POLYBENCH PARALLELIZATION OF GEMVER AND TRISOLV

Haoanqin Gao, Ilir Gashi, Johannes Gasser, Jonathan Fornera, Tom Wartmann

Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

In this work, we address the efficiency of solving large-scale computing problems by parallelizing and optimizing two benchmarks, *Gemver* and *Trisolv*, from the Polybench collection. We employ parallelization using Message Passing Interface (MPI) and OpenMP, aiming to reduce cache misses and floating-point operations. Performance evaluations were conducted on the ETH cluster *Euler*, with a varying number of input sizes, processors, and threads. Our OpenMP implementation for Gemver performs similarly to OpenBLAS when utilizing higher thread numbers, whereas our MPI implementation for Trisolv surpasses the performance of the OpenBLAS routine by nearly 1.5 times and achieves a speedup of 7x over the baseline.

1. INTRODUCTION

Solving computing problems quickly and efficiently has always been important. However, as problem sizes continue to expand, reaching a scale that is impractical for efficient resolution on a single machine, the significance of parallel computing becomes even more pronounced. To facilitate the comparison of system performance, collections of benchmark programs with clear definitions exist. One such collection is Polybench [1], which involves over 30 numerical computations written in C. In this work, we focused on parallelizing and optimizing two specific benchmarks from Polybench, namely Gemver and Trisolv. The Gemver algorithm is a benchmark for vector multiplication and matrix addition and Trisolv is a solver for triangular matrices using forward substitution. Although these problems are simple, their optimization is crucial, as they often serve as integral components within larger and more complex programs. We have parallelized the code with MPI [2] and OpenMP [3], along with optimizations to reduce cache misses and the number of floating point operations (FLOPS). Moreover, we tested and compared the performance of the optimizations on the high-performance cluster "Euler" of ETH Zürich with different numbers of processors, nodes, threads, and input sizes. Finally, we compared our results to a state-of-the-art open-source library for basic linear algebra subroutines (BLAS).

Related work. Gemver is also a well-known routine of the updated BLAS [4] and has already been optimized in other works [5] with an automated code generation for distributed machines using MPI. However, for this benchmark, despite the parallelization across several processors, they could not achieve any speedup compared to sequential execution. The public library OpenBLAS [6] [7] has extensively optimized the BLAS routines, including Trisolv and subroutines of Gemver, and achieved significant speedups compared to the baseline. We sequentially combined the functions *dger*, *daxpy*, and *dgemv* from OpenBLAS to emulate the behavior of Gemver, so we can directly compare the performance of OpenBLAS with our proposed methods.

2. BACKGROUND

In the following, we introduce the mathematical problems covered by the two Polybench benchmarks we selected, Gemver and Trisolv. As mentioned before, these benchmarks are the simplest possible implementations of two problems in the field of linear algebra.

Gemver. Gemver gets as inputs the vectors of length N: $\mathbf{u}_1, \mathbf{u}_2, \mathbf{v}_1, \mathbf{v}_2, \mathbf{y}, \mathbf{z}$, matrix $\mathbf{A}$ of size $N \times N$ and scalars α and β and computes the following outputs:

$$\hat{\mathbf{A}} = \mathbf{A} + \mathbf{u}_1 \otimes \mathbf{v}_1 + \mathbf{u}_2 \otimes \mathbf{v}_2 \tag{1}$$

$$\mathbf{x} = \beta \hat{\mathbf{A}}^T \mathbf{y} + \mathbf{z} \tag{2}$$

$$\mathbf{w} = \alpha \hat{\mathbf{A}} \mathbf{x} \tag{3}$$

Where matrix $\hat{\mathbf{A}}$ has dimensions $N \times N$ and the vectors $\mathbf{x}$ and $\mathbf{w}$ are of length N. It is worth noting that the computation (2) depends on (1), whereas (3) depends on (1) and (2). The baseline implementation requires $10N^2 + N$ FLOPS and $3N^2 + 7N$ memory access operations.

