Performance Comparison Between OpenMP and MPICH

Gard Åckerström Aasness Maximilian Georg Kurzawski
gardaa | mgku @kth.se

October 25, 2023

Abstract

In the realm of distributed computing, where software execution spans multiple computers, message passing is used to handle the communication between the different computers or even within the different cores of one CPU. For that reason, a standard was founded called the Message Passing Interface (MPI). Over the years different actors within the field of computer science have thought out different implementations of the MPI standard. These different MPI libraries naturally provide different advantages and disadvantages. This study delves into the performance differences of two prominent MPI implementations, OpenMP and MPICH, within specific software environments. The research aims to address whether there is a significant difference between the two MPI libraries in our specific environment, and whether or not this result applies to most or all environments or if individual optimization is necessary. By conducting these quantitative tests on distinct hardware and operating systems, we provide insights into the choice of MPI libraries and their implications. More specifically, we will look into how important the choice of MPI can be, and whether the importance could warrant a further testing stage during the development of a product. Our approach focuses on the comparative performance assessment of OpenMP and MPICH, offering a nuanced understanding of their effectiveness in specific software environments. As the results indicate, there is a significant difference in performance between the different MPI libraries when using 4 or more cores.

Contents

List of Acronyms and Abbreviations

- MPI - Message Passing Interface

1 Introduction

In a distributed system, where programs are executing and memory is allocated across several computers, MPI provides an interface for passing messages. OpenMP and MPICH are two widely used MPI implementations, which we will perform a performance analysis on and compare the results. [1]

While different MPI implementations have undergone testing across diverse hardware configurations, our comparison will focus on testing the different libraries. Given the significant impact of the environment on performance, these comparisons will yield valuable insights into the library performance within specific environments, enhancing their relevance. [2] This will allow us to draw some conclusions on whether or not the choice of the library makes a substantial difference in performance. We will however also discuss potential problems with the way we approach this test and the applicability of our results on different test setups.

The objective of conducting a performance analysis is to assess the influence of different MPI libraries, with the aim of establishing practical recommendations for their appropriate usage in specific scenarios, if such guidelines exist. To discover these guidelines and reach our goal, we have formulated the following three research questions that we aim to answer after gathering and analyzing our results:

1. Is there a significant performance difference between the two MPI Libraries in our specific environment?

2. Does a potential performance difference warrant a further testing stage during product development?

3. Are the results found on our test setup applicable to other machines or should every target machine be tested individually to find the optimal tool-chain?

Before performing the performance analysis, we expect the results to indicate some level of performance difference, as the different MPI libraries react differently to various factors such as processing power and how many cores they use. However, we also expect this performance difference to vary depending on the environment, the CPU, and the general machine setup. In addition, we expect an indication that it would probably yield some benefit for developers to test different MPI implementations on their machines to find the optimal setup for their environment.

2 Theoretical Framework

As previously mentioned, performance is dependent on the environment it is running in. Therefore, no tests comparing the same two MPI libraries in the same environment have been conducted in the past. However, there have been various performance tests on MPI libraries done in the past.

To begin with, a study performing micro-benchmarks and applications claims that the choice of the high-speed interconnected hardware, InfiniBand, Myrinet, and Quadrics, is important as they all have advantages and disadvantages and there is a significant difference in favor of InfiniBand for several applications.[2] Therefore, one can assume that the choice of hardware is important to the performance of the MPI libraries.

Another study comparing some of the programming approaches available with OpenMP shows performance differences. It states that the 'naive' loop level approach is not competitive, the 'improved' loop level approach has limited performance, and the 'optimized' loop level approach gives good, yet unstable performance. [3] That shows that not only hardware and the choice of MPI library is a deciding factor, but also the programming approach within the actual MPI implementation.

A very similar study agrees that the choice of programming style and approach has a deciding factor regarding performance reaching the same results. It concludes that certain implementations of OpenMP lead to better performance than MPI, and states that the performance difference is significant enough that in some cases the higher programming effort needed to implement the advanced OpenMP implementation is worth it in terms of the gain in performance. It states that the reasons for good performance turn out to be good communication, and bad performance is bad usage of memory.[4]

A study performing a comparison between two MPI implementations, MPI/Pro and MPICH, comparing performance by testing message passing strategies such as point-to-point and all-to-all among other things, shows that MPI/Pro has a consistently better performance than MPICH. How much better it performed varied depending on which operation or scenario it was testing.[5]

3 Methodology

In this section, we will provide all relevant information necessary to conduct the experiment, such as the testbed, the testing implementation, and which variables we base the comparison on. If you want to take a look at the code implementation, the results or want to recreate the results in your environment, take a look at the github repo. The research is a quantitative study, and we will use statistical methods to analyze the data and determine whether or not there is a significant difference. As there will be no participants included in the research, there will be no ethical considerations, concerning the data of individuals, relevant to this study.

