POLYBENCH: OPTIMIZING DOITGEN AND JACOBI BY HAND

Antoine De Gendt, Quentin Guignard, Marie Jaillot, and Gilles Waeber

Department of Computer Science, ETH Zürich
Zürich, Switzerland

ABSTRACT

We present here implementation concepts optimized for parallelism for selected polybench algorithms: doitgen, jacobi1d and jacobi2d. We explore both a thread-level parallelization using OpenMP and process-level parallelization using MPI to determine the main optimization axis for those problems.

1. INTRODUCTION

Motivation. The goal of this project is to optimize and parallelize benchmarks in the Polybench benchmark suite in order to give a baseline for "human" optimization. We chose the Doitgen kernel and the Jacobi stencil because they represent real world problems and common workloads. Today, parallelization is necessary to fully exploit the capabilities of the hardware. Two APIs will be used : OpenMP and MPI[1]. Our main contribution is a baseline for comparison.

Related work. Polybench[1] provides simple sequential implementations for these algorithms. In [2], some of these benchmarks are automatically parallelized.

2. BACKGROUND: SELECTED ALGORITHMS

Doitgen. This algorithm is performing a series of matrix matrix multiplication given $A \in \mathbb{R}^{\mathrm{NR}\times\mathrm{NQ}\times\mathrm{NP}}$ and $C4 \in \mathbb{R}^{\mathrm{NP}\times\mathrm{NP}}$ and is defined as $A(r,q,p) = \sum_{s=0} A(r,q,s) * C4(s,p)$.

Stencil algorithms. Those where we repeatedly update values in an array according to a pattern, i.e. stencil. Parallel implementations of such algorithms are challenging due to the coupling between a cell and its neighbors. The dependency on the previous value of the cell itself also prevents parallelizing over the time.

Jacobi 1D. Given an array $A \in \mathbb{R}^n$ and a number of steps t, we replace t times A with A' with $A'_1 = A_1, A'_n = A_n, A'_i = \mathrm{mean}(A_{i-1}, A_i, A_{i+1})$ for $i \notin \{1, n\}$.

Jacobi 2D. Given a 2D array $A \in \mathbb{R}^{n\times n}$ and a number of steps t, we replace t times A with A' with $A'_{i,j} = A_{i,j}$ for $i \in \{1, n\} \vee j \in \{1, n\}$, i.e. the border cells, and $A'_{i,j} = \mathrm{mean}(A_{i,j}, A_{i-1,j}, A_{i+1,j}, A_{i,j-1}, A_{i,j+1})$ otherwise.

[1]Message Passing Interface

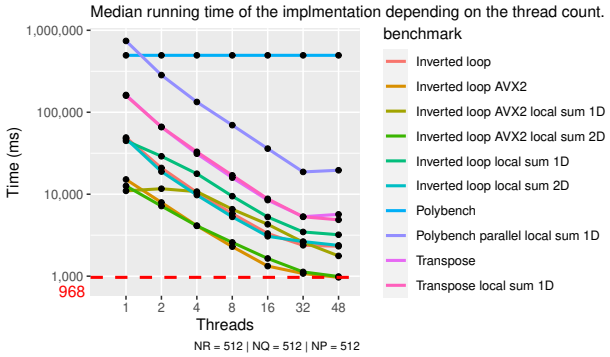

Fig. 1: Comparisons of several implementations for Doitgen.

Verification. To ensure the correctness of the proposed implementation, we tested against the reference Polybench code.

Generating data. To generate the data, we collected 10 runs for each parameter modification. Then, the data was plotted using functions provided in the R scripts of the LibSciBench [3] repo.

3. EXPERIMENTAL SETUP

All our benchmarks are run on Euler VII nodes (CPU: AMD EPYC 7H12, 64 cores). For this project, we were limited to 48 cores and could not obtain exclusive access to a node, which means other jobs could potentially be using the same nodes at the same time, affecting the results. The software stack is gcc/9.3.0+openmpi/4.0.2. Compilation options were -O3 -mavx2 -mfma. Unless otherwise specified, each experiment was run 10 times.

