PARALLELISATION OF SELECT MATRIX MANIPULATION ALGORITHMS

Sina Klampt, Niall Siegenheim, Luca Wolfart, Dana Zimmermann

ETH Zürich
Zürich, Switzerland

ABSTRACT

This project focused on the parallelisation of two PolyBench matrix manipulation algorithms, namely the Jacobi 2D stencil computation and the LU decomposition. The goal was to implement three different approaches: Message Passing Interface (MPI), Open Multi-Processing (OpenMP), and a combination of both MPI and OpenMP (Hybrid) and compare them with each other. In the case of LU decomposition we also compared the performances to the ScaLAPACK implementation. Overall, the results using MPI performed better than the implementations with OpenMP over all ranks for both algorithms. Additionally, the performance of our MPI implementation for the LU decomposition converged towards the ScaLAPACK implementation. However, our Hybrid implementations were inconsistent throughout and not comparable to the other parallel approaches.

1. INTRODUCTION

Over the last two decades parallelising sequential algorithms has acquired greater significance since multi-core processors have become more mainstream and the potential of frequency scaling has been exhausted.

Motivation. Different parallel versions of two select matrix manipulation algorithms were implemented. First, the Jacobi 2D stencil computation that is often used for solving a core part of scientific simulations, namely partial differential equations, using finite difference methods on a two-dimensional regular grid. Second, the LU decomposition, a common and computationally advantageous scientific computing method. Moreover, this was done in a relatively straight-forward way as the motivation was to show how simple implementations will perform compared to different state-of-the-art libraries used for the parallelisation of sequential algorithms, such as ScaLAPACK. All of the implementations can be found in our Git repository.[1] In order to compare performances of the implementations scientific benchmarking, an elegant means of measuring the efficiency of high performance methods, was used. Scientific benchmarking is also utilised to rank the top 500 supercomputers in the world, using LU decomposition with partial row pivoting.[2]

Related work. As a foundation for the implementations PolyBench [1], a collection of benchmarks for many numerical computations with static control flow, was used. The parallel algorithm found in "Parallel Scientific Computation: A Structured Approach using BSP and MPI" [2] was employed for the implementation of the LU decomposition. The algorithm is comprehensible, additionally, there are no benchmarks of the implemented version available, making it a very suitable source to base the implementation on. Similar work for the Stencil computation can be found in an article about compiling stencils in high-performance Fortran [3]. For benchmarking strategies, we used [4] and [5] as reference points.

2. BACKGROUND

This section focuses on the definition and use of the Jacobi 2D stencil computation and the LU decomposition and highlights their importance. Furthermore, the computations performed in PolyBench kernels from a mathematical stand point are explained. Lastly, we will illustrate the programming APIs which we used and explain some concepts in the realm of parallel computing which we will use in our algorithms.

Jacobi 2D stencil computation. The process of updating a matrix following a 5-point stencil.

The point inside the grid is simply computed by taking the average of five points [1]:

$$data_{i,j}^{t} = 0.2 \cdot (data_{i,j}^{t-1} + data_{i-1,j}^{t-1} + data_{i+1,j}^{t-1} + data_{i,j-1}^{t-1} + data_{i,j+1}^{t-1}) \quad (1)$$

In our benchmarks, we repeat this kernel t_{total} times.

LU decomposition. Factorisation of a matrix A into a lower triangular matrix L and an upper triangular matrix U.

The linear algebra solver `lu` in PolyBench [1] is a LU decomposition without pivoting. It takes an $n \times n$ matrix A as input and gives two $n \times n$ matrices L (lower triangular

[1] https://gitlab.ethz.ch/sniall/dphpc-project

[2] See https://www.top500.org/ for a list and further details.

matrix) and U (upper triangular matrix) as outputs, such that $A = LU$.

To avoid round off errors during the row reduction process when the pivot is extremely small, we use partial row pivoting. Before moving to a new pivot column, we first check where in the current column the entry with the maximum absolute value is. We then switch rows to put this entry as our new pivot and continue with the row reduction process [6]. To track these row swaps, we introduce a permutation matrix P, which leads us to the decomposition $PA = LU$. An entry $P_{ij} = 1$ indicates that rows i and j in A have been swapped. All other entries in P are equal to zero.