3.1 Variables

The independent variables in the research consist of both hardware and software. As shown in table 1 in the testbed subsection, we will use three different machines with different OS, RAM, and cores, which will influence the results, and can therefore be considered independent variables. As for the software, the different MPI libraries will define the final results of the performance, as we want to figure out how they compare against each other. Lastly, the number of MPI processes run (N) and optimization levels (O) of the compiler flags are two variables that will also influence the results. Therefore, we have included a total of 6 different independent variables in our research.

The dependent variable in the research is the actual performance of the different MPI libraries. This will be measured in time (s), and it is the metric we gather and analyze to be able to answer our research questions, and it will vary based on the independent variables mentioned above.

3.2 Testbed

As previously mentioned, the tests of the two different MPI implementations will be run on three different machines. The main reason to conduct the tests on multiple machines is to look for patterns that can apply to all machines, or if the fastest MPI library differs from machine to machine. Results from three machines are not enough to deem a library to be faster than the other on all other existing machines, but it will give an indication that can be researched further in future studies. It will also give us an indication to research question 3.

Table 1 shows the relevant hardware and software for each machine being used to conduct the tests. Machines with different OS have intentionally been chosen, to gain some insight into the possibility of the performance optimal MPI library varies depending on OS. There was no specific reason the machines were chosen based on the RAM and Core, but rather a choice made from having a limited budget.

	RAM	Core	OS
Machine 1	64 GB of DDR4-3000 RAM	AMD Ryzen 7 3700X 8-Core	Windows (WSL)
Machine 2	64GB RAM DDR4 ECC	Intel Xeon E3-1275v5	Linux
Machine 3	16GB of 128-bit LPDDR4X SDRAM	Apple M1 Pro 14-Core	MacOS

Table 1: Relevant hardware of the three different machines that is used to gather performance data

For the library versions we used the following for the machines:

- Machine 1: mpicxx: Open MPI 4.0.3 (Language: C++), MPICH Version: 3.3.2

- Machine 2: mpicxx: Open MPI 4.1.2 (Language: C++), MPICH Version: 4.0
- Machine 2: mpicxx: Open MPI 4.1.6 (Language: C++), MPICH Version: 4.1.2

3.3 Data

As we will conduct a quantitative performance analysis, we will gather the data ourselves and create a dataset for comparison of the MPI libraries. The data will be gathered by executing the tasks, found in the GitHub repository, 10 times on each computer. After executing the tasks 10 times, we will calculate the average, which is the value we will use for the comparison. Then we will repeat these tests with N processes (with N = 1, 2, 4, 6 & 8). Finally, we will repeat all these steps with different compiler flags O (with O = 0, 1, 2, & 3). We are hoping that these measures exclude different hardware and variance contributing to bias.

3.4 C++ Function

The function to be parallelized can be found in the appendix B. The function aims to take the input array and loop over the individual elements before calculating the following equation:

$\text{sum} = \sum_i \sqrt{\sqrt{\text{data}[i]} + \sqrt{\text{data}[i]}}$

To parallelize it, one can use the *MPI_Scatter* and the *MPI_Reduce* functions. The *MPI_Scatter* function takes 1/N, with N being the number of MPI processes, and hands it over to one of the processes. Then the local sum will be computed based on the individual input and finally, we will reduce it all down to a single sum. The result of this can be found in appendix C.

3.5 Data Analysis

The metrics we will base our performance comparison on are:

- the time (s) it takes for each MPI library to perform the same task on the same machines
- the number (N) of MPI processes
- the optimization (O) levels of the compiler flags

To gain a necessary and initial overview of the results, we will calculate the average. The average value will help us understand the time the implementations usually cluster, and what time you can expect the implementation to use for this exact test. By doing it this way, we can hopefully reduce variance in the performance from impacting the results too much.

We will utilize Python to create some graphs in order to visualize the results, making it more understandable and easier to interpret the results.