4. DOITGEN

4.1. OpenMP

4.1.1. Proposed methods

In this section, we will talk about the parallelization and optimization of the doitgen kernel using the Open-MP API. We parallelized Doitgen as follows : one thread computes a sequence of **complete** matrix-matrix multiplications. More

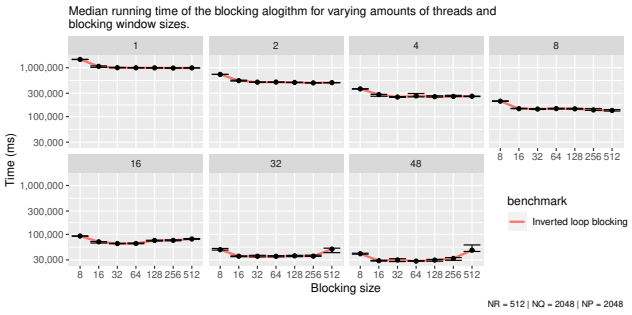

Fig. 2: Study of the blocking algorithm with different parameters. Each plot corresponds to a specific number of threads.

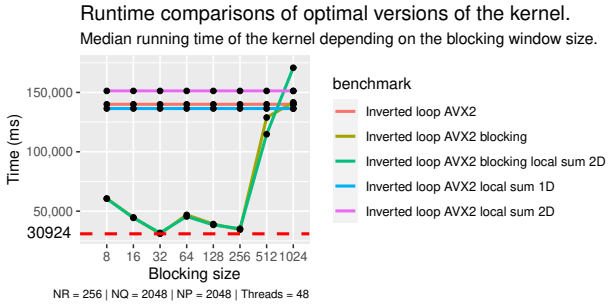

Fig. 3: The dashed line represents the smallest running time.

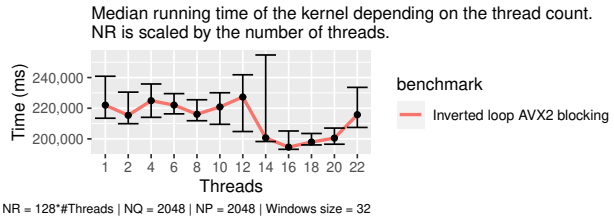

Fig. 4: Study of the inverted loop AVX2 blocking implementation with scaling.

concretely, the parallelism occurs along the NR axis of the input matrix. To extract the most performance out of our CPU, we need to leverage the cache. To explore the impact of better cache utilization, we implemented a version of the algorithm that uses a transposed C4 matrix, a version where the two inner loops are inverted and a blocking version.

Local sum. The initial version of the kernel provided by Polybench does not use an input and output matrix, only an input matrix is used. Then, in the kernel, the elements of a resulting line are incrementally built up and stored into an array that is then put back in the input matrix. This way of performing the computation consumes $\mathrm{NR}\times\mathrm{NQ}\times\mathrm{NP}+\mathrm{NP}$ memory rather than $2 \times \mathrm{NR} \times \mathrm{NQ} \times \mathrm{NP}$ when an output matrix is used. The difference between the versions with a local sum array and without will be explored in the following section. Also, in the parallel algorithms, each thread has its own sum array, resulting in a $\#\mathrm{Threads} \times \mathrm{NP}$ table when a 1D sum array is used and in a $\#\mathrm{Threads}\times\mathrm{NQ}\times\mathrm{NP}$ table when a 2D sum array is used.

Transpose version. In this version, the C4 matrix is transposed, it is thus traversed row by row.

Inverted loops version. We incrementally build the elements of a line of the output matrix by traversing A by rows and C4 by rows. Then, we switch to another line of C and another line of A to build the elements of another line of the output matrix.

Blocking version. In the blocking version of the kernel, the goal is to reuse values brought in from cache as much as possible. This is achieved by multiplying sub-matrices rather than multiply one matrix with another at once. This improvement can be combined with one of the two mentioned previously. As currently implemented, this version only works if the blocking size divides the matrix size.

Vectorization. The vectorization version of the kernel uses the AVX2 and FMA instruction sets. The AVX2 instruction set deals with 256 bits registers, allowing us to load doubles (8 bytes) 4 by 4. As currently implemented, this version only works if the matrix row size is divisible by 4.