A sequential implementation of the LU decomposition can be formulated iteratively for $k \in \{1, \ldots, n\}$. It is an important observation that the decomposition can be constructed in-place, with L in the lower diagonal part and U in the upper diagonal part of the original matrix A. The permutation matrix P is initialised as identity matrix. The following steps describe an iteration k.

(1) Find the row $r \geq k$ with largest absolute value in column k. This element will be the pivot.

(2) Swap rows r and k in A and adjust the permutation matrix P accordingly.

(3) Divide pivot column k by the pivot element: $a_{ik} := a_{ik}/a_{kk}, i \in \{k, \ldots, n\}$

(4) Update matrix elements: $a_{ij} := a_{ij} - a_{ik}a_{kj}$, $i, j \in \{k + 1, \ldots, n\}$

The complexity of solving a linear system of equations $Ax = b$ using Gaussian elimination is $\mathcal{O}(n^3)$. This decomposition also has a complexity of $\mathcal{O}(n^3)$ due to step (4). However, once the matrix A is decomposed as $PA = LU$, the linear system of equations can be repeatedly solved with an arbitrary right-hand side $b \in \mathbb{R}^n$ with a complexity of $\mathcal{O}(n^2)$ using forward and backward substitution.

MPI. The Message Passing Interface (MPI) is a standard specifying routines to build parallel distributed memory programs. MPI programs have to be executed using `mpirun`, which will spawn as many threads as passed as an argument. All threads execute the program in parallel. MPI offers simple send and receive message passing methods between threads, but also more advanced methods, such as broadcasting and reductions are available. Various implementations exist for C and Fortran (e.g. Intel MPI, OpenMPI).

OpenMP. The Open Multi-Processing API (OpenMP) allows us to parallelise code using the fork-join model. This means that a primary thread can fork into sub-threads in specified regions of the code. OpenMP offers a convenient and simple syntax using compiler directives. For example, loops which are bound by a fixed number of iterations during run time can be parallelised using a single directive called `omp parallel for`. OpenMP is targeted towards shared memory systems and cannot be run on distributed memory systems. It is supported by C, C++ and Fortran.

Combining MPI and OpenMP. So-called *Hybrid* binaries allow employing both MPI and OpenMP together. The idea is that within a node, where cores share memory, threads use OpenMP for parallelisation. Between nodes, threads use MPI to communicate. The goal is to put the two frameworks to work in the domains they are designed for, hoping this will lead to speed-up compared to just using one of the two frameworks individually.

Processor grid. In parallel computing, processors can conceptually be placed on a two-dimensional Cartesian grid. This means that every processor gets a row and column index based on its rank (i.e. a serial identification number). One may then speak of rows and columns of processors. This processor grid can then be used to map processors to matrix elements using a matrix distribution.

Matrix distributions. When parallelising a matrix algorithm, one may use a matrix distribution to assign an owner in the form of a thread to each element. Then, this distribution is also used to specify the memory location of each matrix element in the case of distributed memory systems.

Given p cores, a block distribution can be defined as distributing a vector $\mathbf{v} \in \mathbb{R}^n$ in contiguous blocks of $b = \lfloor n/p \rfloor$ across all cores. In the case where n is not perfectly divisible by p, one will usually additionally assign all remaining elements to the last core in the row. This can create serious load-balancing issues.

A cyclic distribution can be defined as handing one vector element in a turn-based manner to each processor. This will create perfect load-balancing. However, this distribution should only be employed in distributed memory environments such that for each core its dedicated matrix elements are spatially coherent in memory. Otherwise, its usage will lead to a great number of cache misses.

BLAS. BLAS stands for Basic Linear Algebra Subprograms and is a library, which contains many routines for common basic matrix and vector operations. We chose this library to perform our matrix operations due to its efficiency and robustness.

