

Darmstadt University of Applied Sciences
– Faculty of Computer Science –

**Lab report for the first high-performance
computing exercise**

by
Zebreus

INTRO

In this lab report, we will compare and evaluate four implementations of an algorithm for calculating the number π . All four implementations use different libraries to make the algorithm run in parallel. We will focus on comparing the performance concerning the number of threads and the precision of π . The algorithm's number of iterations determines the precision of π . We will call the number of iterations `steam:[n]` or the problem size.

First, we will run some small-scale local tests to understand the problem, and then we will run some large-scale tests on the Virgo cluster to get some performance data for bigger thread counts and problem sizes.

BENCHMARKING THE IMPLEMENTATIONS ON A LAPTOP

We tested for different parallel implementations with 1, 2, 4, and 8 threads on Lennart's computer (shown in [Table 1](#)). Each implementation was tested ten times for each number of threads and problem size.

We noticed a significant performance difference between the executables created with GCC and clang. [Figure 1](#) compares the performance of the different compilers. It should be noted that we did not further investigate the performance difference between the two compilers; it could be that the version of GCC shipped with Nix is, for some reason, producing slower code than the version of clang shipped with Nix.

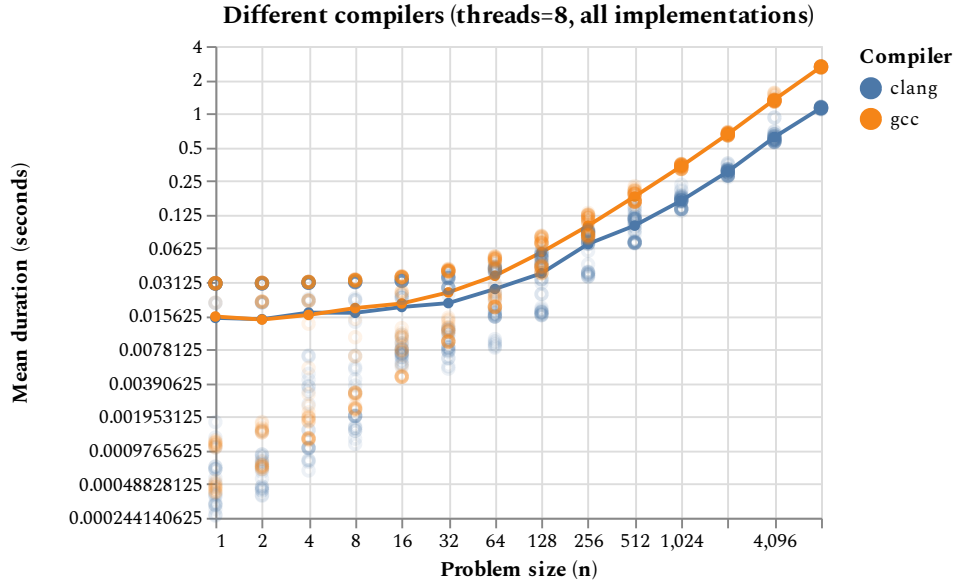


Figure 1. GCC 12 vs. Clang 16

We decided only to use clang for our further measurements. The optimization level was set to `-O3` for all measurements. We measured the performance for 1, 2, 4, and 8 threads. If not specified otherwise, the value for `steam:[n]` is set to 8192.

MEASUREMENTS

[Figure 2](#) shows the mean duration for all implementations for a problem with size 2048. We can see that there is no significant difference between the implementations. However, a different view emerges if we compare different problem sizes for a fixed number of threads. [Figure 3](#) shows that the MPI implementations seem to incur a fixed overhead of 0.03 seconds (at least for

eight threads). For bigger problem sizes, that overhead becomes negligible as the execution time becomes a more significant factor. For problem sizes less than 2048, the overhead becomes relevant.