4.1.2. Experimental results

Here, results of the benchmarking of the different implementations will be presented.

Finding the best implementation. First, we decided to benchmark all the implementations with a rather small matrix size to see what the fastest were. See Figure 1 (beware of the logarithmic scale). We clearly see that the inverted loop AVX2 implementations are the fastest ones.

Blocking window size. Now, we want to know if pursuing blocking is worth it and how it performs under different blocking window sizes and varying amounts of threads. The results are available in Figure 2. We see that the higher the thread count the higher the impact of blocking on performance. This may be because this allows for threads to work on different parts of the matrix, causing less cache coherency traffic and less cache associativity issues.

Finding the best version. With this data we can now conclude that an inverted loop AVX2 implementation with blocking should yield the best results. The outcome of this experiment is available in Figure 3. Blocking improves greatly the performance of the algorithm. It seems to have a greater impact than on single threa-ded applications. The benchmarks appearing on the figure that do not use blocking were not run for each blocking size, they should be read as constant straight lines. Also, the 1D local sum version seems to be faster than the other two. This may be explained once again by cache coherency traffic and cache associativity. In Figure 4[2], the running varies around 220s for different amounts of threads. This may be because the chosen win-

[2]We didn't manage to plot for greater thread counts(the limit is 22) because of memory consumption issues.

dow size is tailored to work well with 48 threads but not necessarily for other thread counts.

4.2. MPI

4.2.1. Proposed methods

In this section we will briefly describe how Doitgen was parallelized with MPI. After that, we will describe our three experiments.

We parallelized Doitgen very similarly as in the OpenMP section. We split the work among processes by attributing to each process a chunk of matrices along the NR axis. Each process is allocating a batch of 64 $\mathrm{NQ} \times \mathrm{NP}$ matrices. It is worth to note that the initial problem size is $128 \times 512 \times 512$ and that we scaled the NR axis with the number of processes. Moreover, our algorithm works in three phases, firstly it initializes A. The second phase is the kernel execution (a full batch in a row). Finally, the last phase consists of writing the batch to a remote location. Note that there is no network communication except when writing the batch. Now that we described our algorithm, let's talk about our experiments.

We made a total of three experiments. The first experiment consisted of comparing four writes strategies along two axis, using or not collective writes and using or not a file window. This was done in order to choose the simplest best write strategy. Secondly, we compared the runtime of each phases of our algorithm and we proposed an improvement by transposing the slices in place before the execution of the kernel. We measured each phases as well as the overall runtime. We also reported the sequential baseline[3]. Finally, we observed the effect of spanning multiple nodes and measured everything exactly as in the second experiment. Now that we have described our experiments, let's see our results.

4.2.2. Experimental results

In this section we will report and discuss our experiments results. We will start with the write strategy.

When comparing the 4 different strategies, we can ask ourselves how much the 4 write strategies affect the runtime. By looking at Figure 5 we can observe that there are two trends, the collective writes are slower than the individual writes and using a window does not affect the results at all. This result follows are intuition since the write regions are disjoints per process and any collective calls imply some synchronization overhead. We therefore choose an individual call and we are now able to compare the proportion of each phase runtime in our second experiment.

Indeed, we need to know which phase dominates the runtime in order to make things faster.

[3]We used the Doitgen implementation of Polybench 4.1

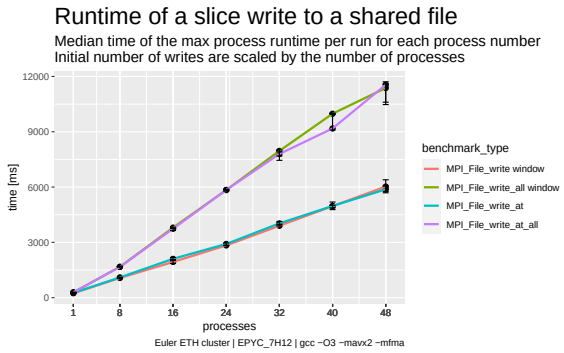

Fig. 5

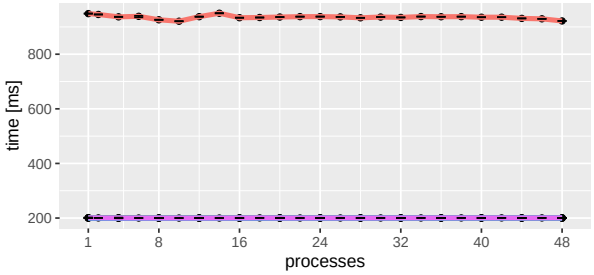

Fig. 6: Runtime of a single iteration of the kernel. Median time per process per iteration for each process number

