Parallel Programming SS21 Final Project

02 Gaussian Elimination

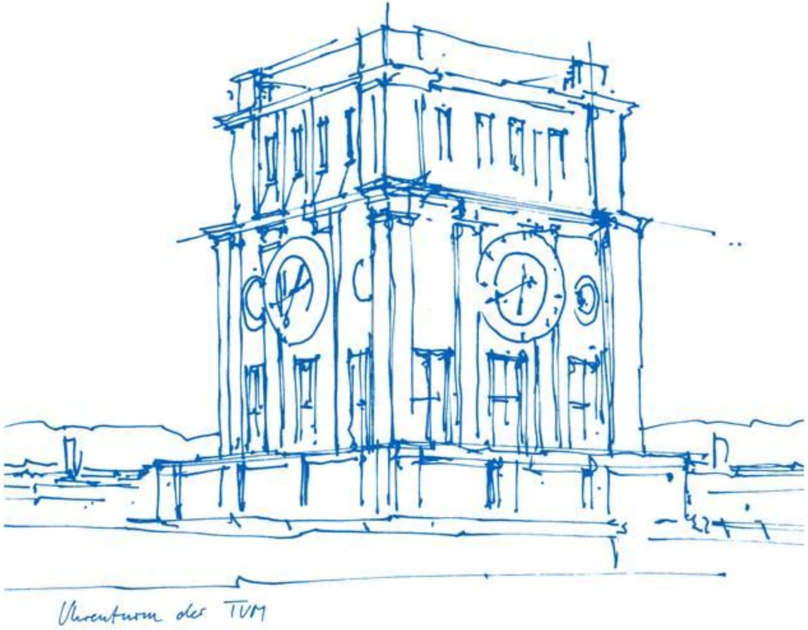
Group 218

13.07.2021

Sandro Bauer

Marcel Braasch

Syed Salman Ahmed



Outline

1. Sequential code analysis
2. OpenMP
3. MPI
4. Hybrid
5. Bonus
6. Conclusion

Sequential code analysis - Forward Elimination

When calculating the forward elimination, for every row, we take the diagonal element d, calculate an elimination factor f=d/x such that for all lower elements in the column r*x-r=0. This diagonal element stays constant with respect to x. Thus we do not need to calculate it over and over again.

Graphically, that is

$d_1 = 1$
$x_{11} = 6$
$f_1 = \frac{1}{6}$
$x_{11} = x_{11} \cdot f_1 - x_{11} = 0$
Repeat for all x_{1j}.

$$\begin{bmatrix} \boxed{1} & 2 & 3 & 1 \\ 6 & -4 & 5 & 0 \\ 2 & 3 & 5 & 0 \\ 5 & -10 & 1 & 0 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} 1 \\ 6 \\ 9 \\ 0 \end{bmatrix}$$

⇒ We can see that d_1 stays constant for all eliminations. The given algorithm constantly calculates d_i, thus we moved this calculation outside. All further analyses apply to the algorithm with this change.

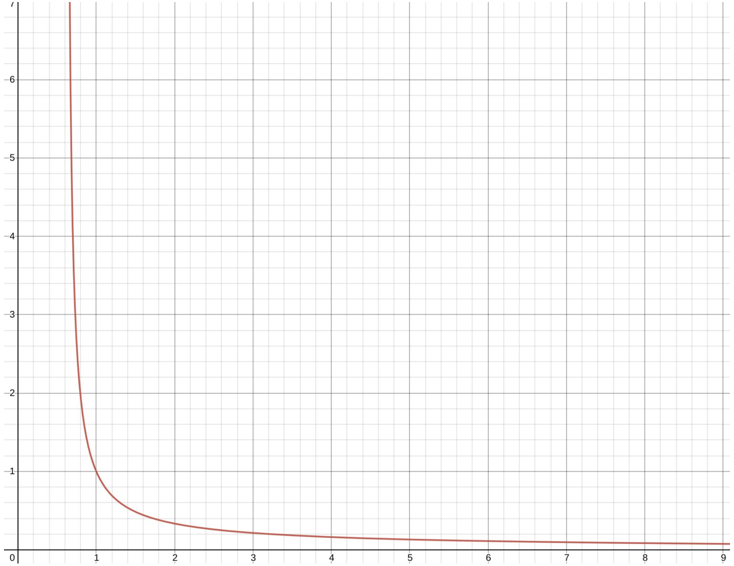
Sequential code analysis - Theoretical Analysis

For the Forward Elimination we have
$n(n+1)/2$ division
$(2n^3+3n^2-5n)/6$ multiplications
$(2n^3+3n^2-5n)/6$ subtractions
$= n(n+1)/2 + (2n^3+3n^2-5n)/3 =$ **FE(n)** operations

For the Backward Elimination we have
$n(n-1)/4$ multiplications
$n(n-1)/4$ additions
$= n(n-1)/2 =$ **BE(n)** operations

Let's look at the ratio for **BE(n)/FE(n)**!