After performing the initial calculations, we will perform paired T-tests on each set of data to check whether or not there is a significant difference in the performance between the two MPI libraries on each machine. More specifically, the data we will use for the T-tests are the performance of the MPI libraries with the same amount of processes being run across the different optimization levels. Seeing how we are gathering data for five (three for Machine 2) different numbers of processes, there will be five T-tests per machine. We will do paired T-tests because we are testing the two MPI implementations under the same conditions by having the same script being run. Therefore, we have paired data points that we wish to compare. In addition, a paired T-test only requires the data to be normally distributed, whereas a two-sample T-test also requires it to have the same variance, which it will not necessarily have.[6] We will use the *stats.ttest_rel()* function in the *SciPy* Python library to perform the T-tests. After running the paired T-tests, it will give us the calculated t-statistic and p-value. The t-statistic tells us the difference between the means of the two libraries and gives us an indication of whether or not a difference is large enough that it is unlikely that it has happened by chance. The p-value tells us whether or not the observed data is consistent or inconsistent with the null hypothesis, which refers to the hypothesis that there is no significant difference between the observed data. Seeing how we do not have too much data to rely on, we have chosen a significance level of 0.05, which equals 5%, meaning that if the p-value provided by a T-test is lower than 0.05, there is a significant difference between the performance with that specific amount of processes.

3.6 Motivation of Choice

We have chosen the methodology based on both the optimal choices for a quantitative study comparing performance between two MPI implementations and what we believe will help answer the research questions.

To begin with, research question 1 will be answered by performing a paired T-test. That way, there will be a concrete answer if there is a significant difference or not, instead of an interpretation of the results.

Secondly, research question 2 is a bit more up for interpretation as the answer is not as straightforward as research question 1. However, no matter what the results show, we believe it will give us an indication to answer the research question. If the results vary quite a bit or there is a significant difference, it might be beneficial for developers to do research about which MPI implementation to use when developing a product, but if they do not vary, it might not be as important.

Lastly, research question 3, while being left up for interpretation, since we will conduct the tests on three different machines with different hardware and software, we will get an indication of an answer that potentially can be investigated further in the future at a larger scale.

4 Results and Analysis

In this section, we will present the results we got after running the tests we described in the data and data analysis sub-sections of the previous section. For each machine, we will visualize the results of the tests by using a plot, followed by presenting the results of the paired T-tests performed as explained in the previous section.

There are a couple of things to be aware of when looking at the results presented below. To begin with, Machine 2 only has 4 cores. This impacted the results for Machine 2 a lot, as we could only gather data for 1, 2, and 4 cores, therefore not being able to compare the performance using 6 and 8 cores. It should however still give a good indication of results.

Secondly, it is important to be aware of the time values on the Y-axis of all the graphs. Otherwise, the graphs can be misleading. Even though the bars in each graph seemingly have the same height, which can be interpreted as similar values, the values differ a lot because of the big gap between the values on the Y-axis.

Furthermore, on all machines, the Y-axis is the dependent variable seeing how it refers to the performance time in seconds. The X-axis and the title of each plot are independent variables, as they refer to the number of processes and optimization level respectively.

Lastly, as mentioned earlier, if you want to take a look at the absolute values take a look at the github repo.