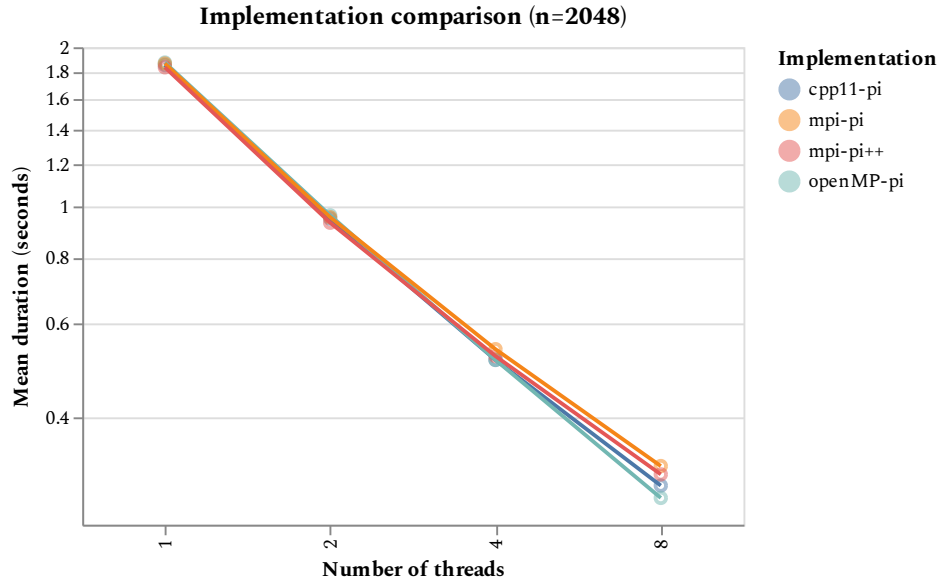


Figure 2. Mean performance for a problem size of $n=2048$

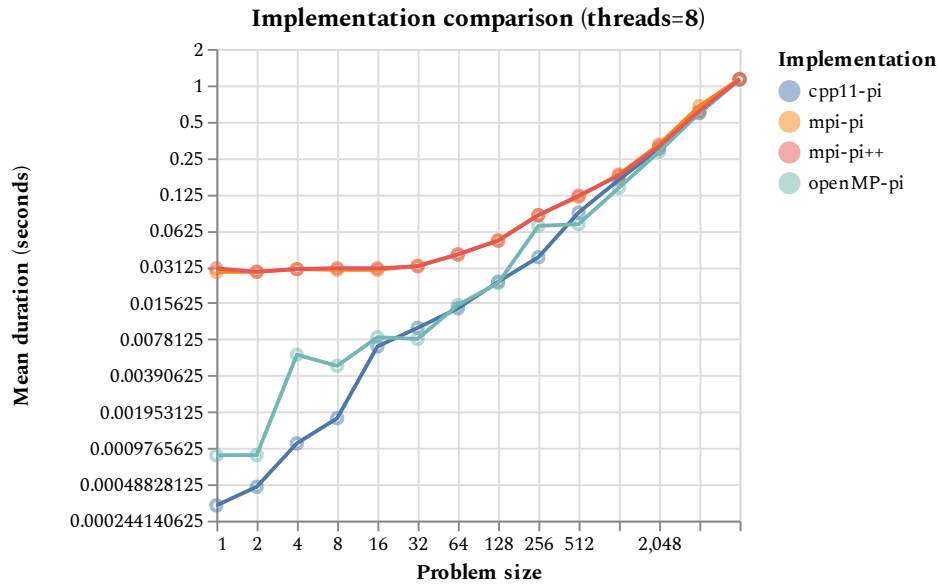


Figure 3. Mean performance with eight threads

This slowdown for the MPI-based implementations seems to be related to the number of threads used. Figure 4 shows that the slowdown is not present when using only a single thread but increases with the number of threads used.

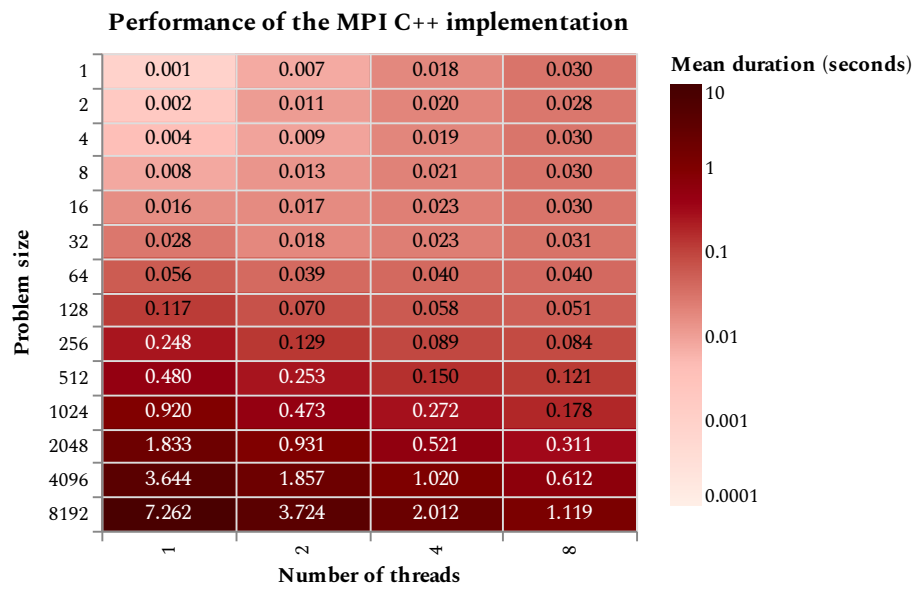


Figure 4. Mean performance for a problem size of $n=2048$

BENCHMARKING THE IMPLEMENTATIONS ON THE VIRGO CLUSTER

NOTE

Test setup

- 128 threads is consistently the highest number of cores for one node on the main partition.
- 1 Job tests all four implementations once for each n.
- For each tested number of threads, we ran ten jobs.
- Compiled with GCC, as we did not manage to install clang on the Virgo cluster.
- We made sure that enough hardware threads were available for every thread.
- We tried to ensure that only one thread runs on each real CPU core but probably failed.

MEASUREMENTS

[Figure 5](#) compares the execution duration of the different implementations when running with only a single thread. All four implementations perform nearly identically. This is expected as there should be virtually no overhead for the different parallelization methods when only using a single thread.

