Comparing Parallel Programming Language Models: OpenMP and MPI

Yantao Luo and Yuchen Zhao

Abstract

This report compares the overhead, programmability, performance and scalability of MPI and OpenMP parallel models using a series of benchmark programs on a given hardware platform. A literature survey is first conducted to provide a comprehensive review of previous research on MPI and OpenMP. The results show that both MPI and OpenMP can provide significant performance improvements over serial execution, but the optimal choice of parallel model depends on the specific application and hardware platform being used. The conclusions of the study suggest that developers should carefully consider the characteristics of their application and hardware platform when choosing between MPI and OpenMP, and that further research is needed to explore hybrid models that combine the two parallel models for improved performance and scalability.

Keywords: Parallel programming, OpenMP, MPI

1 Introduction

Since the end of Moore's law is in sight[1], parallel computing has become an essential technique to achieve high-performance computing for large-scale scientific and engineering applications. Two popular parallel models are OpenMP, which stands for Open Multi-Processing, and MPI, which stands for Message Passing Interface. These models can help programmers divide a task into smaller pieces and utilize multi processors for better performance. Both models are widely used in modern high performance computing.

1.1 History

OpenMP is managed by OpenMP Architecture Review Board (ARB) which involves leading software and hardware service providers, including Intel, Red Hat, etc[2]. The first distribution OpenMP Fortran Interpretations Version 1.0 was published in 1997[3]. It was firstly designed to parallelize regular iterations before version 2.0, but quickly evolved and can parallelize various problems. Now, it is supported by almost all mainstream compilers such as GCC, Clang, Intel Fortran and C/C++ compilers, etc, making it one of mainstream parallel programming models nowadays.

MPI started from a workshop discussing standards for message passing in a distributed memory environment, which was held on April 29–30, 1992 in Williamsburg, Virginia[4]. From then, the MPI working group was formed and published the first MPI distribution version 1.0 in 1994. Since it is based on a distributed memory environment, it is widely used in the modern high performance computing community.

1.2 Mechanism

OpenMP is a parallel programming model that works on shared memory systems[5]. It provides programmers a set of compiler directives rather than functions to implement various types of parallelization. The work is distributed to several threads created by OpenMP which share the same memory address space. This is the reason why OpenMP has much more communication efficiency compared with some distributed memory parallel models.

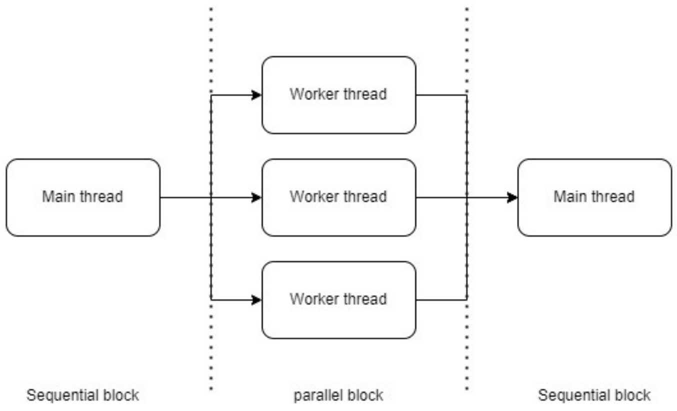

Fig. 1 OpenMP fork and join model

When programming with OpenMP, as shown in Figure 1, the parallel part and the sequential part are combined together as fork-join models[6]. From one sequential block to the following parallel block, the main thread creates several other threads to do the parallel work. After all worker threads have finished, including the main thread itself, a sequential block will begin. To be noticed, the OpenMP runtime provides the implementation of threadpool and many other library routines, which eases the programmers' job.

MPI is a parallel programming model designed for distributed memory systems. Compared to OpenMP, it provides the management of processes, instead of threads. Each process has its own memory address space, and programmers take the responsibility of partitioning the data and sending messages between those processes. This causes more communication overhead compared to shared memory parallelization.

The communication models in MPI comprise point-to-point, collective, one-sided, and parallel I/O operations.

2 Literature Survey

Comparing parallel programming models is a long last subject, and much research has been done in this area. Javier Diaz, etc[7] presented a thorough survey on different types of parallel models including OpenMP, MPI and OpenCL. They compared the mechanism of each model and researched the benefits of combining them to solve problems. Sol ji Kang, etc[8] designed a series of benchmarks to compare the performance of OpenMP, MPI and MapReduce in practical problems. Lorin Hochstein, etc[9], did an experiment to figure out the productivity difference between OpenMP and MPI. They organized a group of programmers and let them develop the same problem. Steve W. Bova, etc[10] did research on how to express and tune the performance of parallel programs by experimenting with different applications, such as CGWAVE, GAMESS, etc.

3 Proposed Idea

From the former research we know that OpenMP is more programmable and has less overhead when creating and destroying parallel workers. However, what is the scale of the difference? It is known that OpenMP is easier to program, but can we have a more clear view on how easier it is? What about the scalability differences between those two parallel programming models? In the work below, we will measure the scale of difference and provide a deeper insight into OpenMP and MPI performance. We will discuss the advantages and disadvantages of each parallel model and provide insights into which model is better suited for specific types of applications. This will help choose the appropriate parallel model for specific applications and achieve better performance and scalability.

4 Experiment Setup

We have designed three benchmark programs to compare different characteristics of OpenMP and MPI parallel programming models.

4.1 Workers Creation and Destruction

To compare the runtime overhead of OpenMP and MPI workers creation and destruction, we designed a benchmark program to measure the time it takes to repeatedly create and destroy workers in both models.