4.1 Machine 1

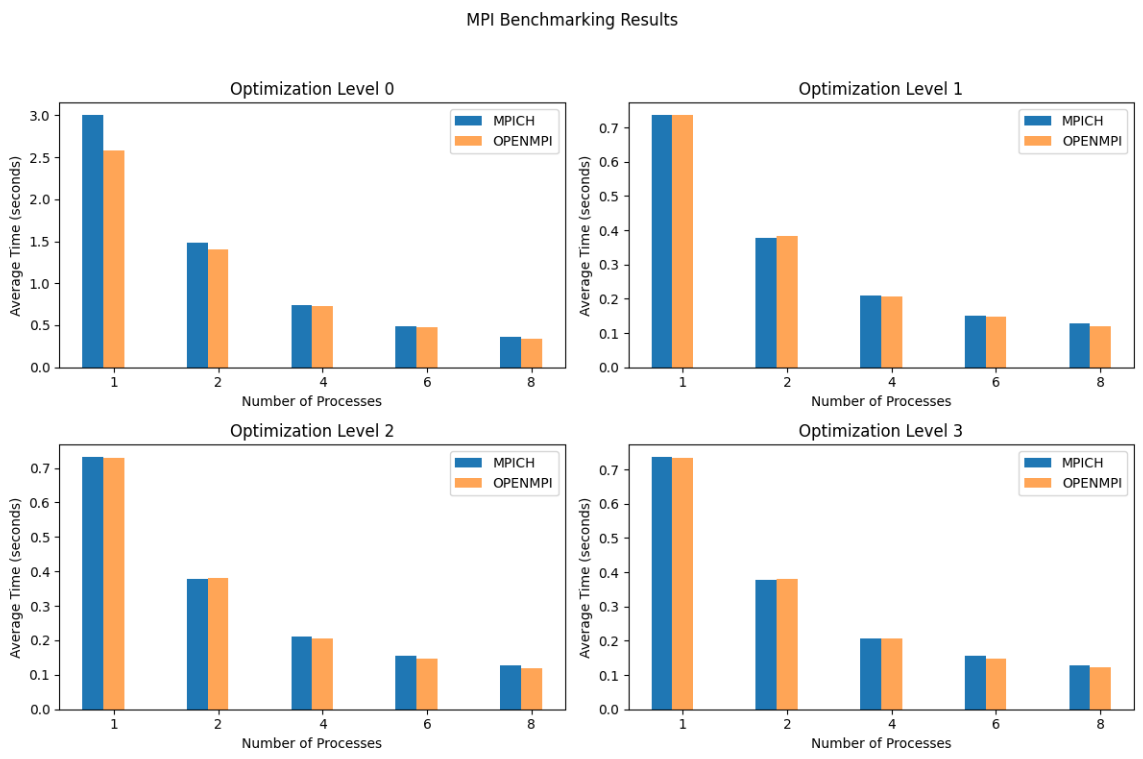


Figure 1: Time results for Machine 1

Processes	t-statistic	p-value	significant difference
1	-1.0085743043796587	0.38747195064163786	No
2	-0.8585038225135933	0.45373581449811984	No
4	-3.680069143324941	0.0347555672917576	Yes
6	-5.769167498293327	0.010352463524861055	Yes
8	-3.871078995917003	0.030505209599973732	Yes

Table 2: T-statitics and p-values provided by the T-tests for Machine 1

4.2 Machine 2

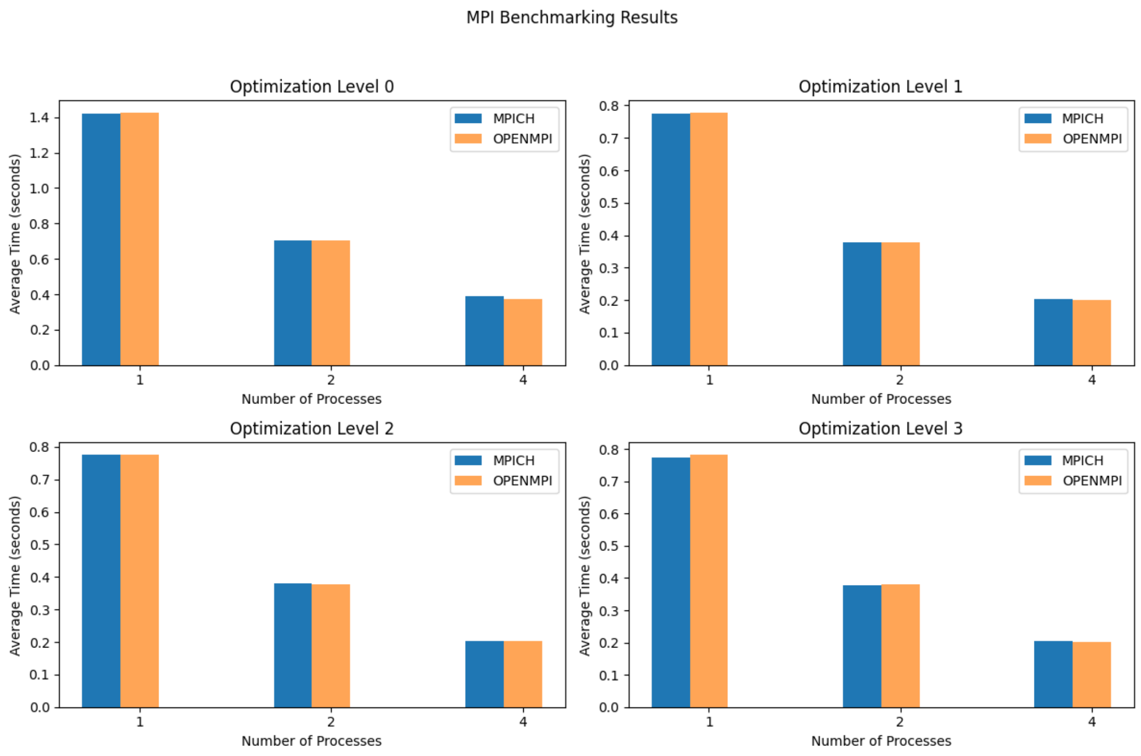


Figure 2: Time results for Machine 2

Processes	t-statistic	p-value	significant difference
1	1.307163304739126	0.2823214808142253	No
2	-0.5569363600035606	0.6164220030005504	No
4	-1.511573121929868	0.22782421546483564	No