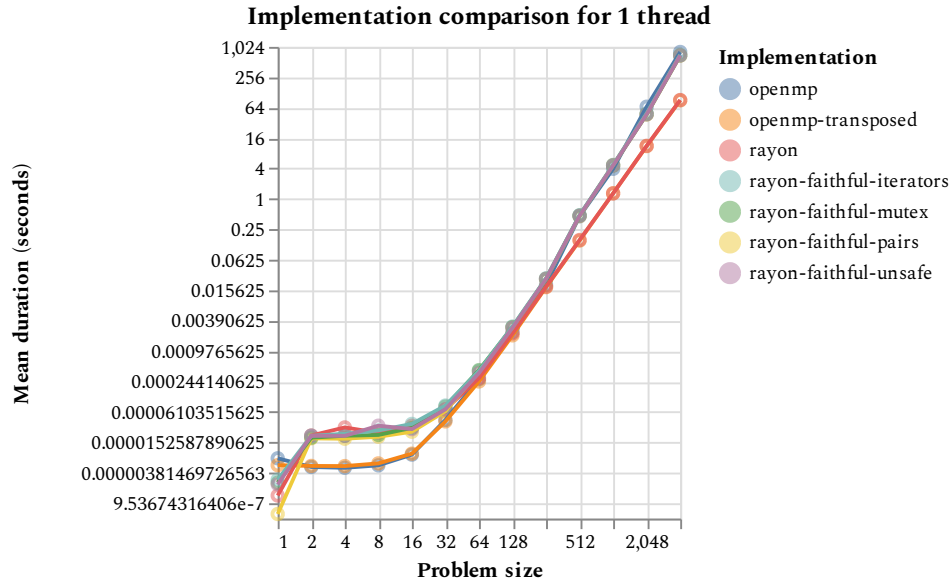


Figure 5. Performance of the different implementations with one thread

NOTE

I like how this shows the linear relationship between the problem size and the execution time if we are not doing any parallelization. This also demonstrates that our test setup is not measuring any weird delays from somewhere else.

Figure 6 and Figure 7 show that when using a more considerable number of threads, all implementations still behave similarly for big problem sizes. For smaller problem sizes, the performance of the implementations diverges. The overhead per thread seems to be the lowest when using MPI, higher for native C++ threads, and higher for openMP implementation. If the problem is big enough, the overhead of the different implementations is negligible.