ScaLAPACK. ScaLAPACK is a scalable version of LAPACK (Linear Algebra Package), which is a library containing more advanced linear algebra operations and is built using BLAS. Similarly, ScaLAPACK is built using the parallelised distributed memory version of BLAS called BLACS. Under the hood, ScaLAPACK uses MPI. We used Intel's implementation of ScaLAPACK and BLACS, which is con-

sidered state-of-the-art.

3. PROPOSED METHOD

In this section, we outline the different parallel versions we implemented for the Jacobi 2D stencil computation and the LU decomposition. This includes implementations of each algorithm using MPI, OpenMP, and a Hybrid using both OpenMP and MPI.

Stencil computation using MPI. Our parallel MPI implementation of the Jacobi 2D stencil follows directly from the sequential. Here, we chose a block distribution for the matrix. Every thread performs the serial 2D Jacobi computation on the local matrix. Each local matrix also has two auxiliary rows (at the top and bottom) and two columns (on the left and on the right) to hold the elements it will receive from its neighbours, also known as *ghost cells*. This is because the elements on the boundary need to know what their neighbours are in the global matrix to compute the average.

At the beginning of every iteration, every thread sets up communication with its respective neighbours using non-blocking send and receive methods. Then, it computes the stencil on the inside of its domain. Finally, it waits until it has received all of the needed ghost cells from its neighbours and computes the stencil on the boundary.

Stencil computation using OpenMP. For the OpenMP implementation of the Jacobi 2D Stencil Computation we simply use a `parallel for` construct with a `collapse` clause to parallelise the two nested loops over all matrix elements. Conveniently, all matrix element updates can be performed independently from each other. Furthermore, one may use the new `simd` clause in OpenMP 4.0 to make use of platform-specific vector instructions.

Stencil computation using OpenMP and MPI. OpenMP can easily be applied to our existing MPI implementation. In total, each iteration has at most five loops: One double loop for the inside elements, and then a maximum of four single loops for the elements on the boundary of the local matrix, depending on how many neighbours the thread has. Each of these loops can be parallelised using the already explained `parallel for` OpenMP-specific compiler directive.

It is important to note that the parallelisation by OpenMP and MPI is strictly separated. There is no MPI communication in parallel OpenMP regions. All MPI communication is performed by the master thread, as seen from the OpenMP scope. This design is intended to be simple and to avoid errors, which are difficult to debug.

LU decomposition using MPI. Our implementation is based upon a parallelisation strategy proposed in the book "Parallel Scientific Computing: A Structured Approach Using BSP." by Rob H. Bisseling [2].

Our code builds on the author's considerations for distributed-memory systems and is implemented using MPI. As outlined in section 2, the algorithm consists of a loop with $k \in \{1, \ldots, n\}$. During every iteration k, only the elements $(A)_{k:n,k:n}$ are modified, where A is the initial matrix. Therefore, for optimal load balancing, it is crucial to employ a cyclic matrix distribution. For the indexing of the processor grid, we make use of convenient MPI routines which allow direct access to rows and columns of processors on the grid using special *communicators*. Collective communication routines can then be called for specific processors in a specific row or column, instead of *all* available processors. In the case of a block distribution, increasingly many threads would stall as the algorithm progresses in iterations, as none of the matrix elements dedicated to them would be modified. For memory-efficient reading and writing of the input and output, we make use of MPI-specific input-output routines which allow efficient and easy memory access for cyclically-distributed matrices. Furthermore, we use BLAS routines as much as possible to speed up trivial operations, such as matrix updates and copying of elements.

Step (1) of the algorithm is performed by each processor which owns elements in column k. First, they perform a local search for the row with the largest absolute value using the BLAS routine `idamax`. Then, all of those processors perform two collective gathering operations, such that all of them obtain a list of the absolute maximum value and its index across all threads in the column. Once again, using `idamax`, each processor finds the global absolute maximum and its index in that column. Finally, each thread broadcasts this information to all other threads in the thread row. In the end, all threads know the location and value of the pivot element.

In step (2), the processor which own elements in rows r and k will concurrently swap these, such that the pivot row is located in the correct location.