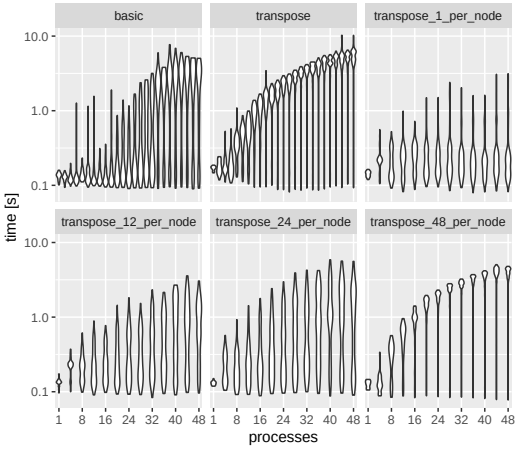

Fig. 7: Runtimes of a batch write. All time measurements per process per iteration for each process number

First, by looking at the *basic* versions of Figure 8, we can observe that the runtime of the algorithm is about 125 seconds and match the baseline as expected. We did not included the plot for the phase 1 because it was always taking around $75ms$ and was therefore insignificant to the rest. For the second phase on Figure 6, we can see that the basic kernel takes about $900ms$ per iteration. For the third phase on Figure 7 and by looking at the basic violin, we can observe that a single batch write can take up to 10 seconds. Therefore, since there is only two write of batch per process and 128 kernel iterations per process, the overall runtime is dominated by the kernel execution. With this information

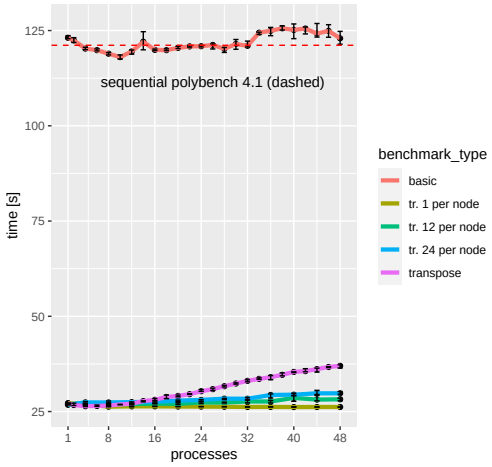

Fig. 8: Overall runtimes with baseline. Median time of the median process runtime per run for each process number.

in mind, we introduce an improvement. Indeed, similarly as in the openMP version, we could ask us how much would the kernel benefits from friendlier memory accesses. In order to answer this question we transposed the batch in place before executing the kernel. We first noted that the phase 1 runtime remained unchanged even with the transposition (still not showed). Secondly, we noted a performance improvement as it can be observed by looking at both the *basic* and *transposed* version on Figure 6 where a single kernel iteration dropped to around $200ms$ and the overall runtime dropped to around $20s$ of runtime. We won't cover much about the cache issues and why its faster as it is already done in the OpenMP section. However, we can note a progressive overhead in the runtime in the *transpose* version still on Figure 8 as the number of processes increases.Could this be the cache? Interestingly, we don't observe the same progressive overhead on each kernel iterations runtime on Figure 6 therefore the kernel phase is not at the origin of this overhead. What about the writes? Indeed, we observe a change on Figure 7, the more nodes we span, the lower the clusters are located which indicates that the writes took less time (Note the logarithmic scale). This progressive overhead could be due from either network bandwidth sharing, from cache contention (not in the kernel) or from disk contention.The latter can be excluded because if this was the disk, spanning more nodes would have no effect. This let us with the network and cache contention in another phase than 2. As said previously, the phase 1 was taking around $200ms$ for a single batch and therefore cannot be guilty for the overhead. We don't see another place where cache contention could build up except in the write because we don't know the implementation of the MPI write. Perhaps MPI is sharing memory for the processes on same node which would incure an overhead when lots of processes are on the same node? Unfortunately, we don't have any experiment for that but that could be aligned with Figure 7 and would