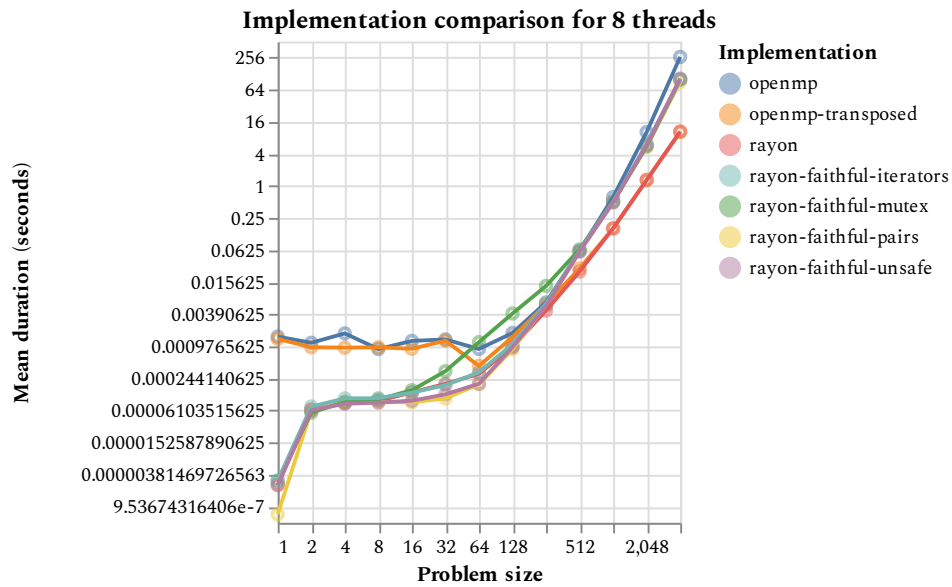


Figure 6. Performance of the different implementations with eight threads

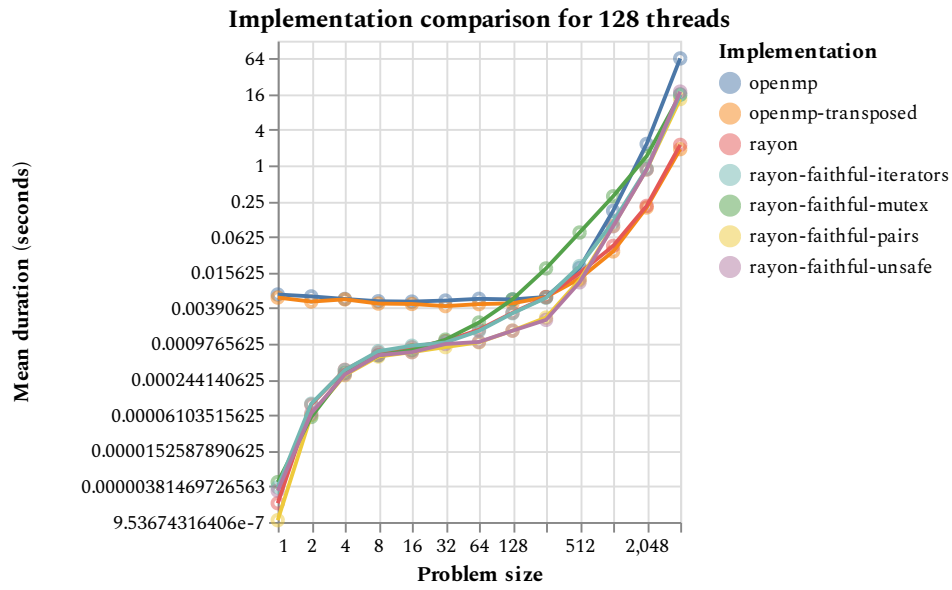


Figure 7. Performance of the different implementations with 128 threads

We can see that both mpi implementations behave nearly identically. In the more detailed charts, the MPI C implementation will be omitted.