To normalise the pivot column in step (3), the processors in that column can then use the BLAS routine `dgemm` to scale the column elements appropriately. Even though `dgemm` is originally meant for matrix-matrix multiplication, it can also be used for various other arithmetic manipulations of matrices.

Lastly, in step (4), column $(A)_{k,k:n}$ and row $(A)_{k:n,k}$ are broadcasted according to the matrix distribution so every thread can perform the update $a_{ij} := a_{ij} - a_{ik}a_{kj}$, again using `dgemm` for optimal performance.

LU decomposition using OpenMP. As described in section 2, each iteration of the algorithm consists of four loops. Each loop is well-defined and can therefore be conveniently parallelised using OpenMP's `parallel for` compiler directive. The `shared`, `private` and `lastprivate` arguments allow for defining shared and private variables between the threads. Ad-

ditionally, the new `simd` argument in OpenMP 4.0 may be used to make use of architecture-specific SIMD instructions.

LU decomposition using MPI and OpenMP. Our MPI implementation of the LU decomposition uses BLAS calls for various operations. Conveniently, the Intel MKL implementation of BLAS has built-in OpenMP support. Therefore, we can retrieve a Hybrid binary by compiling our MPI implementation with the `-fopenmp` flag and linking the correct Intel MKL implementation.

4. EXPERIMENTAL RESULTS

Experimental setup. All experiments were conducted at ETH Zürich on the high-performance cluster nodes "Euler IV" with two 18-core Intel Xeon Gold 6150 @ 2.70 GHz processors each. These nodes are connected by an 100 GB/s InfiniBand EDR network and run CentOS 7 with Linux kernel version 3.10.0. To make the most of the available hardware, we used the Intel C Compiler (`icc`) and its MPI counterpart (`mpiicc`), version 2018.1, with compilation flags `-O3` and `-march=skylake-avx512` and appropriate linking to the MPI and OpenMP runtime libraries. Furthermore, all calls to the BLAS and ScaLAPACK used the Intel MKL implementation, also version 2018.1.

As MPI and OpenMP are targeted towards different memory systems, we chose different experimental designs when benchmarking the respective implementations. For the MPI benchmarks, all ranks were located on distinct nodes to enforce a distributed memory environment. In contrast, for the OpenMP benchmarks, all threads were located on the same node.

Hybrid benchmarks display a special case where each MPI rank gets a fixed number of OpenMP threads assigned. To that end, the total number of processors benchmarked p has to be decomposed into two factors: $p_{\text{total}} = p_{\text{MPI}} \cdot p_{\text{OpenMP}}$, where $p_i \in \mathbb{N}, i \in \{\text{total}, \text{MPI}, \text{OpenMP}\}$. We chose these two factors to be as close to each other as possible, where the greater factor was assigned to MPI. Hence, in the case where the total number of processors is prime, the Hybrid benchmark is equivalent to the MPI benchmark as $p_{\text{OpenMP}} = 1$.

Specifically for the LU decomposition, we also examined the performance of ScaLAPACK to have a benchmark to compare our implementations against. We chose the same experimental design as for the MPI benchmarks, as it is also geared towards distributed memory systems.

In strong scaling benchmarks, the problem size stays constant across all benchmarks. A lower bound for strong scaling is given by $t(p) = t_{\text{seq}}/p$, where $t(p)$ is the parallel runtime given p number of available cores, and t_{seq} is the runtime of the sequential algorithm. In weak scaling benchmarks, the problem size scales according to the available number of processors. In this case, good scaling is indicated by approximately constant runtimes across all numbers of threads. For the Jacobi 2D stencil computation, we scale the size of the dimensions the input matrix $A \in \mathbb{R}^{n \times n}$ with a factor of $\sqrt{p}$, but we do not scale the number of time steps, since the computation across time steps is not parallelised. For the LU decomposition we scale the size of the dimensions of the input matrix $A \in \mathbb{R}^{n \times n}$ with a factor of $\sqrt[3]{p}$.