Trisolv. The Trisolv benchmark solves the linear system of equations (LSE) $Ax = b$ for the vector x, where A is a lower triangular matrix of dimension $N \times N$ and x and b are vectors of size N. An entry of vector x is calculated in the

following way:

$$x_1 = \frac{b_1}{a_{1,1}} \tag{4}$$

$$x_i = \frac{b_i - \sum_{j=1}^{i-1} a_{i,j} \cdot x_j}{A_{i,i}} \quad i > 1 \tag{5}$$

In terms of parallelization, one can see that the computation for x_i has dependencies on all the computations of x_j, where $j < i$. That means a full parallelization of the algorithm is not possible and has to be implemented with these dependencies in mind.

3. METHOD

In this section, we shortly explain how we structured and worked with our code, and then go into more detail about what we implemented to achieve our goal of better performance.

Data initialization and measurements. To measure the execution time of our functions we used the C function `clock_gettime()` with identifier `CLOCK_MONOTONIC`. For better comparisons with the original Polybench functions, we followed their approach and only timed the actual computations, and not the data initialization. When it comes to MPI functions, we initialized all the necessary data separately on all different processes. We stop timing a function as soon as all the values have been computed and are present on process 0.

Implementations. For both problems, we have multiple implementations that use MPI, OpenMP, or both. We tested all implementations for correctness against the original Polybench benchmarks for many different input sizes and values, including randomly generated ones.

Gemver, OpenMP. The OpenMP implementation of Gemver focuses on exploiting parallelism for shared memory devices. By employing `pragma omp parallel for` directives with collapse for the computation of the matrix $\hat{\mathbf{A}}$, and reductions for the vectors $\mathbf{x}$, and $\mathbf{w}$, we optimized the loops for concurrent execution.

Gemver, MPI. To parallelize Gemver over multiple processes distributed over different nodes we used MPI. Since the communication between nodes is time-consuming we aimed to reduce the overall number of messages passing to the absolute minimum. To realize this objective, we performed computations for all three dependent components of Gemver on a single process, ensuring the absence of communication intermissions during the computation phase. Instead, our approach involved solely reducing and gathering results from various processes at the end of the computational process. To this end, we initialized k columns of matrix $\mathbf{A}$ on each node, where k is defined as the number of elements (N) divided by the number of processes (p). Additionally, we initialized the first k elements of vectors $\mathbf{v1}$, $\mathbf{v2}$ and $\mathbf{z}$ along with the complete vectors $\mathbf{u1}$, $\mathbf{u2}$ and $\mathbf{y}$ on every node. This leads to the following computations on process i where $i \in \{0, 1, \ldots, p-1\}$ and j is the slice of a vector or matrix from index $i \cdot k$ to $(i+1) \cdot k$:

$$\hat{\mathbf{A}}_{:,\mathbf{j}} = \mathbf{A}_{:,\mathbf{j}} + \mathbf{u}_{1_:} \otimes \mathbf{v}_{1_j} + \mathbf{u}_{2_:} \otimes \mathbf{v}_{2_j} \tag{6}$$

$$\mathbf{x_j} = \beta \hat{\mathbf{A}}^T_{j,:} \mathbf{y}_: + \mathbf{z_j} \tag{7}$$

$$\mathbf{w}'_: = \alpha \hat{\mathbf{A}}_{:,\mathbf{j}} \mathbf{x_j} \tag{8}$$

The final results are the concatenation of all p $\hat{\mathbf{A}}_{:,\mathbf{j}}$ and $\mathbf{x_j}$ and the reduction with addition using all $\mathbf{w}'_:$ onto the final $\mathbf{w}$ into the main process. This results in a minimal number of three MPI communications, albeit at the expense of a higher number of data initialization and memory accesses. This implementation increases the number of memory accesses compared to the baseline to $3N^2 + 4N + 3N \cdot p$ while maintaining the number of FLOPS, considering no further optimizations.