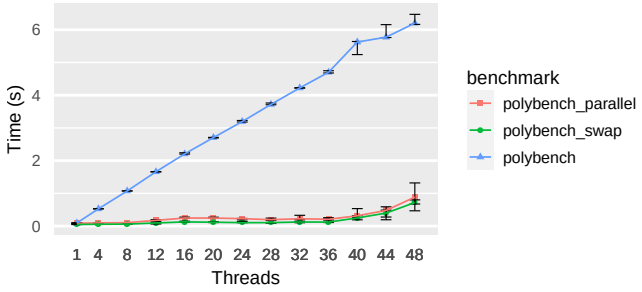

Fig. 9: Jacobi 1D: execution time per number of threads, with $N = 10000*$ number of threads

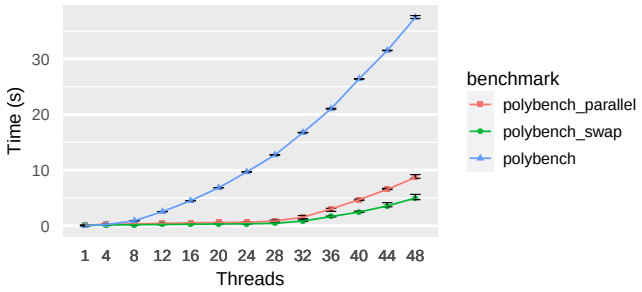

Fig. 10: Jacobi 2D: execution time per number of threads, with $N = 100*$ number of threads

explain the speed up when spanning more nodes. Lastly, it could be the network bandwith sharing from the node to the remote write location. However, doing an approximate computation leads to 200ms per iteration and 64 iteration to write $1072Mbit$ (a batch) imply $89Mbit/s$ and at most $4288Mbit/s$ with 48 processes. This is far from the $100GBits$ bandwith in euler VII to scratch location and therefore could explain the overhead in Figure 8, the speed ups in writes on Figure 7 and the absence of overhead on Figure 6. To conclude, we could optimize and provide a human optimization for Doitgen implemented with MPI but we cannot explain precisely the reason of the overhead except with the above given ideas. One sure thing is that this overhead is due to something on the node, either the network or the cache. Let's now switch to Jacobi.

5. JACOBI

5.1. OpenMP

5.1.1. Proposed methods

In this section, we will present the different parallelizations of Jacobi 1D and 2D achieved with OpenMP.

Jacobi 1D. Since Jacobi iteratively updates the cells of the given array A, we could not parallelize the execution of all the time steps. We thus only parallelized the computation of each cell's new value within a single time step. The first loop, which performs the computation over the cells of array A and stores them in a second array B, is divided between all threads and once this computation is finished, the second

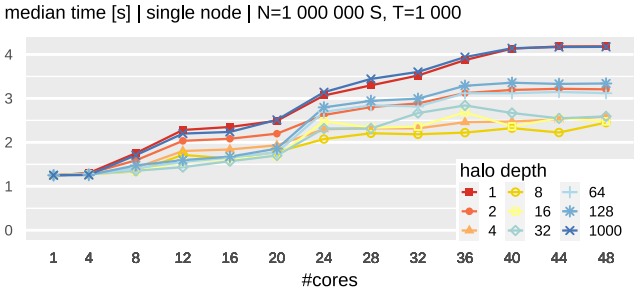

Fig. 11: Jacobi 1D MPI: comparing halo depths

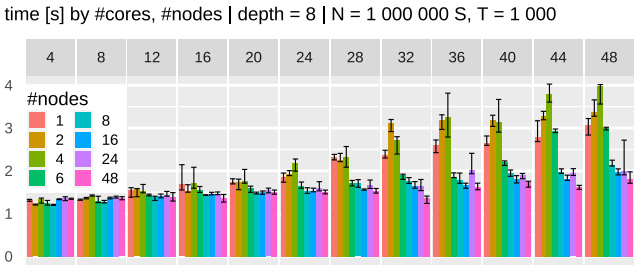

Fig. 12: Jacobi 1D MPI: spanning multiple nodes

loop is also split between the threads to update A with these new values. Another more efficient method was to replace the second loop, responsible for transferring the values from array B to array A, with a simple swap between the two array pointers.