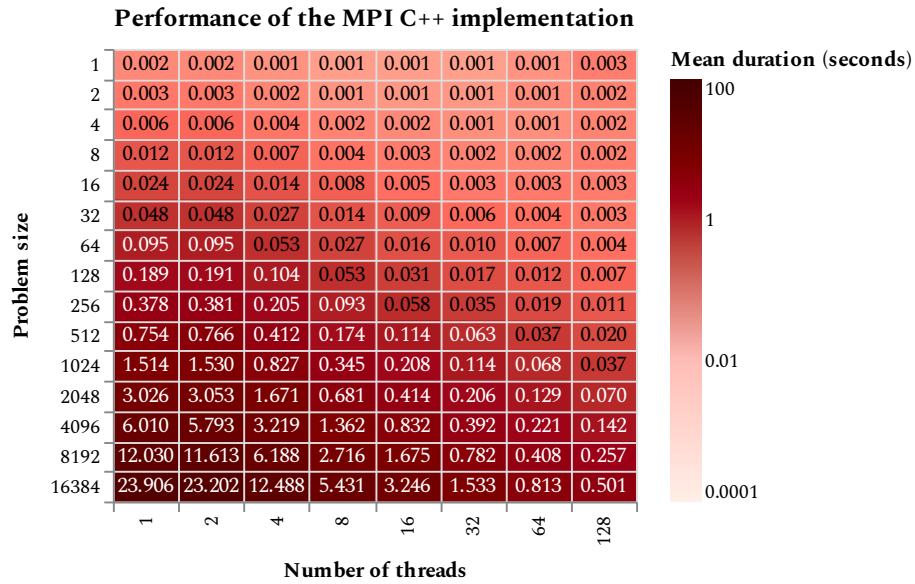


Figure 8. Performance of mpi-cpp

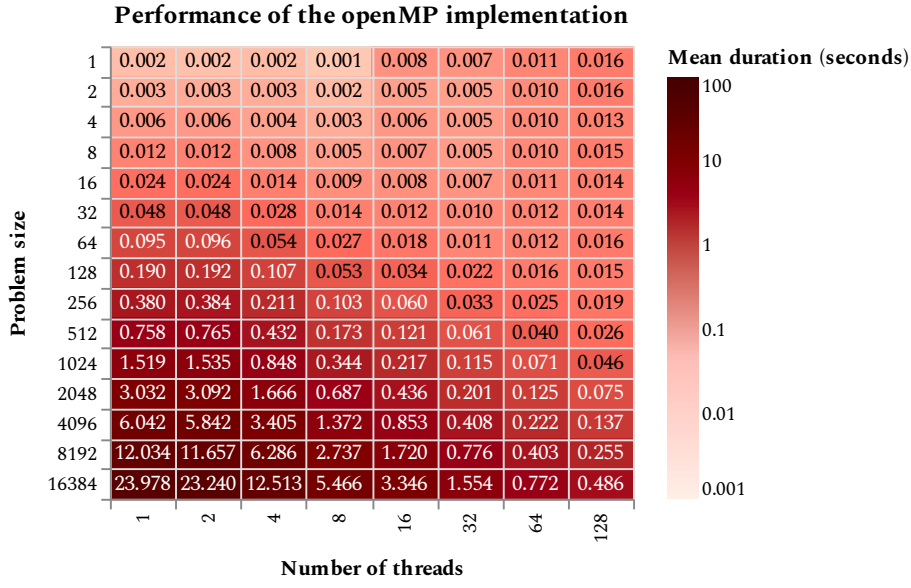


Figure 9. Performance of OpenMP

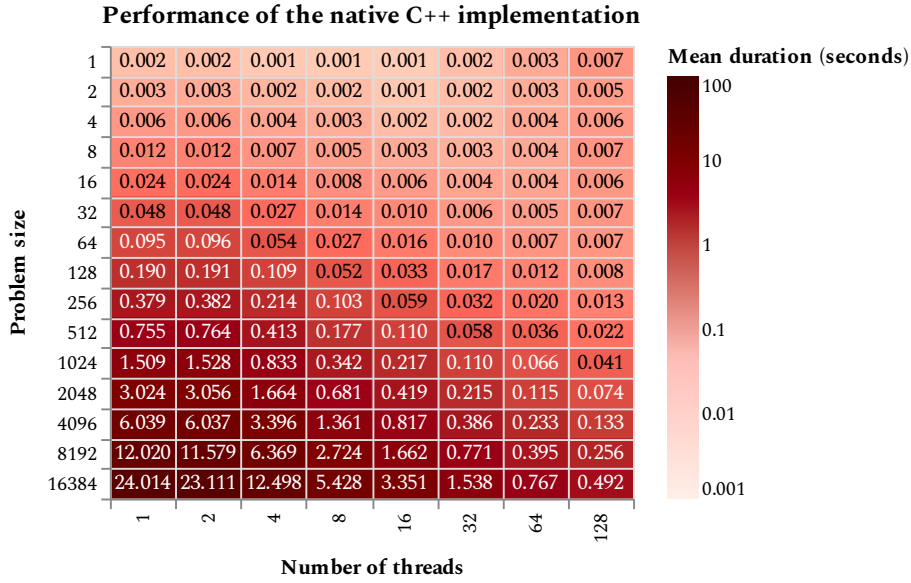


Figure 10. Performance of native C++ threads

ADVANCED EVALUATION

Figure 11 shows the relative speedup when running the MPI C++ implementation with multiple threads. The speedup is always relative to the execution time of the same problem size with a single thread.

It seems like there is no speedup when using two threads instead of one. We suspect that this is due to SMT. When we request two cores from Slurm, we probably only get one real core, and the second one is just another thread on

the same core. This could explain why the speedup for two cores is close to one. Another theory is that the overhead of MPI starting a second thread cancels out precisely the speedup from using two threads.

The figure shows that for bigger problem sizes, the speedup nearly doubles when doubling the number of threads after the second thread. It is not a perfect doubling because the overhead of starting the threads increases with the number of threads. With bigger problem sizes, the overhead becomes less significant, and the speedup approaches a doubling.

It is interesting to see that for very small problem sizes, the overhead for starting many threads is so significant that the speedup becomes less than one.

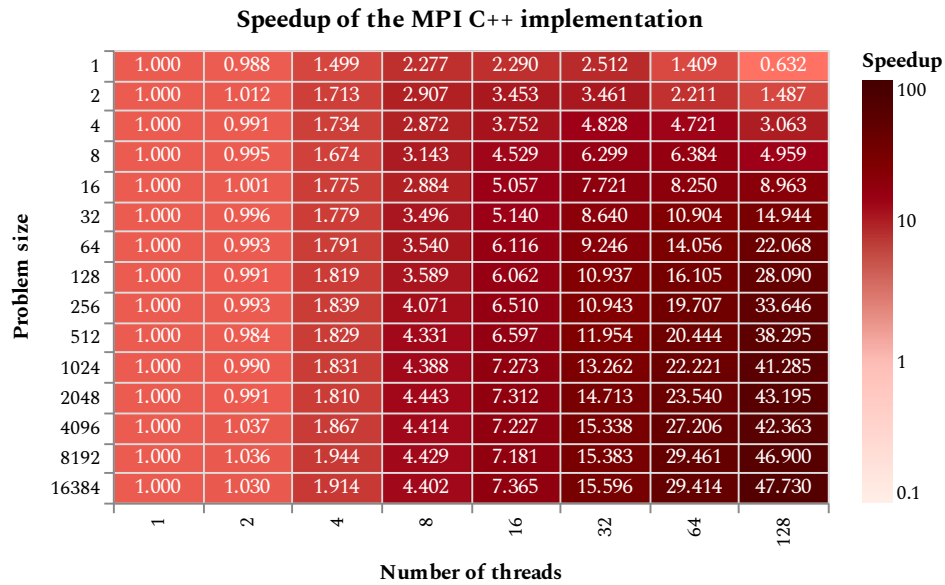


Figure 11. Speedup of mpi-cpp

The efficiency is defined as the speedup divided by the cost. In our example, the cost is defined as the number of threads used. [Figure 12](#) shows that the efficiency decreases the more threads are used. However, the rate of decrease is getting smaller for bigger problem sizes. This is because the overhead of starting the threads becomes less significant for bigger problem sizes.

[Figure 12](#) also shows again that we seem to have a problem with our measurements, as the efficiency drops to 50% after the first doubling.

It is interesting to see that the efficiency for eight threads is consistently slightly better than expected. We have not found any explanation for this.