Gemver, Further optimizations. By altering the sequence of scalar multiplication, specifically transitioning, e.g., from $(\alpha\mathbf{A})\mathbf{x}$ to $\alpha(\mathbf{A}\mathbf{x})$, we effectively decreased the total number of FLOPs to $8N^2 + 3N$. Further, we unrolled the loops for matrix multiplication by a factor of 4 to reduce the cache misses.

Trisolv. To parallelize Trisolv, one has to take the dependencies induced by the calculation of vector x into account. We propose two methods to parallelize this workload. The first method takes the straightforward approach of parallelizing the sum when calculating an x_i as can be seen in equation 5. The second approach distributes already computed values x_i to all other processes so that they can concurrently compute the sum of equation 5.

Trisolv, OpenMP. For the OpenMP approach, we took the already existing Trisolv implementation of Polybench and tried to apply OpenMP pragmas to it. Since the Polybench algorithm calculates x entry after entry, we decided to do the same and apply different OpenMP pragmas to the inner for loop, which corresponds to the sum as seen in equation 5.

Since the amount of computations is proportional to i, there will be an overhead for parallelizing the sum for small i without any real gain, since there are very few computations to be performed. However, the bigger the matrix size N, the less impact this initial overhead will have, as there is more to gain from parallelization.

Trisolv, MPI. Using the straightforward approach of parallelizing the summation in equation (5) with MPI requires a lot of costly data movement between processes. In the most naive implementation with p processes and N elements of the vector $\mathbf{x}$, there would be pN sent to the root process. These communications all need to be synchronized and would negatively affect any performance gained through the parallelization. In fact, in some of our earlier

tests, we measured that calls to communication functions such as MPI_Bcast would take up to 80% of the execution time.

What we instead opted to do is to equally divide the rows of **A**, **x**, and **b** among all processes, so that each one is only responsible for a fraction of the final values. When a process has computed a value of **x**, it sends it to all other MPI processes so that they can already subtract it from the temporary value of all the entries of **x** that they are responsible for. We were then able to reduce the number of communications by packaging and sending multiple computed entries of **x** at a time. Through further measurements we found 8 double floating point variables to be the optimal amount to balance communications and the other processes sitting idle.

The last important choice is the type of MPI communication used to send values from one process to another. We implemented similar versions of the Trisolv benchmark with two different two-sided communication methods. The first one is broadcast, which sends data from one root process to all other processes inside a communicator. The second one is the non-blocking send: the root process needs to issue a separate isend for each process that it sends to, and they all need to be matched by a corresponding receive statement.

We also experimented with column major and row major storage orders for matrix **A** in our implementations. For the version using MPI_Bcast, column-major order is used and during the subtraction phase, we access the matrix column by column. For the other MPI implementations, we use row-major order.

Trisolv Hybrid. The hybrid implementation of Trisolv, that is, using both OpenMP and MPI, is based on the non-blocking send MPI implementation. The only difference is the usage of OpenMP to speed up the subtraction of values received from another process. If, e.g., a process is responsible for 16 rows of the vector **x** and it can use 4 OpenMP threads, each thread will do the subtraction on a block of 4 rows.

4. EXPERIMENTAL RESULTS

In this section, we present the performance of our parallelization methods, comparing them to both the baseline and the OpenBLAS implementation. Additionally, we demonstrate scaling across various input sizes, processes, and threads.

Experimental setup. All our experiments were run on ETH's Euler cluster, which runs multiple computation nodes, each containing two AMD EPYC 7742 processors with a nominal clock speed of 2.25 GHz and a peak clock speed of 3.4 GHz. With the Slurm Workload Manager, we were able to request a fixed amount of cores and memory to perform our experiments. In general, we matched the number of threads and tasks we use for a job with the number of cores the job gets. So when we perform an experiment using 2 tasks and 4 threads per task, we allocate 8 cores to this job.

The outcomes presented in this paper are generated through C++ code, compiled with the GCC compiler 11.4.0, using the c++14 standard and utilizing optimization flags *-O3* and *-ffast-math*.