Table 3: T-statitics and p-values provided by the T-tests for Machine 2

4.3 Machine 3

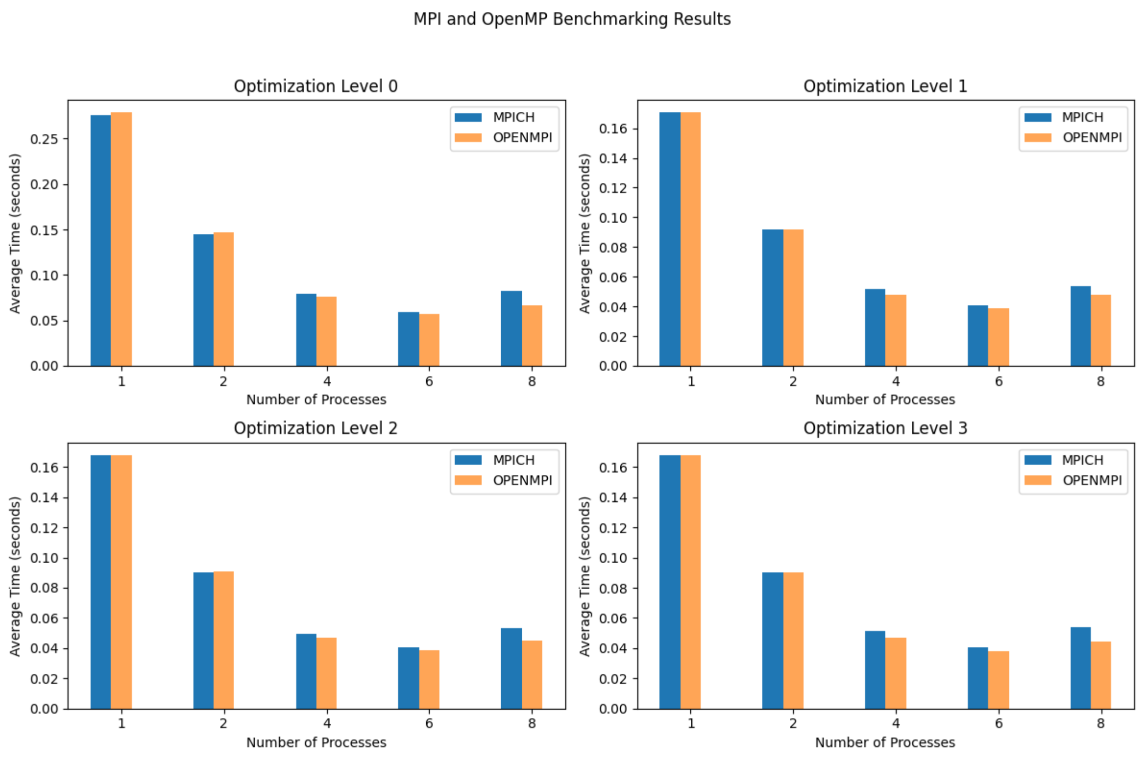


Figure 3: Time results for Machine 3

Processes	t-statistic	p-value	significant difference
1	1.0441214704596726	0.373157561454249	No
2	1.3848744002341007	0.2601098606125628	No
4	-8.816074058266999	0.0030752925527587982	Yes
6	-8.316926871760355	0.0036427462141564947	Yes
8	-4.567377742680935	0.01968683836843631	Yes

Table 4: T-statitics and p-values provided by the T-tests for Machine 3

4.4 Performance Insights and Observations

Our performance analysis revealed a difference in the execution times of the OpenMP and MPICH libraries across a range of hardware and operating systems. The variations in execution times, were substantial in some cases, highlighting the importance of selecting the right MPI library for specific environments.

The results, which can be found in figures 1, 2 and 3, showed a performance boost of OpenMP over MPICH, in environments with increasing core numbers. This theme holds true between all three test machines, operating systems, and processor architectures.