We define a benchmark as the timing of a specific implementation for a given number of available threads $p \in \{1, 2, \ldots, 36\}$. Unfortunately, it was not possible to scale beyond 36 processors due to user limitations on the high-performance cluster. Every benchmark was repeated 25 times in the same environment (same node, etc.).

Results. Here, we discuss the benchmarking results of both the LU decomposition and the Jacobi 2D stencil computation. The first plot for each algorithm shows the strong scaling results of our implementation, whereas the second plot represents the weak scaling results. In both cases, we plotted the runtime as a function over the number of processors p represented in a semi-logarithmic graph. Points in the graphs indicate the median of the benchmark, and error bars indicate the 95% confidence intervals around the median, computed according to [5]. In most cases, the confidence intervals are so small they are not visible.

First, we look at the plots of the Jacobi 2D stencil computation. We compared the results of the MPI, Hybrid, and OpenMP implementations.

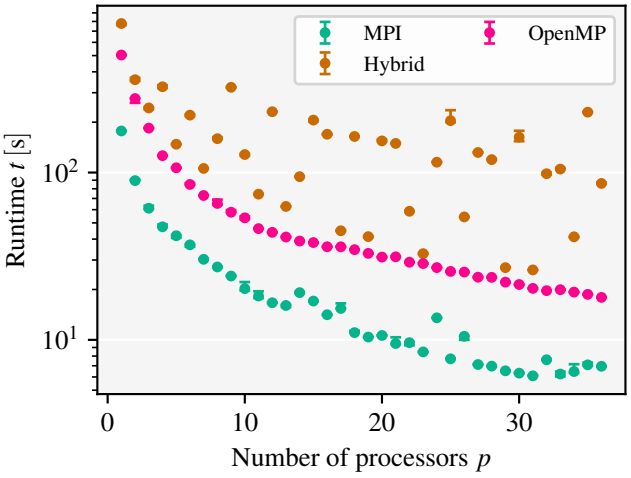

Fig. 1. Benchmark of our implementation of the Jacobi 2D stencil computation, strong scaling for $n = 8192$, $t_{\text{total}} = 1000$.

In Figure 1, we can see the strong scaling results of the implementations mentioned above. Our MPI implementation is about two to three times faster than our OpenMP

implementation over the whole range of available processors. Since both implementations are similarly optimised regarding cache locality and instruction-level parallelism, we attribute this to a lower parallelisation overhead in MPI compared to OpenMP. In particular, MPI gives more control over the distribution of matrix elements than OpenMP given its more explicit nature. Furthermore, both implementations scale well. Therefore, it is likely our MPI implementation has a better cache access pattern. Even for a high number of available ranks, runtimes continue to improve. This indicates that neither of the implementations suffer from a communication bottleneck. Given the trend, we can assume that even for a higher number of threads, runtimes would continue to decrease.

For the Hybrid implementation, we observe erratic runtimes significantly worse than both the MPI and OpenMP implementations. Our explanation for the poor and inconsistent performance is that the Euler cluster is not properly configured to run Hybrid jobs, especially when using Intel MPI runtime libraries. The user guide for the cluster strongly discourages from running Hybrid jobs on the cluster.[3]. Therefore, in our further analyses, we will not go into further detail about the results of the Hybrid implementation.

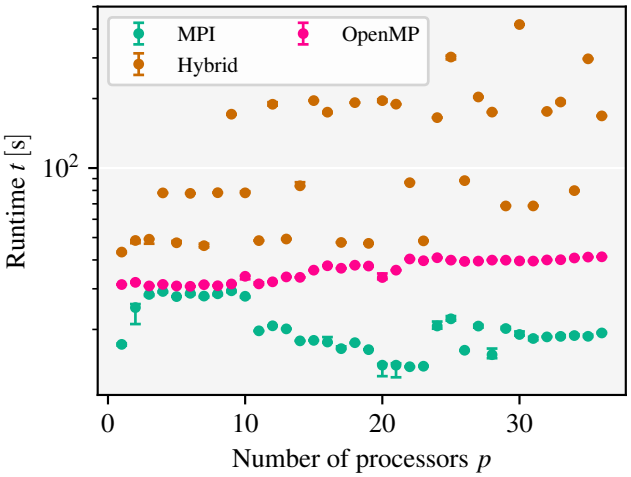

Fig. 2. Benchmark of our implementation of the Jacobi 2D stencil computation, weak scaling for $n_{\text{base}} = 2048$, $n(p) = n_{\text{base}}\sqrt{p}$, $t_{\text{total}} = 1000$.