4.1. Gemver

In this sub-section, we want to make statements about our most successful solutions for the Gemver problem using OpenMP, MPI, and a hybrid approach. Our most successful OpenMP implementation parallelized the first of the three loops with a `parallel for collapse(2)` pragma, as the loop parallelization is straightforward. The other loops were parallelized using the reduction to a local sum pragma.

We tried different approaches to optimize and parallelize Gemver with MPI, including many different approaches to distribute the input matrices, transpose the matrix by distributing the data column-wise to the other processors, and other optimizations. In this part, we present the best-performing version of MPI where the data is distributed and the code optimized as described in the method section 3 for *MPI* and *Further optimizations*. We also show that the hybrid version does not improve performance.

Gemver MPI Scaling. The MPI implementation of Gemver does not scale proportional to the number of processors, as we can see in Figure 1. Performance is low for fewer than 2 processors, peaks between 4 and 6 as well as between 18 and 24 processors, and shows again a slight increase with 48 processors.

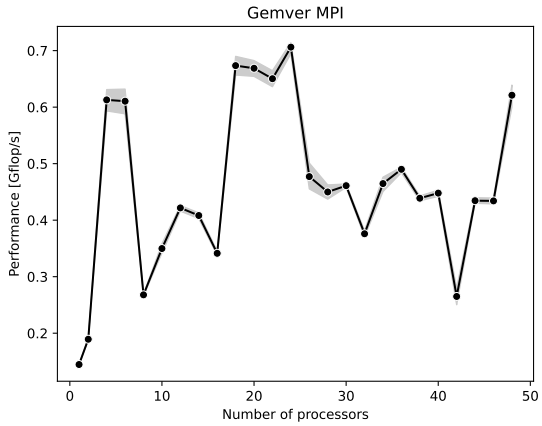

Fig. 1. Performance of **MPI** Gemver for different number of processors. Input size of $N = 2^{14}$. The confidence interval of 95% is plotted in gray.

Gemver OpenMP Scaling. In figure 2, the OpenMP version of Gemver shows a consistent performance for less than four threads, increases linearly between 4 and 32 threads, and levels off at a performance of 1.9 Gflops/s with a higher

number of threads. Nevertheless, the performance is far from perfect scaling, i.e. doubling the number of threads does not lead to a doubling of performance.

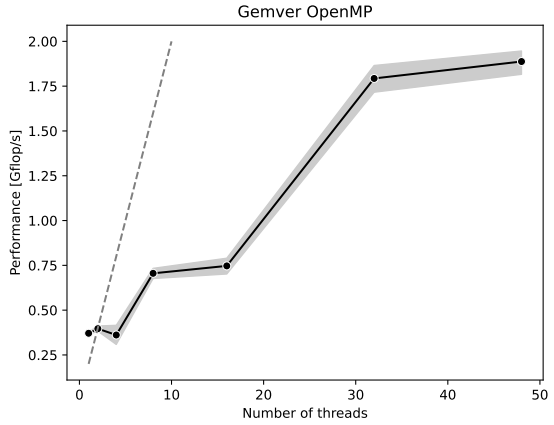

Fig. 2. Performance of **OpenMP** Gemver for different number of threads. Input size of $N = 2^{14}$. The confidence interval of 95% is plotted in gray and dashed lines represent linear scaling with 0.2 [Gflops/s/thread].

Gemver Speedups. We benchmarked our different approaches to the parallelization of Gemver against the baseline polybench implementation and our implementation using OpenBLAS routines. Our goal is to assess the performance of our solution relative to a sophisticated and popular implementation such as OpenBLAS. We tested all implementations with 32 cores. The OpenMP implementation as well as the OpenBLAS implementation both were allowed to run with 32 threads. The MPI implementation ran with 32 tasks with one core allocated to each task. The MPI OpenMP hybrid implementation ran with 4 tasks, 8 threads for each task, and performed slightly worse than the standalone MPI version.