The results of the T-tests, which can be seen in tables 2, 3 and 4, back up these claims, as they show a p-value lower than 0.05 for Machine 1 and Machine 3 when using 4, 6, and 8 processes (cores), stating that there is a significant difference with these exact prerequisites.

5 Discussion

In this section, we will delve into the implications of our performance analysis within specific environments.

5.1 Relevance of Performance Differences

The results indicate the performance differences between the three different CPUs. Although a direct comparison between the three machines is not reasonable since we can not determine which of the factors (operating system, library or CPU), is the underlying cause, the resulting conclusions are still relevant.

The performance of MPICH vs OpenMP in all environments seems to be pretty consistent. With an increasing number of cores, OpenMP seems to outperform MPICH.

5.2 Hardware and OS Impact

Our analysis underscored the influence of hardware and operating systems on MPI library performance. Machine specifications, including RAM capacity as well as throughput and processor type, play a crucial role in determining the performance of a system as well as any library that runs on said system. Furthermore, the choice of operating systems, such as Windows, macOS, or Linux, influenced the performance outcomes. Developers should consider these factors when selecting an MPI library, as the optimal choice may vary depending on the underlying hardware and OS.

5.3 Compiler Optimization Levels

Compiler optimization levels had a noticeable impact on performance. Developers might want to use lesser optimization levels during development to make it easier to debug. However, this can lead to skewed results in terms of performance tests, since the results may indicate a different result than the resulting end system with higher performance levels.

In some cases, higher optimization levels led to improved performance for both OpenMP and MPICH. An example of this can be seen in the results from Machine 1 with different optimization levels, see Figure 1. If you look at the performance results from Level 1, it shows that OpenMPI performs slightly better on several processes greater than 2. This would indicate that this library leads to better results. However, if you now look at the results from Level 3, you see that MPICH outperforms OpenMPI, Indicating that MPICH would be more beneficial on higher performance levels.

5.4 MPI Overhead for Smaller Computation

If you take the results from Machine 3 you can see that the average time per computation increases from six to eight cores. Our interpretation of this is that the message passing has some overhead attached to it, and the computation that we are performing is not benefiting enough from the distribution to outweigh the overhead that the message passing introduces. Therefore, and due to the fact that Machine 3 was by far the fastest machine to run this simulation, we assume that you can expect to see similar results with an even higher amount of cores.

5.5 Applicability for Developers

While our study provides valuable insights into the performance of OpenMP and MPICH on our testbed, it is essential to acknowledge that the results may not be universally applicable. The results might differ for individual MPI calls, for example, OpenMP could outperform a specific MPI call that is used a lot in a potential project, compared to what we have used. Similarly, MPICH could outperform on a different call that is used in a different project. The choice of MPI library should be considered on a case-by-case basis, taking into account the requirements and characteristics of the target environment.

5.6 Ethical Considerations

Our study raises ethical considerations regarding energy consumption and sustainability. If performance differences in MPI libraries lead to variations in power consumption, developers have a responsibility to select the most energy-efficient option. Minimizing energy usage aligns with ethical principles in sustainable computing and reduces the environmental impact of distributed computing applications.

5.7 OpenMP Directives

This section will delve into an interesting aspect that we encountered during this project, it will explain a simplification for OpenMP which can be used to simplify the work of a developer.

OpenMP directives are a set of (pre-)compiler instructions and library functions that allow you to simplify the development of parallelized code. For our example of appendix B, one could parallelize this function as follows:

```
double Sum2SqrtArray(int* data, long int size) {
    double sum = 0.0;
    #pragma omp parallel for reduction(+:sum)
    for (int i = 0; i < size; ++i) {
        sum += sqrt(sqrt(data[i]) + sqrt(data[i]));
    }
    return sum;
}
```

The instruction can be explained as follows:

- a ”pragma”, is a compiler directive that allows you to give information to the compiler.
- ”omp” is short for OpenMP.
- ”parallel” defines a parallel region for OpenMP, it allows the following instructions (or region of instructions) to be executed by multiple threads in parallel
- ”for” specifies that the work inside the for loop is to be completed in parallel
- ”reduction (operation:var)”, is an extension to the ”for” command that specifies that the operation is to be executed on the distributed results. This means that the ”+” operation is to be executed on the distributed results of the sum variable and then loaded into the final ”sum” which can then be returned.

