-3}	3.85×10^{-3}	N.A.
4000	2.89×10^{-1}	3.04×10^{-2}	5.79×10^{-3}	3.11×10^{-2}	1.12×10^{-2}	1.59×10^{-2}	N.A.
5000	1.40	2.98×10^{-2}	2.44×10^{-2}	3.48×10^{-2}	2.64×10^{-2}	2.69×10^{-2}	N.A.

Table 2: Execution time for different matrix sizes and number of threads.

matrix size	32 threads		64 threads		100 threads	
50	0.22	0.18	0.17	0.14	0.19	N.A.
500	20.86	10.15	8.49	14.42	8.57	N.A.
2000	44.32	87.99	17.78	43.81	30.22	N.A.
4000	9.35	49.09	9.28	25.75	18.02	N.A.
5000	42.54	52.05	40.17	52.79	48.55	N.A.

Table 3: speedups without memory transfer.

matrix size	32 threads		64 threads		100 threads	
50	0.17	0.14	0.13	0.11	0.15	N.A.
500	17.04	8.87	7.18	12.43	7.48	N.A.
2000	42.15	81.91	16.99	41.79	28.96	N.A.
4000	9.25	47.69	9.16	25.35	17.81	N.A.
5000	42.00	51.75	39.62	52.51	48.08	N.A.

Table 4: speedups with memory transfer.

3.2 Explaination of the results

Tables 2, 3 show that as long as the number of threads per block does is not too large, the shared memory version is faster than the global memory version, even with memory transfers included (Table 4). This is because as stated in the assignment description, shared memory is much faster than global memory.

However, as the number of threads per block increases, more and more memory is stored on-chip (L1 cache) of any SM. At 100 threads per block the shared memory requirement becomes $100^2 \times 8 = 80$ kB. Shared memory is limited to 64 kB per SM on Nvidea's T4 architecture. Hence, it is not possible to run the shared memory version with 100 threads per block. This is why these time measurements are not included in the tables above.

Furthermore, we recognise the principle that was discussed in the lectures as well; the task needs to be computationally intensive enough for the GPU to be faster than the CPU. That is to say, for small matrices there is no speedup, slow down even, of the GPU variant of the power method.

Bibliography

Author, A. (2019). Title. *Journal*, *1*(1), 1–2. https://doi.org/10.1109/ACCESS.2019.2897270