The baseline for the speedup had the following runtimes:

N	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}
time[s]	0.0109	0.0650	0.290	1.18	4.08	15.0

Table 1. Absolute runtimes of the Gemver speedup experiment baseline.

As shown in figure 4.2, we built an OpenMP implementation that can compete with the OpenBLAS implementation. The MPI implementations both slowed down the computation.

4.2. Trisolv

In this sub-section, we want to make statements about our most successful solutions for the Trisolv problem using

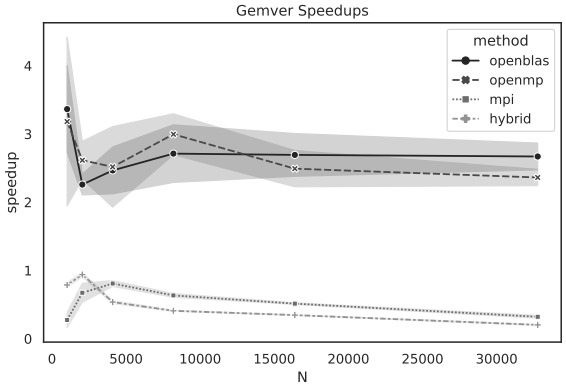

Fig. 3. Speedup of our different Gemver implementations with 32 Cores. The confidence interval of 95% is plotted in gray.

OpenMP, MPI, and a hybrid approach. For each of the three, we picked the most successful ones in terms of runtime and compared them to OpenBLAS.

Trisolv OpenMP Scaling. In this experiment, we want to figure out, how well our Trisolv OpenMP implementation scales using multiple cores. We run our most successful Trisolv implementation on Euler with differing amounts of cores and a fixed input size $N = 2^{14}$. We then compare the results, to make a statement about the scalability of this solution.

During the evaluation, we encountered issues with our OpenMP implementation of Trisolv on Euler. It did not scale as well as we had observed on a local machine. Additionally, we sometimes measured slowdowns on Euler, which we could not explain, of up to a factor of 1000, during which the CPU was fully utilized. These measurements amounted to roughly 5% of our measurements, increasing with more number of threads.

So in addition to measuring the scaling of our solution on Euler, we want to show this effect by comparing the Gflop/s when running this experiment on Euler to running it on one of our machines. The machine was equipped with an AMD Ryzen 7 PRO 6850U containing 8 physical or 16 virtual cores with a nominal clock speed of 2.7 GHz and a peak clock speed of 4.7 GHz. It used the same compiler setup we used on Euler to rule out potential optimization misses caused by different compiler versions.

As can be seen, we achieved a consistent scaling from 1 to 4 threads on the local machine, which stagnated at 8 threads and broke down at 16. On Euler in contrast, we got an improvement for two threads and a decrease in performance after that. We can conclude, that the OpenMP implementation best performs at 8 threads locally and did not scale with more than 2 threads on Euler.

Trisolv MPI Scaling.

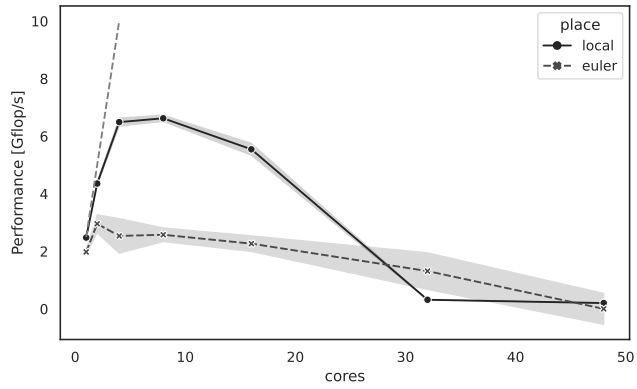

Fig. 4. Comparison of GFlop/s for different number of threads at an input size $N = 2^{14}$. The dashed line shows the potential linear speedup for the local version. The confidence interval of 95% is plotted in gray.