This leads us to an interesting problem. At the beginning of this project, we assumed that since OpenMP and MPICH are implementations of the same interface, all operations were implemented based on the interface, and therefore all operations that can be performed work for both libraries. This can be a pitfall for developers unfamiliar with the libraries.

Additionally, we have performed the performance tests for this set of instructions as well. You can see the results of this test in figure 4.

As you can see the results indicate a very varying performance gain or even loss. While Machine 1 profits from parallelizing this way until 4 cores and then loses performance. Machine 2 and Machine 3 seem to be losing performance with an increasing number of cores in this implementation.

6 Future Work

This section highlights the most prominent directions to explore in future work.

First, it would be interesting to get further insight into the compiler directives that OpenMP offers. The performance difference is significant enough to warrant a manual implementation of distribution and reduction. Since our knowledge about the internal translation/compilation into machine code is limited, it would be nice to know how OpenMP actually translates the code, to see what differences there are and if there are any performance improvements possible.

Second, while the results indicate a performance benefit of OpenMP over MPICH for higher numbers of cores, we do not know whether or not this would still hold true for a large-scale supercomputer such as Dardel here at KTH. Since supercomputers usually have a more optimized pipeline of tools at their disposal that have been optimized for them specifically, it remains to be seen whether or not we can reproduce similar results in an environment like this.

7 Conclusion

- RQ1: Is there a significant performance difference between the two MPI Libraries in our specific environment?

Yes, there appears to be a significant performance difference between OpenMP and MPICH in our specific environment. OpenMP demonstrated better performance, especially as the number of cores increased on all three test machines. However, the magnitude of this performance difference varied depending on the specific environment and the number of cores.

- RQ2: Does a potential performance difference warrant a further testing stage during product development?

Yes, in our interpretation, a potential performance difference between MPI libraries does warrant further testing during product development. The observed performance differences between OpenMP and MPICH in different environments and scenarios indicate that developers should carefully evaluate and choose the appropriate MPI library for their specific application. Testing it beforehand can help select the best-performing library for a particular project.

- RQ3: Are the results found on our test setup applicable to other machines, or should every target machine be tested individually to find the optimal tool-chain?

While our results provide valuable insights into the performance of OpenMP and MPICH in specific environments, they may not be directly applicable to all other machines. The choice of MPI library should be considered on a case-by-case basis, taking into account the target machine's hardware, operating system, and other specific requirements. Testing on each target machine individually or in representative environments is advisable to find the optimal tool-chain and ensure the best performance for a given application.

In conclusion, our performance analysis of OpenMP and MPICH on three different machines with varying hardware and operating systems has provided valuable insights into the choice of MPI library for distributed computing applications. The results indicate that the performance of these libraries can vary significantly based on the environment, including hardware specifications and the choice of operating system.

References

[1] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open mpi: Goals, concept, and design of a next generation mpi implementation," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, D. Kranzlmüller, P. Kacsuk, and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. ISBN 978-3-540-30218-6 pp. 97–104.

[2] J. Liu, B. Chandrasekaran, J. Wu, W. Jiang, S. Kini, W. Yu, D. Buntinas, P. Wyckoff, and D. K. Panda, "Performance comparison of mpi implementations over infiniband, myrinet and quadrics," in *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, ser. SC '03. New York, NY, USA: Association for Computing Machinery, 2003. doi: 10.1145/1048935.1050208. ISBN 1581136951 p. 58. [Online]. Available: https://doi.org/10.1145/1048935.1050208

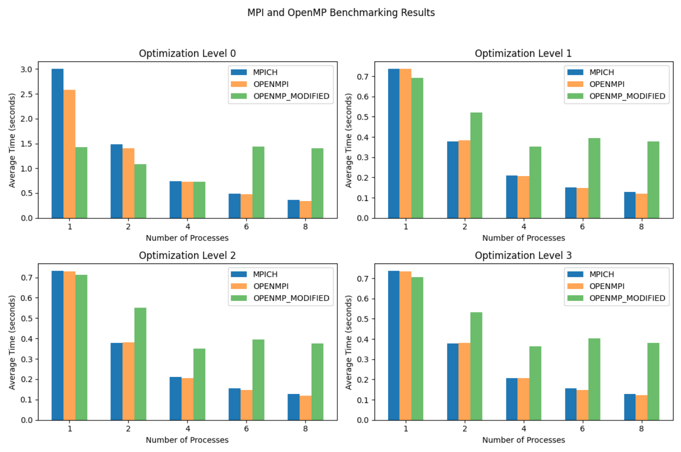
[3] G. Krawezik, "Performance comparison of mpi and three openmp programming styles on shared memory multiprocessors," in *Proceedings of the Fifteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '03. New York, NY, USA: Association for Computing Machinery, 2003. doi: 10.1145/777412.777433. ISBN 1581136617 p. 118–127. [Online]. Available: https://doi.org/10.1145/777412.777433