⇒ Quickly converges to 0
⇒ FO outweighs BO by **a lot**
⇒ E.g., for n = 1024, FO causes 99,85% of the computations
⇒ Gives rise to <u>only optimizing **FO**</u> due to high overhead



The graph f(n)=BE(n)/FE(n) showing the ratio of BE and FE indicating the importance of BE

Sequential code analysis - Profiling and Backward

- Perf output roughly coincides with the theoretical calculation
- FE percentage slightly lower since other operations cause overhead
- Backward elimination not even visible

The backward algorithm is

1. doing a negligible amount of work
2. has strong sequential data dependency

⇒ Neglect the backward algorithm throughout all our future optimizations and the rest of the presentation

```
Samples: 520  of event 'cache-references', Event count (approx.): 122283612
Overhead  Command   Shared Object        Symbol
  97,42%  serialge  serialge             [.] Serial::ForwardEliminationOptimized
   0,36%  serialge  serialge             [.] Serial::SerialSolve
   0,35%  serialge  libstdc++.so.6.0.28  [.] std::num_put<char, std::ostreambuf_iterator
   0,19%  serialge  libstdc++.so.6.0.28  [.] std::num_get<char, std::istreambuf_iterator
   0,14%  serialge  libstdc++.so.6.0.28  [.] std::istream::sentry::sentry
   0,13%  serialge  libstdc++.so.6.0.28  [.] std::num_get<char, std::istreambuf_iterator
   0,13%  serialge  libc-2.31.so         [.] __GI_____strtod_l_internal
   0,08%  serialge  libstdc++.so.6.0.28  [.] std::string::reserve
   0,07%  serialge  libc-2.31.so         [.] malloc
   0,06%  serialge  libstdc++.so.6.0.28  [.] std::string::_Rep::_M_clone
   0,05%  serialge  libc-2.31.so         [.] __strlen_avx2
   0,05%  serialge  libstdc++.so.6.0.28  [.] std::__convert_to_v<double>
   0,05%  serialge  [kernel.kallsyms]    [k] clear_page_erms
   0,04%  serialge  serialge             [.] ReadLine
```

Figure: Perf profiling output for serial implementation

Sequential code analysis - Amdahl‘s law

Q: What portion of the algorithm is parallelizable?

$$\begin{bmatrix} 1 & 2 & 3 & 1 \\ 6 & -4 & 5 & 0 \\ 2 & 3 & 5 & 0 \\ 5 & -10 & 1 & 0 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} 1 \\ 6 \\ 9 \\ 0 \end{bmatrix}$$

- Low data dependency
- Each of i+1th rows is only dependent on the element A_{ii}
- Therefore, propagating down the diagonal element can be nicely parallelized
- As this is all the algorithm is doing (besides a few other operation), **we expect roughly 97%** (according to Perf) to be parallelizable

Sequential code analysis - Amdahl‘s law

Q: What portion of the algorithm is parallelizable?

So let’s evaluate this, given our OMP speed up of L = 14.

Amdahl’s Law is given by

$$L_p(s) = 1 / (1 - p + p/s)$$

where L is the latency (i.e., the speed up), p is the portion to be parallelizable and s the amount of workers.

Plugging in all the values yields

$$14 = 1 / (1 - p + p/s)$$
$$\Leftrightarrow \quad p \approx 0.96$$

⇒ According to a speed up of factor 14, approximately 96% of the program is parallelizable

Similarly, we can ask ...

Sequential code analysis - Amdahl‘s law

Q: Given the portion of the algorithm to parallelizable (p=0.97), what’s the theoretical speed up maximum?

Plotting the $L_p(s)$ yields

⇒ Assuming 97% parallelizability using 32 CPUs theoretical speed up of factor 16.5

⇒ Clearly sublinear but still very nice curve.

⇒ Theoretical max speed up: $L_p(s)$ converges towards 33.3 for $s \rightarrow \infty$.

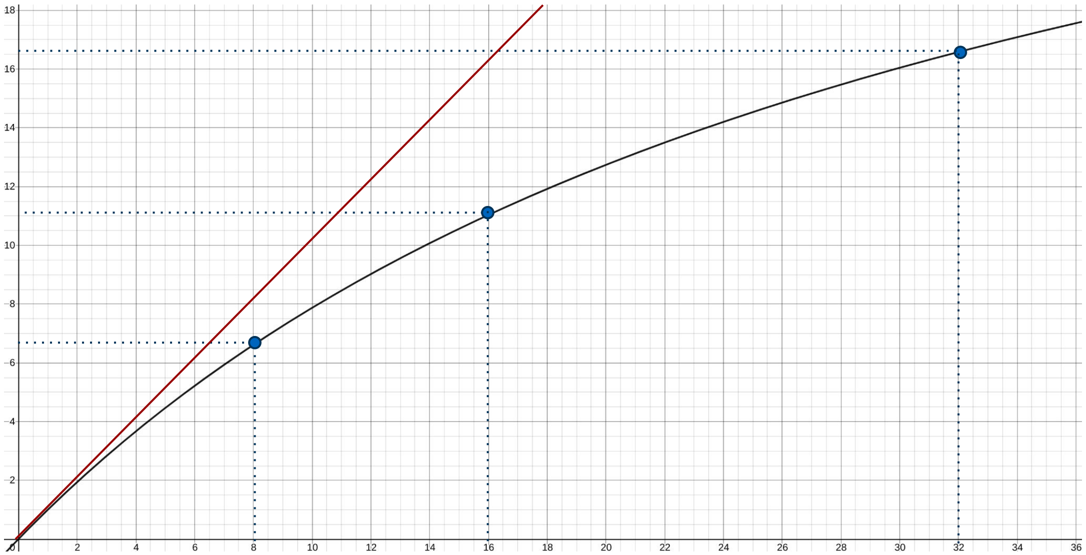


Figure: Amdahl’s law with p=0.97.