While investigating the scaling of our Trisolv MPI Bcast implementation, we kept noticing irregularities, an example of which can be seen below:

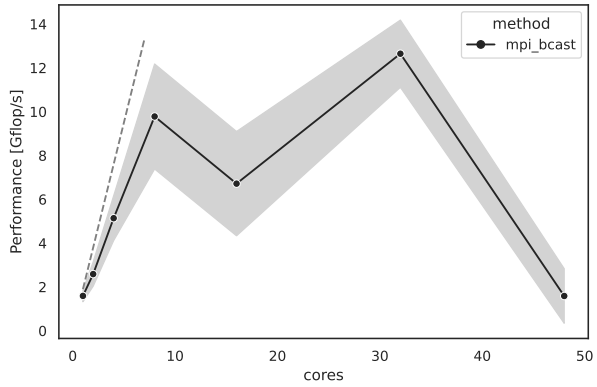

Fig. 5. Comparison of GFlop/s for different number of tasks at an input size $N = 2^{14}$. The dashed line shows the potential linear speedup. The confidence interval of 95% is plotted in gray.

One would expect the performance to at least monotonically increase, and sometimes this is the case, with the performance scaling almost perfectly from 1 to 8 cores, but we cannot reproduce such a result reliably. In the extreme case such as at 48 cores in figure 5, the performance is even worse than running the program on a single core.

Trisolv Speedups. Again, we benchmarked our different approaches against the baseline polybench implementation of Trisolv and the OpenBLAS method `cblas_dtrsv`. Our goal is to make a statement about how well our solutions perform compared to a sophisticated and popular implementation such as OpenBLAS. As with Gemver, we tested all implementations using 32 cores. The OpenMP implementation as well as the OpenBLAS implementation both were allowed to run with 32 threads. The MPI implementation ran with 32 tasks with one core allocated to each task. The MPI OpenMP hybrid implementation ran with 4 tasks and 8 OpenMP for each task.

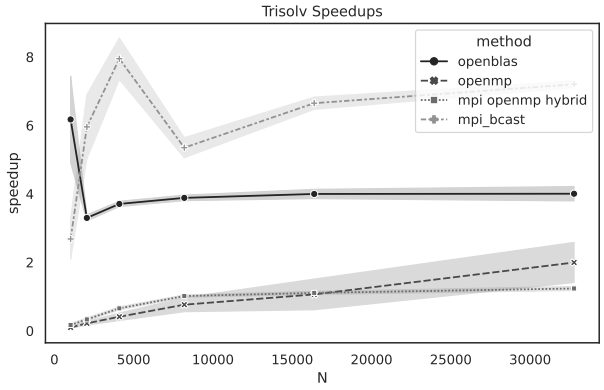

Fig. 6. Achieved speedup of our different Trisolv implementations with 32 cores of AMD EPYC 7742 on the Euler VII cluster

The baseline for the Trisolv speedup experiment had the following runtimes:

N	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}
Time[ms]	0.725	3.18	12.3	48.6	193	773

Table 2. Runtimes of the Trisolv speedup experiment baseline

The OpenMP implementation performed badly in this experiment because of the measurement problems we mentioned earlier and because it performs poorly on small input sizes. Using `perf` we profiled the OpenMP solution against the OpenBLAS method. Both solutions used roughly the same amount of instructions. The amount of cache misses was slightly worse for the OpenMP solution for small inputs (18% for OpenBLAS vs 24% for OpenMP). Lastly, the OpenMP implementation branched around 10 times more than the OpenBLAS implementation, which could explain the slower runtime. Unfortunately, time did not allow us to further investigate and optimize our solution for better performance at small inputs.

Our broadcast MPI implementation outperformed OpenBLAS on all input sizes except 2^{10} by almost a factor of 2. We conclude that for scaling with many threads, MPI seems to be the right approach for Trisolv, as OpenBLAS as well as the OpenMP was limited to a 4 times speedup at best,

while the MPI implementation reached speedups of a factor between 6 and 8.