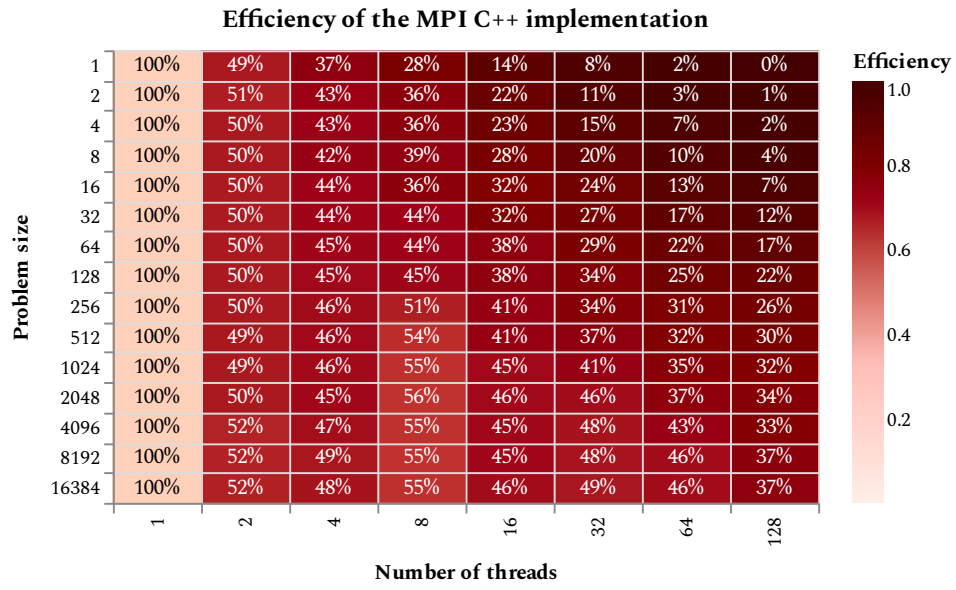


Figure 12. Efficiency of mpi-cpp

CONCLUSION

While our measurements and results may not be perfect, we have gained hands-on experience with the Virgo cluster and learned to use OpenMP and MPI.

FUTURE WORK

We could investigate why we did not experience any speedup when using two threads vs one thread on the Virgo cluster. If the SMT theory is correct, it is possible that we could get a speedup by requesting two cores instead of two threads.

Making measurements for problem sizes bigger than $n = 16384$ would also be interesting.


In our measurements, we only benchmarked MPI on a single host. While this brought some level of comparability to openMP and native C++ threads, this is not the way MPI is supposed to be used. It would be interesting to see how MPI compares when using multiple hosts.

Our time management skills also have room for improvement, as we did not manage to hand in this report in time. In future work, we should probably try to start earlier.

We also were not able to use clang on the Virgo cluster. It would be interesting to see if the performance difference between clang and gcc we measured on our machine is also present on the Virgo cluster.

LIST OF ABBREVIATIONS

SMT

Simultaneous multithreading 

APPENDIX

Table 1. Our computers

	Lennart	Björn
CPU	i9-11950H @ 2.6GHz	i5-7200U @ 2.5GHz
CPU-Kerne / Threads	8 / 16	2 / 4
GPU	RTX3070 mobile	Intel HD Graphics 620
OS	NixOS unstable/latest - 64 Bit	Ubuntu 23.04 - 64 Bit
RAM (GB)	32	8

Listing 1. C++ threads implementation

```
1 /* -*- Mode: C++; c-basic-offset:4 ; -*- */
2 /* This code borrows heavily from the file mpi-c from the
3  * Argonne National Laboratory.
4  */
5
6 #include <chrono>
7 #include <iomanip>
8 #include <iostream>
9 #include <sstream>
10 #include <string>
11 #include <thread>
12 #include <vector>
13 // uncomment to disable assert()
14 // #define NDEBUG
15 #include <cassert>
16 #include <cmath> // for fabs()
17
18 // ***** Utilities
19 int string2int(const std::string& text) {
20     std::stringstream temp(text);
21     int result = 0xffffffff;
22     temp >> result;
23     return result;
24 } // end string2int
25
26 // ***** Functions for calculating PI (borrowed from mpi)
27 double f(double a) {
28     return (4.0 / (1.0 + a * a));
```

CPP

```

29 } // end f
30
31 const double PI25DT =
32     3.141592653589793238462643; // No, we're not cheating -this is for testing!
33
34 const double maxNumThreads = 1024; // this is only for sanity checking
35
36 long n;
37 double h;
38
39 void pi_thread(int thread_num, int numThreads, double* partial_pi) {
40     assert(0 <= thread_num);
41     assert(thread_num < maxNumThreads);
42     double sum = 0.0;
43     /* It would have been better to start from large i and count down, by the way. *
44     /
45     for (long i = thread_num + 1; i <= n; i += numThreads) {
46         double x = h * ((double)i - 0.5);
47         sum += f(x);
48     }
49     *partial_pi = h * sum;
50 } // end pi_thread
51
52 // ***** main
53 int main(int argc, char* argv[]) {
54     int numThreads = 0;
55
56     if (3 == argc) {
57         numThreads = string2int(argv[1]);
58         n = string2int(argv[2]) * 10241 * 10241;
59         h = 1.0 / (double)n;
60     } else // if number of args illegal
61     {
62         std::cerr << "Usage: " << argv[0] << " number-of-threads n" << std::endl;
63         return (-1);
64     }; // end argc check
65
66     assert(0 < numThreads);
67     assert(numThreads <= maxNumThreads);
68
69     std::chrono::system_clock::time_point startTime =
70         std::chrono::system_clock::now();
71
72     std::thread threads[numThreads]; // Note: No REAL threads yet...
73     double partials[numThreads];
74
75     //Launch the threads