Figure 2 shows the comparison of weak scaling for the various Jacobi 2D stencil computations. We can see that the OpenMP implementation scales well over all cores, as the runtime stays between 20 s to 30 s. However, runtime seems to slightly increase as the number of available cores increases. This may be attributed to communication overhead, once again. In contrast, we observe that our MPI implementation shows some inconsistencies. Strikingly, the runtime decreases after being constant for up to ten available cores. For $p > 10$, the runtimes are significantly lower by a factor of about 2 and higher than compared to $p \leq 10$. This may be due to load differences on the cluster between benchmarks, as our implementation does not offer any explanation why in weak scaling the performance should suddenly double when going from 10 to 11 cores.

Now, in figure 3 and figure 4, we look at the performance plots of the different implementations of LU decomposition. We compared the results of the MPI, Hybrid, ScaLAPACK and OpenMP implementations.

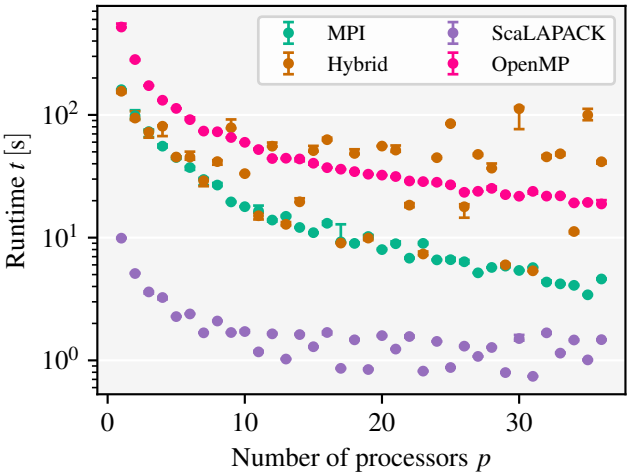

Fig. 3. Benchmark of our implementation of the LU decomposition, strong scaling for $n = 8192$.

In figure 3, the strong scaling results of our implementations are depicted. Overall our observations were similar to those of the Jacobi 2D stencil computation. Our MPI implementation is again faster than our OpenMP implementation over all threads as expected for a large matrix. In this case, this difference can not only be attributed to less communication overhead, but also to being computationally more efficient due to the use of BLAS routines for various operations. Both MPI and OpenMP scale well up to 36 threads and their trends lead to the assumption that this will continue for an even higher number of threads.

The runtime of the ScaLAPACK version of the LU decomposition is significantly better than our implementations over all ranks. It shows once again that the algorithms in these libraries are optimised to the fullest and that it is difficult to achieve similar performance, especially given that the software engineers at Intel may have more insight into the hardware functionality of Intel processors than what is publicly available. Interestingly, this implementation performs about 10 % to 20 % better when p is odd than when it is even. This is a surprising observation, as especially for

[3] https://scicomp.ethz.ch/wiki/Using_the_batch_system#hybrid_jobs

small p most odd numbers are prime and therefore do not lend to good two-dimensional matrix distributions.

While ScaLAPACK is quicker by an order of magnitude compared to our MPI implementation for a small number of cores, this difference gradually diminishes for an increasing number of cores. This is because the ScaLAPACK runtimes saturate for about $p > 12$, while the MPI implementation continues to scale. Most likely, at that point, the matrix is too small for ScaLAPACK to scale and the additional computational resources by adding more cores are equalised by the additional communication overhead.

Regarding the Hybrid version, we observe the same issue as in the case for the Jacobi 2D stencil computation. It only performs closely to the MPI implementation for prime number of ranks, as in this case no OpenMP-related parallelisation is performed.