[4] G. Krawezik and F. Cappello, "Performance comparison of mpi and openmp on shared memory multiprocessors," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 1, pp. 29–61, 2006. doi: https://doi.org/10.1002/cpe.905. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.905

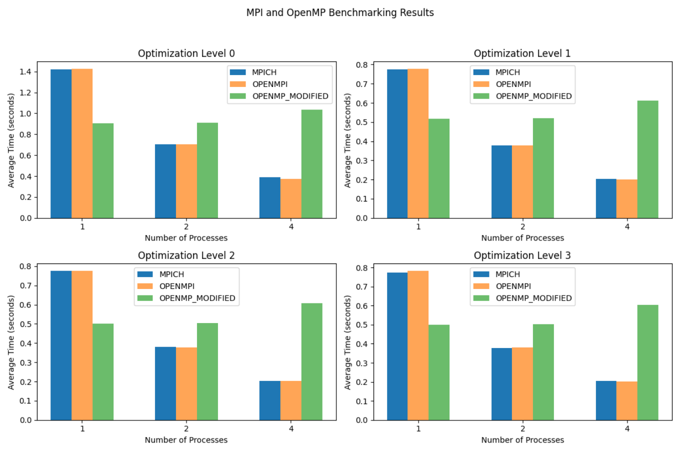
[5] R. Dimitrov and A. Skjellum, "Software architecture and performance comparison of mpi/pro and mpich," in *Computational Science — ICCS 2003*, P. M. A. Sloot, D. Abramson, A. V. Bogdanov, Y. E. Gorbachev, J. J. Dongarra, and A. Y. Zomaya, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003. ISBN 978-3-540-44863-1 pp. 307–315.

[6] M. Xu, D. Fralick, J. Z. Zheng, B. Wang, X. M. Tu, and C. Feng, "The differences and similarities between two-sample t-test and paired t-test," Jun 2017. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5579465/

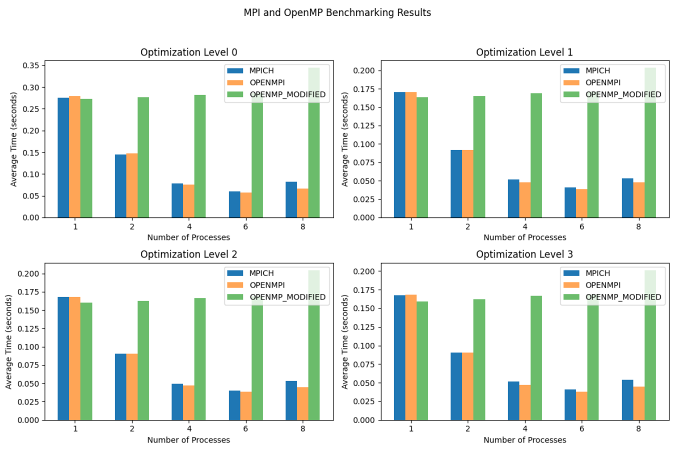
A Results - Including OpenMP Directives



(a) Time results for Machine 1



(b) Time results for Machine 2



(c) Time results for Machine 3

Figure 4: Time results for Machines 1, 2, and 3

B Function to be Parallelized

```
double Sum2SqrtArray(int* data, long int size) {
    double sum = 0.0;
    for (int i = 0; i < size; ++i) {
        sum += sqrt(sqrt(data[i]) + sqrt(data[i]));
    }
    return sum;
}
```

C Function Parallelized

```
double Sum2SqrtArray(int* data, long int size) {
    double local_sum = 0.0;
    double sum = 0.0;
    int rank, comm_size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
    int chunk_size = size / comm_size;
    int* local_data = (int*)malloc(chunk_size * sizeof(int));
    MPI_Scatter(&data[0], chunk_size, MPI_INT, local_data,
        chunk_size, MPI_INT, 0, MPI_COMM_WORLD);
    for (int i = 0; i < chunk_size; ++i) {
        local_sum += sqrt(sqrt(local_data[i]) + sqrt(local_data[i]));
    }
    MPI_Reduce(&local_sum, &sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    free(local_data);
    return sum;
}
```