OpenMP - Parallelized implementation and approach

Initial approach

- Used **OMP task** to parallelize program
- Resulted in **no speed up at all**
- Taking a closer look we recognized **not the correct approach**
- OMP Tasks are (loosely speaking) a way to dynamically distribute workload
- Use case would be recursion with no interdependence between tree branches
- GE has low data dependence in forward algorithm, however,
- **Dynamically distributing matrix rows very inefficient**

Current approach

- For every pivot row from step i=1, we have to modify the following n-i rows
- All following n-i rows are only dependent on i-th row’s pivot element
- Statically spread the workload across all workers minimizing overhead

OpenMP - Parallelized implementation and approach

We found that

1. Out of the explicit scheduling schemes (static, dynamic, guided) static is the fastest
2. Runtime system using schedule(auto) picks static too →static seems like a good strategy

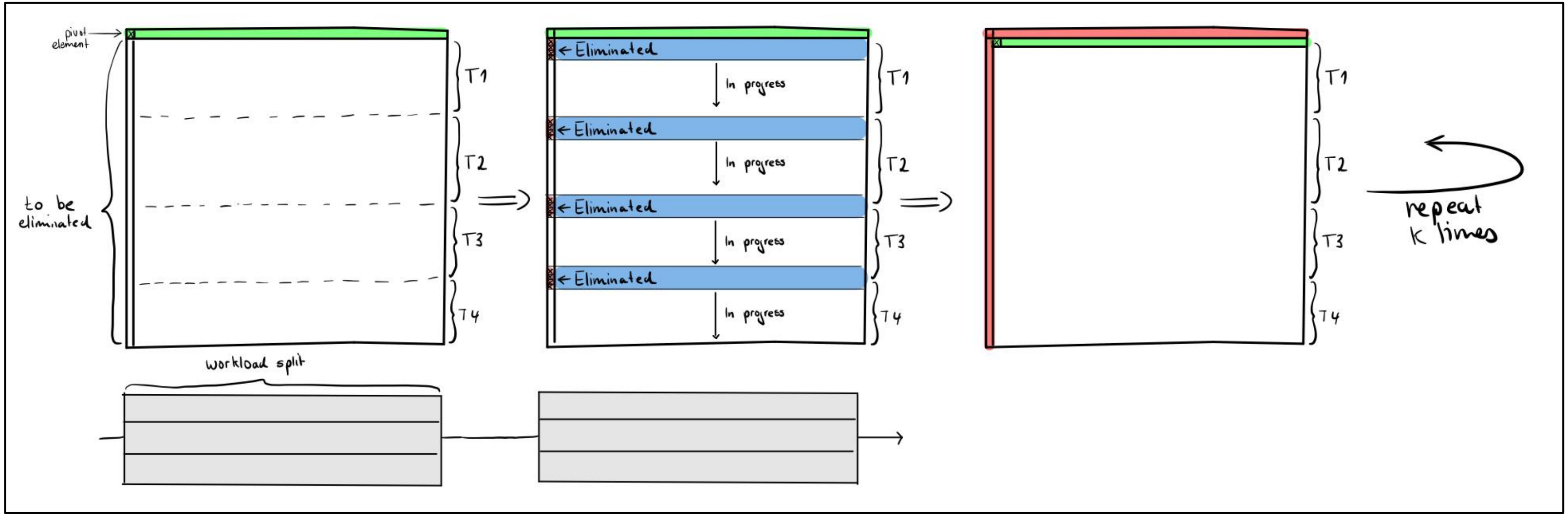


Figure: Schematic workload spread for k × k in forward algorithm with n=4 and static scheduling.

OpenMP - Overhead and when to parallelize

Q: At what problem size does parallelization make sense?

- Ran every problem size 10 times and averaged over computation vs. overhead
- For small problems overhead takes almost 100% of computing time ⇒ parallelization makes no sense
- For the sample problems given, starting from size 512 overhead starts to decrease (but still, 91% overhead)
- Size 2024 fist point very computation time > overhead

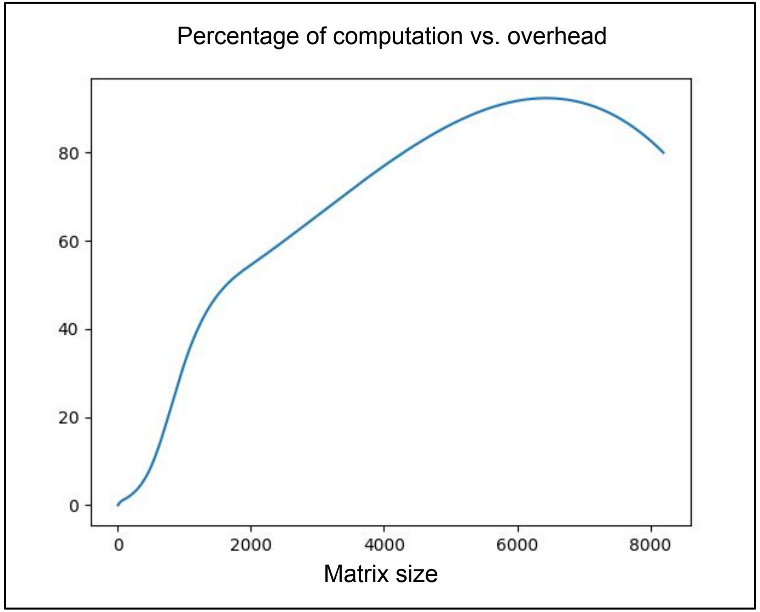


Figure: Spline (with k=3) displaying ratio computation vs. overhead.