```

```

76     for (int i = 0; i < numThreads; ++i) {
77         threads[i] = std::thread(pi_thread, i, numThreads, &(partials[i]));
78     }
79
80     ///Join the threads with the main thread
81     double pi = 0;
82     for (int i = 0; i < numThreads; ++i) {
83         threads[i].join();
84         pi += partials[i];
85     }
86
87     std::chrono::system_clock::time_point endTime =
88         std::chrono::system_clock::now();
89     std::chrono::microseconds microRunTime =
90         std::chrono::duration_cast<std::chrono::microseconds>(
91             endTime - startTime
92         );
93     double runTime = microRunTime.count() / 1000000.0;
94
95     std::cout << std::setprecision(16) << "Pi is approximately " << pi
96         << ", Error is " << std::fabs(pi - PI25DT) << std::endl;
97     std::cout << std::setprecision(8) << "Wall clock time = " << runTime
98         << " seconds." << std::endl;
99     std::cout << "There were " << numThreads << " threads." << std::endl;
100    std::cerr << runTime << std::flush;
101    return 0;
102 }

```

Listing 2. C++ openMP implementation

```

1  /* -*- Mode: C++; c-basic-offset:4 ; -*- */
2  /* This code borrows heavily from the file mpi-c from the
3   * Argonne National Laboratory.
4   */
5
6  #include <chrono>
7  #include <iomanip>
8  #include <iostream>
9  #include <sstream>
10 #include <string>
11 // uncomment to disable assert()
12 // #define NDEBUG
13 #include <omp.h>
14
15 #include <cassert>
16 #include <cmath> // for fabs()
17
18 // ***** Utilities
19 int string2int(const std::string& text) {
20     std::stringstream temp(text);
21     int result = 0xffffffff;
22     temp >> result;
23     return result;

```

CPP


```

24 } // end string2int
25
26 // ***** Functions for calculating PI (borrowed from mpi)
27 double f(double a) {
28     return (4.0 / (1.0 + a * a));
29 } // end f
30
31 const double PI25DT =
32     3.141592653589793238462643; // No, we're not cheating -this is for testing!
33
34 const double maxNumThreads = 1024; // this is only for sanity checking
35
36 // ***** main
37 int main(int argc, char* argv[]) {
38     int numThreads = 0;
39     long n; // default # of rectangles (421 = long int 42)
40     double h;
41
42     if (3 == argc) {
43         numThreads = string2int(argv[1]);
44         n = string2int(argv[2]) * 10241 * 10241;
45         h = 1.0 / (double)n;
46     } else // if number of args illegal
47     {
48         std::cerr << "Usage: " << argv[0] << " number-of-threads n"
49             << std::endl;
50         return (-1);
51     }; // end argc check
52
53     assert(0 < numThreads);
54     assert(numThreads <= maxNumThreads);
55
56     std::chrono::system_clock::time_point startTime =
57         std::chrono::system_clock::now();
58
59     double sum = 0;
60
61     // tag::openmp[]
62     // *** Here is the OpenMP Magic!! (All in one line) ***
63     #pragma omp parallel for num_threads(numThreads) reduction(+ : sum)
64
65     for (long i = 1; i <= n; i += 1) {
66         double x = h * ((double)i - 0.5);
67         sum += f(x);
68     }
69     // end::openmp[]
70     /* It would have been better to start from large i and count down, by the way. */
71

```

```

72     double pi = h * sum;
73
74     std::chrono::system_clock::time_point endTime =
75         std::chrono::system_clock::now();
76     std::chrono::microseconds microRunTime =
77         std::chrono::duration_cast<std::chrono::microseconds>(
78             endTime - startTime
79         );
80     double runTime = microRunTime.count() / 1000000.0;
81
82     std::cout << std::setprecision(16) << "Pi is approximately " << pi
83         << ", Error is " << std::fabs(pi - PI25DT) << std::endl;
84     std::cout << std::setprecision(8) << "Wall clock time = " << runTime
85         << " seconds." << std::endl;
86     std::cout << "There were " << numThreads << " threads." << std::endl;
87
88     std::cerr << runTime << std::flush;
89
90     return 0;
91 }

```

Listing 3. C++ MPI implementation

```

1  /* -*- Mode: C++; c-basic-offset:4 ; -*- */
2  /* This code borrows heavily from the file mpi-c from the
3   * Argonne National Laboratory.
4   */
5
6  #include <chrono>
7  #include <iomanip>
8  #include <iostream>
9  #include <fstream>
10 #include <sstream>
11 #include <string>
12 // uncomment to disable assert()
13 // #define NDEBUG
14 #include <mpi.h>
15
16 #include <cassert>
17 #include <cmath> // for fabs()
18
19 // ***** Functions for calculating PI (borrowed from mpi)
20 double f(double a) {
21     return (4.0 / (1.0 + a * a));
22 } // end f
23
24 const double PI25DT =
25     3.141592653589793238462643; // No, we're not cheating -this is for testing!
26
27 // ***** Utilities
28 int string2int(const std::string& text) {
29     std::stringstream temp(text);
30     int result = 0xffffffff;