Jacobi 2D. With Jacobi 2D the parallelizations were done in the same fashion. The time steps are executed sequentially and for each time step, the computation of the new values is parallelized over the two dimensions. Again, another optimization was achieved by replacing the transfer of values between matrix A and B by a swap of pointers.

5.1.2. Experimental results

We will now discuss the benchmark results of our different implementations. To test the efficiency of our implementation, we performed 10 runs for each execution parameter set. We used 1000 time steps and an array size of $N = 10000$ and $N = 100$ for Jacobi 1D and 2D respectively, multiplied by the number of processes. In Figure 9 with Jacobi 1D, we see that the two parallelizations achieved better performances, especially the implementation using the pointer swap mentioned above. For Jacobi 2D in Figure 10 the two parallelizations were just as efficient and scale well.

5.2. MPI

5.2.1. Proposed methods

For Jacobi MPI implementations, we assume that each process should only store a small part of the complete array in its memory. Given the dependency between a cell and its

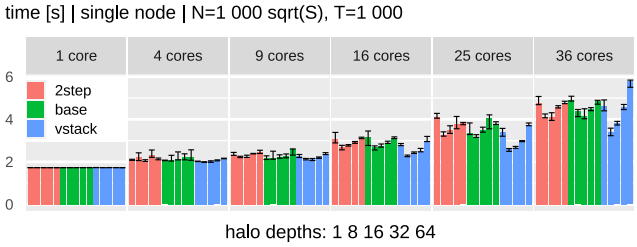

Fig. 13: Jacobi 2D MPI: comparing program variants

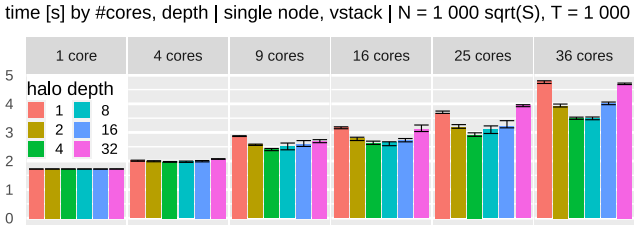

Fig. 14: Jacobi 2D MPI: halo depth for vstack

neighbors, we need to share data between the different processes, a solution being to share data with the neighboring processes when necessary.

Ghost Cells. An applicable pattern here are ghost cells [4]. We store a padded array, slightly larger than the array to compute, and only compute the values of the cells in the central part while the cells at the borders are computed by other processes and exchanged whenever necessary. Concretely, for Jacobi 1D, we store an additional cell at the beginning and at the end of the array, and after every iteration, each process sends the first and the last values it computed to the previous resp. next process, which uses it to update its border cells.

Deep Halo. Extending on the idea of ghost cells, we store a bigger array to reduce the number of inter-process data exchanges required. Thus, data packets are bigger and the relative data transfer overhead is reduced. As the stencil must now also be applied on some border cells too, the total computation increases. We define the halo depth as the number of iterations between every data exchange, equal here to the number of additional "padding" cells on each side of the array. Ghost cells correspond to a depth of 1. A depth greater than the number of iteration means that no data exchange is necessary. Data exchange and computation overhead must be balanced. Greater depth could lead to data exchanges with processes further away.

Striped Synchronization. To avoid unnecessary wait times, we use striped synchronization: we number all processes sequentially, then processes with an odd number start by synchronizing with their previous neighbor and those with an even number start by synchronizing with their next neighbor.

2-Step Synchronization. For Jacobi 2D, we can cut the area as a grid. With the deep halo approach, each process would need to exchange data with up to 8 neighbors.

Due to the grid structure, data between processes in diagonal is small and also passed to their two common neighbors. An alternative is a two-step synchronization: first vertically only, then horizontally also passing corners' data whenever relevant. This introduces an additional wait, but reduces the number of communication channels.