OpenMP - Perf and possible bottlenecks

- **No change** between actual work across the programs
- Makes sense as the program **semantics do not change** at all - only the way work is being executed

- **65% of program are arithmetics**

⇒ Straightforward element-wise operations
⇒ Theoretically suitable SIMD use case
⇒ Compiler is already taking care of it

- **30% of program are reading**, writing and moving of elements

⇒ Use matrix instead of element-wise operations
⇒ Again, compiler is already taking care of it

Serial

```
Samples: 4K of event 'cycles', 4000 Hz, Event count (approx.): 4661322483
Serial::ForwardEliminationOptimized  /home/marcelbraasch/CLionProjects/openmp/serialge [Percent: local period]
  2,55 │       vinsertf128  $0x1,0x10(%rbx,%rax,1),%ymm7,%ymm1
  7,37 │       vmovupd      (%rdx,%rax,1),%xmm6
 12,54 │       vinsertf128  $0x1,0x10(%rdx,%rax,1),%ymm6,%ymm0
 17,44 │       vmulpd       %ymm3,%ymm1,%ymm1
 16,41 │       vsubpd       %ymm1,%ymm0,%ymm0
  2,45 │       vmovups      %xmm0,(%rdx,%rax,1)
  6,23 │       vextractf128 $0x1,%ymm0,0x10(%rdx,%rax,1)
 31,91 │       add          $0x20,%rax
  2,21 │       cmp          %rsi,%rax
```

Tasking

```
Samples: 13K of event 'cycles', 4000 Hz, Event count (approx.): 11385415991
OMP::ForwardElimination  /home/marcelbraasch/CLionProjects/openmp/ompge [Percent: local period]
  8,20 │       vinsertf128  $0x1,0x10(%r14,%r13,1),%ymm5,%ymm0
 12,77 │       vmulpd       %ymm3,%ymm1,%ymm1
 16,99 │       vsubpd       %ymm1,%ymm0,%ymm0
  3,67 │       vmovups      %xmm0,(%r14,%r13,1)
  7,45 │       vextractf128 $0x1,%ymm0,0x10(%r14,%r13,1)
 33,84 │       add          $0x20,%r13
  0,87 │       cmp          %rsi,%r13
```

Parallel for

```
Samples: 27K of event 'cycles', 4000 Hz, Event count (approx.): 25442538013
OMP::ForwardElimination  /home/marcelbraasch/CLionProjects/openmp/ompge [Percent: local period]
  0,05 │       vmovupd      (%rax,%r13,1),%xmm6
  3,21 │       vinsertf128  $0x1,0x10(%rax,%r13,1),%ymm6,%ymm1
  8,45 │       vmovupd      (%r14,%r13,1),%xmm5
  8,24 │       vinsertf128  $0x1,0x10(%r14,%r13,1),%ymm5,%ymm0
 13,75 │       vmulpd       %ymm3,%ymm1,%ymm1
 16,93 │       vsubpd       %ymm1,%ymm0,%ymm0
  3,82 │       vmovups      %xmm0,(%r14,%r13,1)
  7,52 │       vextractf128 $0x1,%ymm0,0x10(%r14,%r13,1)
 32,14 │       add          $0x20,%r13
  0,84 │       cmp          %rsi,%r13
```

Figure: Perf output for the respective OMP vs. serial programmes

MPI - Parallelized Implementation and Approach

- Initial parallelization approach
  - Calculate distribution based on current status
  - Send rows from root to other ranks
  - Get calculated rows back

loop

- Motivation of implementation
  - Easy to implement / intuitive solution
  - Organization only done in rank 0

- Bottleneck due to high amount of communication

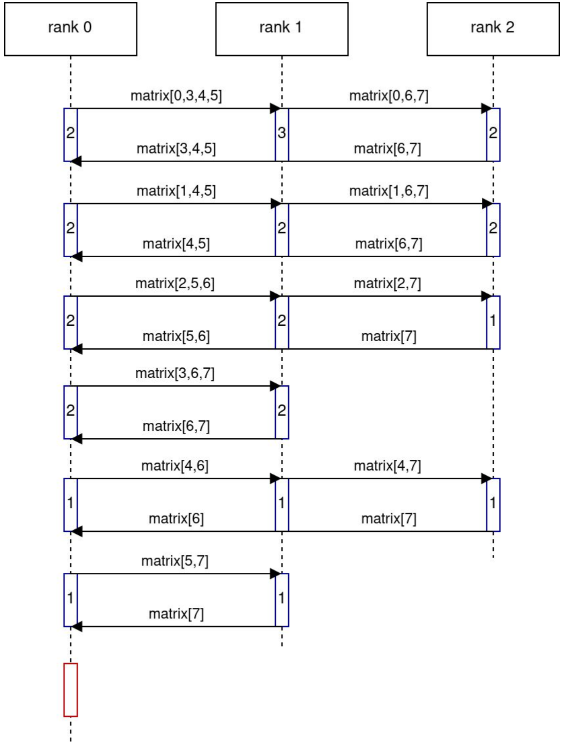


Figure: Schematic sketch of our first approach.