```

CPP

```

31     temp >> result;
32     return result;
33 }
34
35 // ***** main
36 int main(int argc, char* argv[]) {
37     /** Standard MPI opening boilerplate **/
38     int myid, numprocs, namelen;
39     char processor_name[MPI_MAX_PROCESSOR_NAME];
40     MPI_Init(&argc, &argv);
41     MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
42     MPI_Comm_rank(MPI_COMM_WORLD, &myid);
43     MPI_Get_processor_name(processor_name, &namelen);
44
45     std::cout << "Process " << myid << " of " << numprocs << ", running on "
46               << processor_name << std::endl;
47
48     /** Actual work starts here */
49
50     double startwtime = 0.0;
51     // Strictly speaking, we could have made n a constant, like it is in our other p
rograms,
52     // but this way we can demonstrate how a broadcast works.
53     long n; // n is the number of rectangles
54     if (0 == myid) {
55         startwtime = MPI_Wtime();
56         n = strtol(argv[1], nullptr, 10) * 1024l * 1024l;
57     };
58
59     MPI_Bcast(
60         &n,
61         1,
62         MPI_LONG_INT, // Broadcast n, which is 1 long integer, where
63         0,
64         MPI_COMM_WORLD
65     ); // ID 0 sends & everyone else in COMM_WORLD receives.
66
67     double h = 1.0 / (double)n;
68     double sum = 0;
69
70     /* It would have been better to start from large i and count down, by the way. */
71     for (long i = myid + 1; i <= n; i += numprocs) {
72         double x = h * ((double)i - 0.5);
73         sum += f(x);
74     }
75
76     double pi, mypi = h * sum;

```

```

77     MPI_Reduce(
78         &mypi,
79         &pi,
80         1,
81         MPI_DOUBLE,
82         MPI_SUM, // Everyone's copy of mypi (one double) are summed up,
83         0,
84         MPI_COMM_WORLD
85     ); // and the result is sent to ID 0, and stored in pi.
86
87     if (0 == myid) {
88         double endwtime = MPI_Wtime();
89         std::cout << std::setprecision(16) << "Pi is approximately " << pi
90             << ", Error is " << fabs(pi - PI25DT) << std::endl;
91         std::cout << "Wall clock time = " << (endwtime - startwtime)
92             << " seconds." << std::endl;
93         std::cout << "There were " << numprocs << " processes." << std::endl;
94
95         std::cerr << (endwtime - startwtime) << std::flush;
96     }
97
98
99
100     MPI_Finalize();
101
102     return 0;
103 }

```

Listing 4. C MPI implementation

```

1  /* -*- Mode: C; c-basic-offset:4 ; -*- */
2  /*
3   * (C) 2001 by Argonne National Laboratory.
4   * See COPYRIGHT in top-level directory.
5   */
6
7  #include <math.h>
8  #include <mpi.h>
9  #include <stdio.h>
10 #include <stdlib.h>
11
12 double f(double a) {
13     return (4.0 / (1.0 + a * a));
14 }
15
16
17
18 int main(int argc, char* argv[]) {
19     // Strictly speaking, we could have made n a constant, like it is in our other pr
20     // ograms,
21     // but this way we can demonstrate how a broadcast works.
22     long i, n; // n is the number of rectangles, i is the rectangle number.
23     int myid, numprocs;

```

```

23  const double PI25DT = 3.141592653589793238462643;
24  double mypi, pi, h, sum, x;
25  double startwtime = 0.0, endwtime;
26  int namelen;
27  char processor_name[MPI_MAX_PROCESSOR_NAME];
28
29  /** Standard MPI opening boilerplate */
30  MPI_Init(&argc, &argv);
31  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
32  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
33  MPI_Get_processor_name(processor_name, &namelen);
34
35  fprintf(
36      stdout,
37      "Process %d of %d is on %s\n",
38      myid,
39      numprocs,
40      processor_name
41  );
42  fflush(stdout);
43
44  /** Actual work starts here */
45
46  if (0 == myid) {
47      startwtime = MPI_Wtime();
48
49      n = strtol(argv[1], 0, 10) * 10241 * 10241;
50  };
51
52  MPI_Bcast(&n, 1, MPI_LONG_INT, 0, MPI_COMM_WORLD);
53
54  h = 1.0 / (double)n;
55  sum = 0.0;
56  /* It would have been better to start from large i and count down, by the way. */
57  for (i = myid + 1; i <= n; i += numprocs) {
58      x = h * ((double)i - 0.5);
59      sum += f(x);
60  }
61  mypi = h * sum;
62
63  MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
64
65  if (0 == myid) {
66      endwtime = MPI_Wtime();
67      printf(
68          "Pi is approximately %.16f, Error is %.8e\n",
69          pi,
70          fabs(pi - PI25DT)

```

```
71     );  
72     printf("Wall clock time = %.8f seconds.\n", (endtime - startwtime));  
73     fprintf(stderr, "%.8f", (endtime - startwtime));  
74     fflush(stderr);  
75     fflush(stdout);  
76 }  
77  
78 MPI_Finalize();  
79 return 0;  
80 }
```