Stacked processes. Instead of disposing processes' areas in a grid, we simply divide the space over one axis, e.g. vertically. This reduces the number of neighbor for each process but the total amount of data exchanged increases.

5.2.2. Experimental results

Problem sizes were scaled using the number of cores used (denoted S) as in [2]. Unless for varying the number of node, each experiment was run as a single batch. All experiments use striped async data exchange. Striped vs. non-striped did not make a significant difference.

Jacobi 1D. In Fig. 11, we examine the effect of deep halo on the computation time. Looking especially at the depths of 1 and 8, we see that the runtime is halved when using 48 cores. For 8 cores or more, we see that using a depth between 4 and 32 produces particularly stable results. In Fig. 12, we compare the runtimes for different number of nodes, to check if any adversary effect occurs when spanning the cores over more nodes. To the contrary, spanning over more nodes seems to decrease runtimes. While this may be caused by under-utilization of the specific processes in the cluster, it demonstrates that the proposed implementation scales properly with nodes.

Jacobi 2D. Since the array has size N^2, we scale the problem to $N = 1000\sqrt{S}$. In Fig. 13, we compare the different implementation: diagonal sync, 2-step sync, and stacked processes for different halo depths. The stacked processes implementation scales better than the other two, despite exchanging more data between the processes. It is however greatly affected by the halo depth. The 2-step and base (diagonal sync) implementation seem to be roughly on par. The effects of varying halo depth are weaker. In Fig. 14, we focus on the latter to determine the best halo depth. Between 4 and 8 seems optimal in our case.

6. CONCLUSIONS

We managed to provide manual optimizations that improves well over the base sequential implementations. This was done using different techniques.

Doitgen. We observed scalable results for both OpenMP and MPI using different techniques. We found that the combination of vector instructions, blocking and inverting the loops for matrix-matrix multiplication yielded the best results. For MPI we found that using individual write calls would yield good results over collective calls. We also verified that doitgen could easily span multiple nodes. One possible improvement would be to apply all the OpenMP techniques to the MPI version.

Jacobi. Despite the challenge of the algorithm, we were able to parallelize it using both OpenMP and MPI. For OpenMP, the most efficient was combining a pointer swap with a classic parallelization of the computation loop. Exploring overlapped tiling might provide better improvements. For MPI, using deep halo techniques provided to most important improvement. We believe that it would be possible to obtain even better results by reorganizing the memory accesses s.t. the local part that is being worked on fits in cache. The 2-step synchronization performed less well in our test. Another way to reduce data exchange overhead would be to let synchronizations happen at the same time as we compute other parts of the matrices, which however greatly improves code complexity.

7. REFERENCES

[1] Tomofumi Yuki Louis-Noel Pouchet, "Polybench," https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/, 2011-2016.

[2] Alexandros Nikolaos Ziogas et al., "Productivity, portability, performance: Data-centric python," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, New York, NY, USA, 2021, SC '21, ACM.

[3] Salvatore Di Girolamo Josh Milthorpe, "Libscibench," https://spcl.inf.ethz.ch/Research/Performance/LibLSB/, 2011-2016.

[4] Fredrik Berg Kjolstad and Marc Snir, "Ghost cell pattern," in *Proceedings of the 2010 Workshop on Parallel Programming Patterns*, New York, NY, USA, 2010, ParaPLoP '10, ACM.

[5] Wesley Bland Wes Kendall, Dwaraka Nath, "Mpi tutorials," https://mpitutorial.com/tutorials/, 2022.

[6] Torsten Hoefler Pavan Balaji, "Mpi for dummies," https://htor.inf.ethz.ch/teaching/mpi_tutorials/ppopp13/2013-02-24-ppopp-mpi-basic.pdf, 2013.

[7] Philippe Wautelet, "Best practices for parallel io and mpi-io hints," http://www.idris.fr/media/docs/docu/idris/idris_patc_hints_proj.pdf, 2015.

[8] Stephen Chong, "Cache performance," https://cs61.seas.harvard.edu/wiki/images/0/0f/Lec14-Cache_measurement.pdf, 2011.