MPI - Parallelized Implementation and Approach

- Bottleneck is removed by:
  - Distribute in the beginning
  - Only share current master row

- Improvements
  - Less communication overhead
  - Worse distribution of calculation load
  - More complex implementation

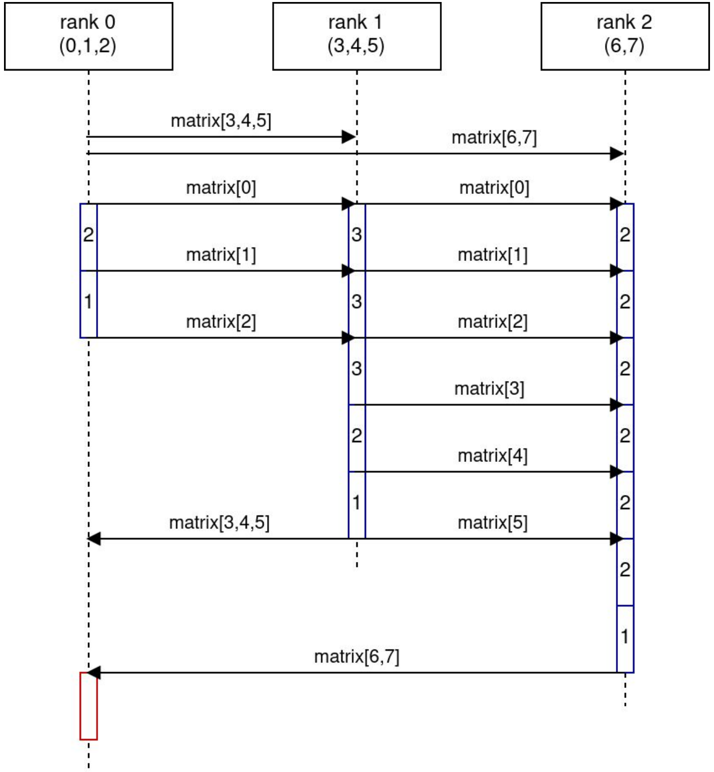
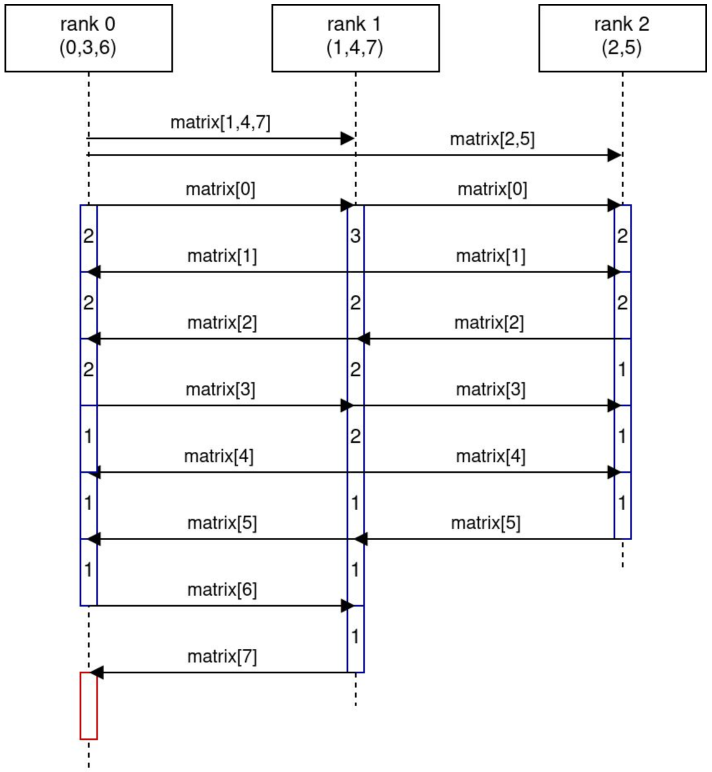


Figure: Schematic sketch of the improved approach.

MPI - Final Implementation improvements and new speed-up

- When starting with hybrid, we developed an even better approach
  - Distribute in the beginning in round robin sequence
  - Matrix is worked off evenly by all processes