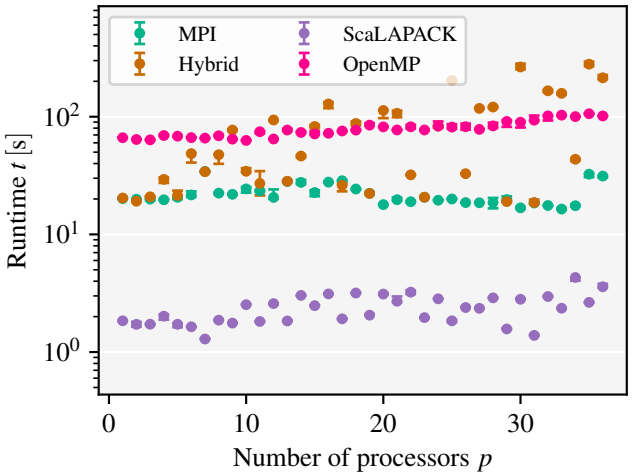

Fig. 4. Benchmark of our implementation of the LU decomposition, weak scaling for $n_{\text{base}} = 2048$, $n(p) = n_{\text{base}} \sqrt[3]{p}$.

Finally, figure 4 shows the weak scaling behaviour of our implementations for the LU decomposition. We can see that the runtime for both the OpenMP as well as the MPI implementation stay mostly constant with increasing problem size and therefore scale well over all ranks. It can again be seen that the ScaLAPACK implementation has a much faster runtime than our implementations with a ten-fold speedup compared to the MPI implementation and about a 15-fold speedup compared to the OpenMP implementation.

5. CONCLUSIONS

This project focused on the parallelisation of the Jacobi 2D stencil computation and the LU decomposition. We implemented different approaches and compared them with each other. In the case of LU decomposition we additionally compared their performance to the ScaLAPACK implementation.

We saw in the previous chapter that our relatively simple MPI implementation can almost reach the same runtime as the ScaLAPACK implementation. The performance for higher ranks could not be evaluated, nevertheless we can see the MPI runtime converging to the ScaLAPACK runtime.

Another observation that we made is that the performance for the significantly more complicated MPI implementation was better than our OpenMP implementation, both for the Jacobi 2D stencil computation and for the LU decomposition. Depending on the goal that needs to be achieved, this should also be taken into consideration. In most situations, we would recommend to use OpenMP as it is simpler to implement while still providing good results. For large matrices however, using MPI will lead to much better scaling.

Overall, we got similar results for both algorithms with the MPI implementation performing better than the OpenMP and the Hybrid version not being comparable.

To further improve the performance of our implementations, we would suggest to analyse the mathematical properties of the algorithms in more detail, especially the LU decomposition, and parallelise those parts.

The last point we want to mention is the poor performance of our Hybrid implementation. We would therefore like to suggest towards the Euler support team to add (well-documented) support for hybrid problems.

6. REFERENCES

[1] T. Yuki and L.-N. Pouchet, "Polybench 4.2.1 (pre-release)," Tech. Rep., Inria/LIP/ENS Lyon, Ohio State University, 2016.

[2] R. H. Bisseling, *Parallel Scientific Computation: A Structured Approach using BSP and MPI*, Oxford Scholarship Online, 2004.

[3] Gerald Roth, John Mellor-Crummey, Ken Kennedy, and R. Gregg Brickner, "Compiling stencils in high performance fortran," in *In Supercomputing '97: Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*. 1997, pp. 1–20, ACM Press.

[4] T. Hoefler and R. Belli, "Scientific benchmarking of parallel computing systems - twelve ways to tell the masses when reporting performance results," Tech. Rep., ETH Zurich, Zurich, Switzerland.

[5] Jean-Yves Le Boudec, *Performance Evaluation of Computer and Communication Systems*, EPFL Press, Lausanne, Switzerland, 2010.

[6] S. Andrilli and D. Hecker, *Elementary Linear Algebra*, Academic Press, 5th edition, 2016.