For the sake of clarity, the other pure MPI implementations using different MPI communication calls that we have experimented with are not shown in Figure 5, given that the Bcast version is the fastest out of them all. To explain our Bcast implementation's performance, we logged the cache misses for each of our implementations:

Implementation	Cache misses
Baseline	0.481M
Isend (row-major)	33.9M
Onesided (row-major)	34.6M
Bcast (column-major)	2.32M

Table 3. Cache misses of the different Trisolv MPI implementations (in million) for a matrix of size $N = 2^{14}$

We think it is not that MPI_Bcast is superior to the other MPI functions for solving a triangular matrix, instead, it is due to the choice of column-major storage order for the matrix and also how well the code is optimized, leading to an order of magnitude decrease in cache misses.

5. CONCLUSIONS

In this report, we showed different approaches to the parallelization of the two polybench benchmarks Gemver and Trisolv. We analyzed the scalability of our most promising solutions over multiple cores and compared them to the widely used OpenBLAS library on different input sizes.

For Gemver, our OpenMP implementation competes with our OpenBLAS-based version at 32 and 48 threads, yet is inferior performance with fewer threads. With nonlinear scalability, it achieves a 2.5x speedup over the baseline at 32 threads. We also confirmed like other authors before us, that the MPI approach did not perform exceptionally well for Gemver, mainly because of the overhead created by MPI communications in combination with the compiler's effective optimization of the relatively straightforward calculations. It is also interesting that the MPI communication is particularly slow for 2^x numbers of processors like 2, 8, 16, and 32. However, our MPI implementation would be essential as soon as the memory capacity of a node, in our case 512GB i.e. input size of about 150000, would be exceeded. This would be an interesting topic to work on further but would require increased access to Euler.

One factor that we can not control that leads to inconsistent measurement, especially for MPI and hybrid versions, is the fact that the program is run on a different physical Euler node every time we change the number of cores. However, accurately identifying the root cause is challenging without additional information and full control of the underlying hardware.

For Trisolv, our OpenMP approach performed badly on the Euler environment, causing slowdowns in most cases. However, the MPI approach we created, outperformed the OpenBLAS routine for Trisolv by almost a factor of two. Although this is only in the specific situation of running on one Euler node, the performance of our fastest approach drops below that of the OpenBLAS routine once multiple Euler nodes are involved and the communication time between the MPI processes increases. One possible improvement worth looking into would be increasing the block size further, thus reducing the number of MPI communication calls.

6. FURTHER NOTES

Our source code as well as the experiment results can be found on our GitLab Repository[8].

7. REFERENCES

[1] "L.-n. pouchet. polybench/c 3.2," https://www.cs.colostate.edu/ pouchet/software/polybench/, Accessed: 2023-10-23.

[2] Marc Snir, *MPI–the Complete Reference: the MPI core*, vol. 1, MIT press, 1998.

[3] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[4] "An updated set of basic linear algebra subprograms (blas)," *ACM Trans. Math. Softw.*, vol. 28, no. 2, pp. 135–151, jun 2002.

[5] Nelson Lossing, Corinne Ancourt, and Francois Irigoin, "Automatic code generation of distributed parallel tasks," in *2016 IEEE Intl Conference on Computational Science and Engineering (CSE) and IEEE Intl Conference on Embedded and Ubiquitous Computing (EUC) and 15th Intl Symposium on Distributed Computing and Applications for Business Engineering (DCABES)*, 2016, pp. 234–241.

[6] Qian Wang, Xianyi Zhang, Yunquan Zhang, and Qing Yi, "Augem: Automatically generate high performance dense linear algebra kernels on x86 cpus," 11 2013.

[7] Zhang Xianyi, Wang Qian, and Zhang Yunquan, "Model-driven level 3 blas performance optimization on loongson 3a processor," in *2012 IEEE 18th International Conference on Parallel and Distributed Systems*, 2012, pp. 684–691.

[8] "Gitlab repository," https://gitlab.ethz.ch/jgasser/dphpc2023/, Accessed: 2024-01-12.