MPI - Speed up comparison

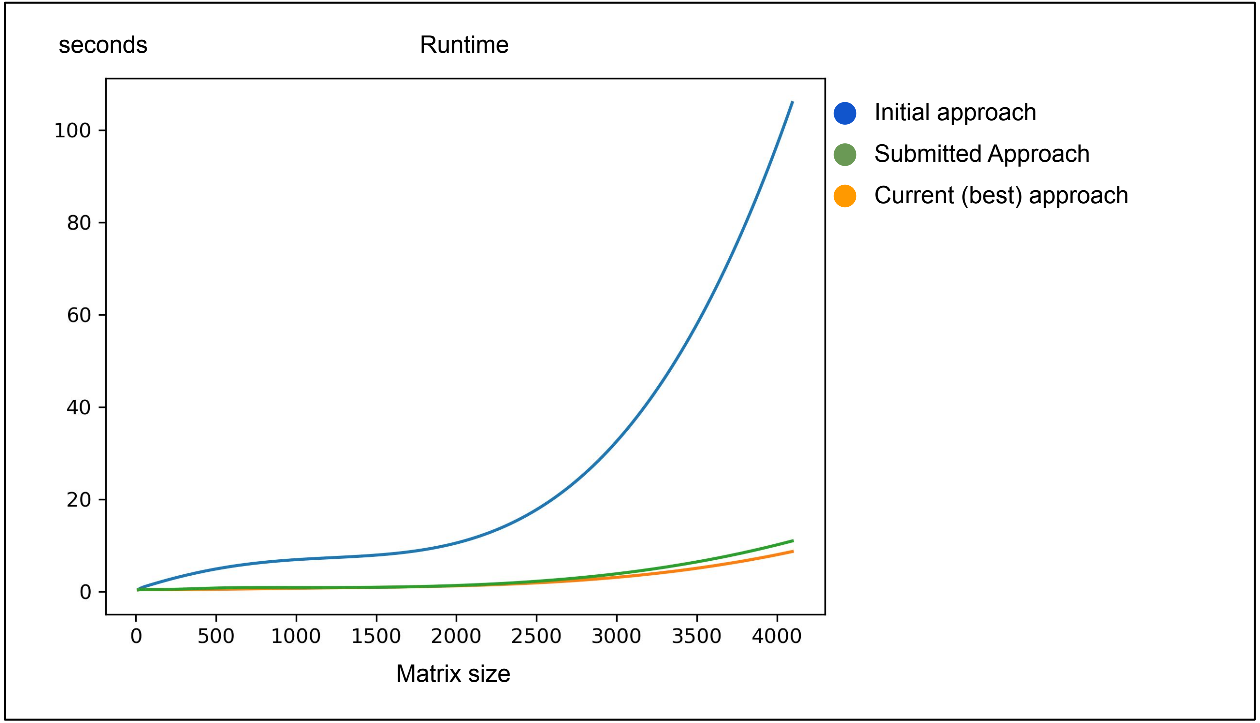


Figure: Compiler already vectorizing element-wise operations.

MPI - Communication vs. Calculation

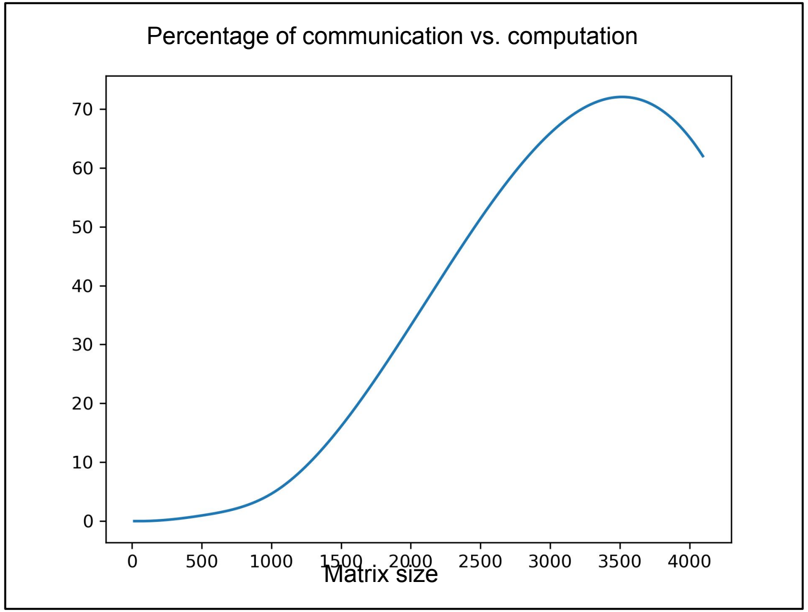
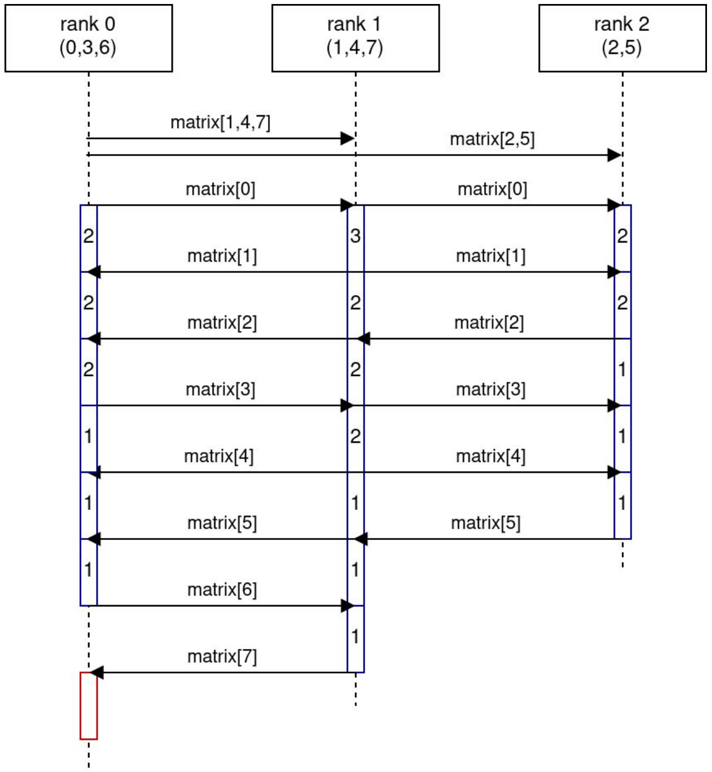


Figure: Spline (with k=3) displaying ratio communication vs. computation using the same procedure as in the OMP part.

Hybrid - Parallelized implementation and approach

- Built on our current best approach
- Addition of OMP similar to OMP part



Hybrid - Final Performance Results

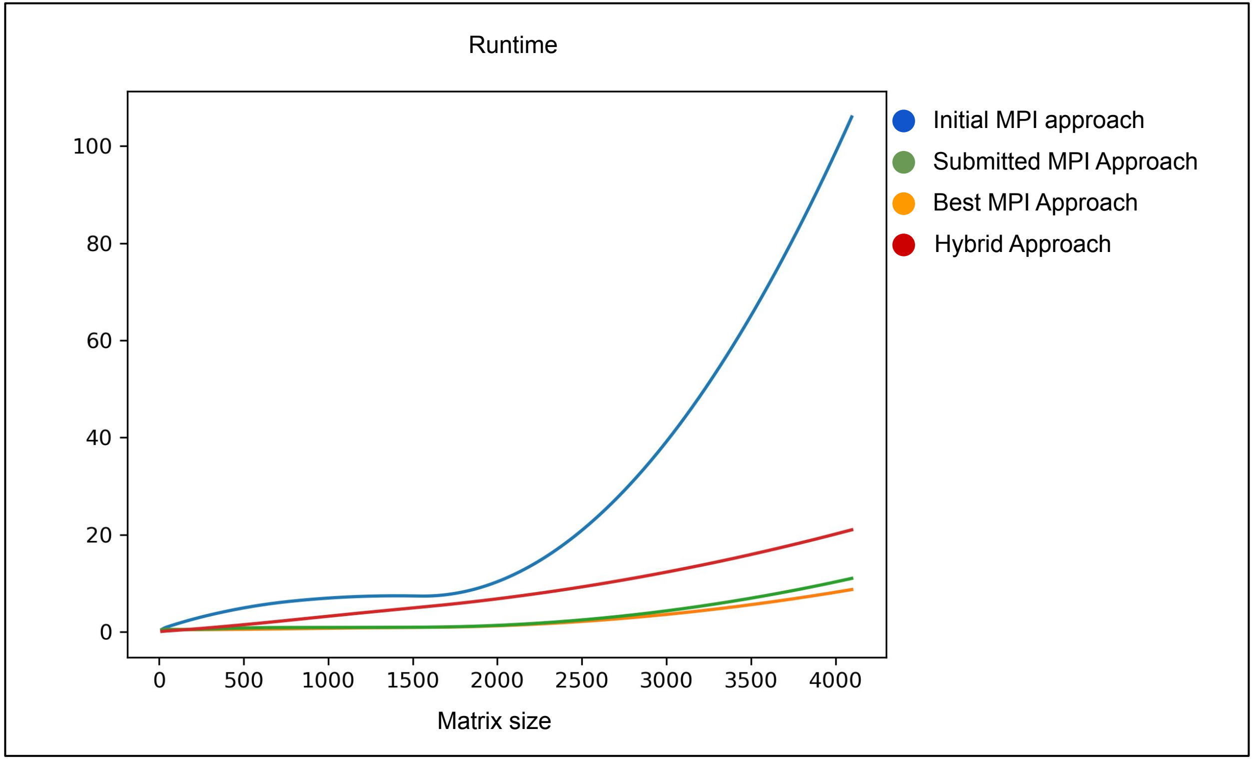


Figure: Runtime comparison between MPI

Hybrid - Communication vs. Calculation

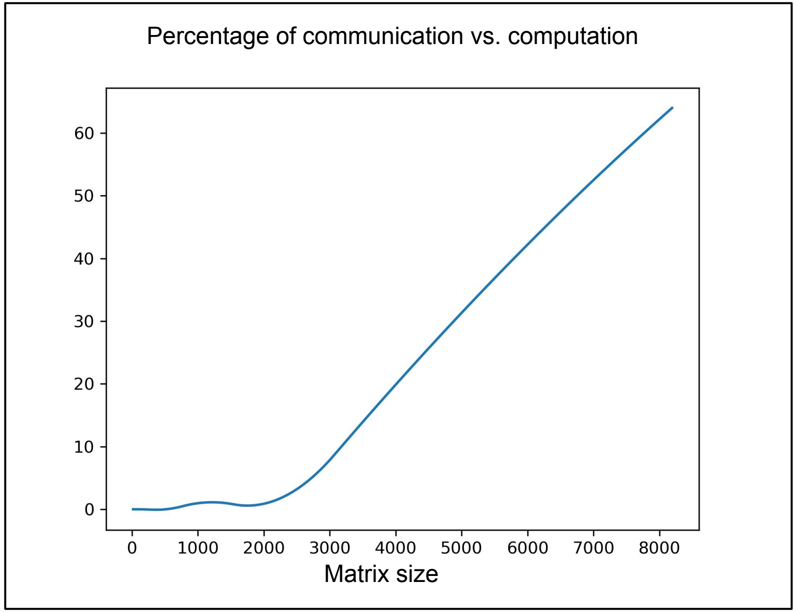


Figure: Spline (with k=3) displaying ratio communication vs. computation using the same procedure as in the OMP part.

Bonus - Parallelized implementation and approach

- Parallelized the inner loop using SIMD intrinsics
- Using `double`, we can calculate 4 values at once
- Achieved no speed-up
- Using -O3 flag compiler already figures out what to vectorize

```
Samples: 4K of event 'cycles', 4000 Hz, Event count (approx.): 4661322483
Serial::ForwardEliminationOptimized  /home/marcelbraasch/CLionProjects/openmp/serialge [Percent: local period]
  2,55 │       vinsertf128  $0x1,0x10(%rbx,%rax,1),%ymm7,%ymm1
  7,37 │       vmovupd      (%rdx,%rax,1),%xmm6
 12,54 │       vinsertf128  $0x1,0x10(%rdx,%rax,1),%ymm6,%ymm0
 17,44 │       vmulpd       %ymm3,%ymm1,%ymm1
 16,41 │       vsubpd       %ymm1,%ymm0,%ymm0
  2,45 │       vmovups      %xmm0,(%rdx,%rax,1)
  6,23 │       vextractf128 $0x1,%ymm0,0x10(%rdx,%rax,1)
 31,91 │       add          $0x20,%rax
  2,21 │       cmp          %rsi,%rax
```

Figure: Compiler already vectorizing element-wise operations.

- Using 32 threads, vectorizing rows by a factor of 4
- Theoretically, should yield a work split of 4 × 32 = 128
- By Amdahl's law, theoretical max speed up is then
  - $1 / (1 - 0.97 + 0.97/(4 \times 32)) = 26.61$

Conclusion

- OMP outperforms all other up to 2048
- MPI is faster for matrices with n ≥ 4092
- Hybrid did not work very well compared to MPI
- The best we got was a speed up of 14
- Amdahl’s law indicates a theoretical max of 17
- The compiler is already vectorizing matrix operations

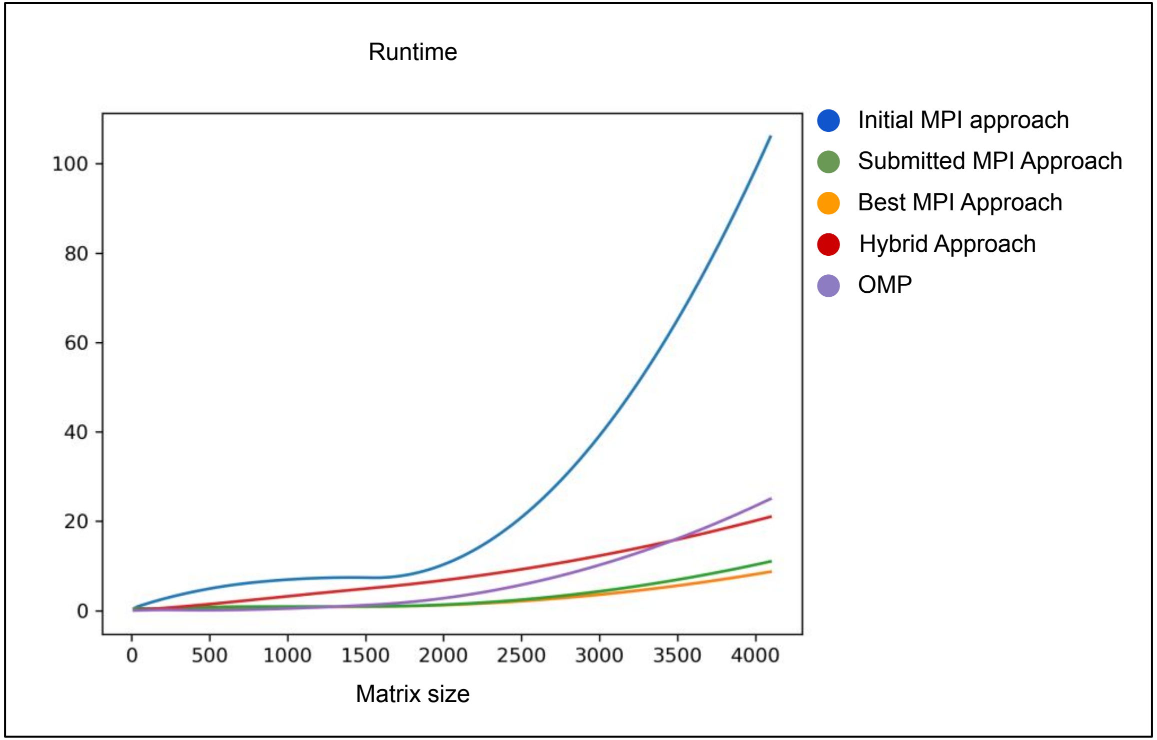


Figure: Runtime comparison between MPI

Additional

System specifications we performed the tests on:

- OS: Ubuntu 20.04.2 LTS
- RAM: 32 GB
- CPU: AMD Ryzen 9 5950X
- GPU: Nvidia GeForce RTX 3090