OpenMP worker thread creation and destruction pseudocode:

```
begin
    for i := 1 to number of threads do
        #pragma omp parallel num_threads(2)
        dummy_func()
    end for
end
```

In the above program, we used a for loop to repeatedly fork and join, each time creating one thread and destroying it. The total number of threads was given through command line arguments.

MPI worker processes creation and destruction pseudo code:

```
begin
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    dummy_func();

    MPI_Finalize();
end
```

In the above program, we used the MPI API to create a number of processes specified by the command line arguments, each process runs a dummy_func() and returns.

This benchmark run ten times and the execution times were recorded. The average time was calculated as the final result.

4.2 Reduction

In the Reduction Benchmark, we parallelized the reduction operation but used different parallelization approaches. OpenMP is a shared-memory parallel programming model and parallelizes the code using compiler directives, while handling the reduction operation implicitly through the reduction clause. MPI is a message-passing programming model and parallelizes the code by explicitly exchanging data among processes, performing the reduction using the MPI_Reduce() function.

This benchmark was designed to test the scalability, programmability and performance of each parallel programming model

Pseudo code for OpenMP implementation:

```
begin
    initialize an array of numbers
    get_time(time1)
    #pragma omp parallel for reduction(+:sum)
    for i := 0 to N do
        sum += (sin(arr[i]) + cos(arr[i]))
    end for
    get_time(time2)
    time = time2 − time1
end
```

Pseudo code for MPI implementation:

```
begin
    initialize MPI and an array of numbers
    get_time(time1)

    Divide the array among the processes using MPI_Scatter().
    Each process does the calculation for its chunk of the array.
    for i := 0 to chunk_size do
        local_sum += sin(local_arr[i]) + cos(local_arr[i])
    end for

    Use MPI_Reduce() to add local sums to the global sum
    get_time(time2)
    time = time2 - time1
end
```

4.3 Matrix Multiplication

Similar to the Reduction Benchmark, we designed a matrix multiplication benchmark to test the scalability, programmability and performance of each parallel programming model as well.

Pseudo code for OpenMP implementation:

```
begin
    initialize two input matrices and the result matrix
    get_time(time1)
    #pragma omp parallel for
    for i := 1 to N do
        for j:= 1 to N do
            for k := 1 to N do
                C[i*N+j] += A[i * N + k] * B[k * N + j]
            end for
        end for
    end for
    get_time(time2)
    time = time2 - time1
end
```

Pseudo code for MPI implementation:

```
begin
    initialize MPI and two input matrices and the result matrix
    get_time(time1)

    MPI_BCast(matrix B)
    MPI_Scatter(matrix A, chunk_number)
    for i := 1 to N / size do
        for j:= 1 to N do
            for k := 1 to N do
                C[i*N+j] += A[i * N + k] * B[k * N + j]
            end for
        end for
    end for

    MPI_Gather;
    get_time(time2)
    time = time2 - time1
end
```

5 Experiment and Analysis

The machine we run our benchmark programs on is the crunchy5 of the NYU CIMS system[10], which has four AMD Opteron 6272 2.1 GHz 64 cores CPU and 256GB memory. The operating system is CentOS 7.

5.1 Workers Creation and Destruction

Table 1 OpenMP threads creation time

Number of threads	total time(s)	average time(ms)
100	0.0004274	0.0043
1,000	0.0022969	0.0023
10,000	0.0239583	0.0024
100,000	0.2513301	0.0025
500,000	1.2195126	0.0024
1,000,000	2.5043612	0.0025

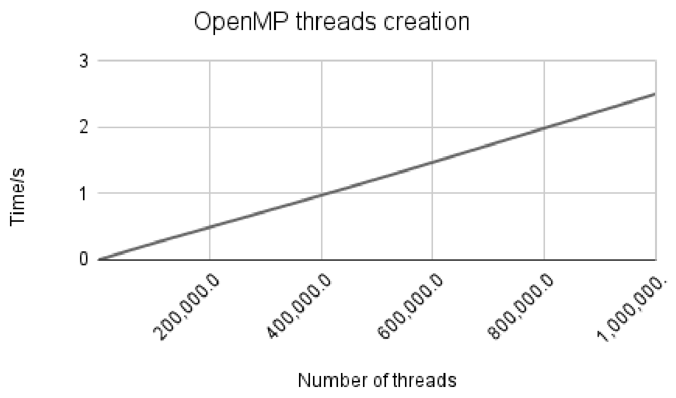

Fig. 2 OpenMP threads creation and destruction: the x-axis is the number of threads created and destroyed, the y-axis is the time it takes to complete the job

Table 2 MPI processes creation time

Number of processes	time(s)	average time(ms)
2	0.4554	227.7000
4	0.4954	123.8500
8	0.5641	70.5125
16	0.681	42.5625
32	1.028	32.1250
64	3.8691	60.4547

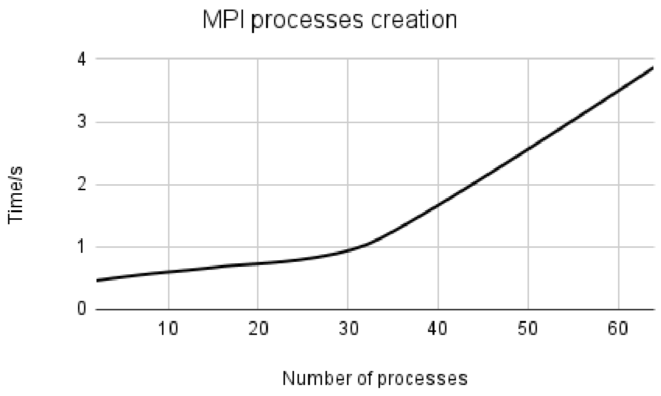

Fig. 3 MPI processes creation: the x-axis is the number of processes created and destroyed, the y-axis is the time it takes to complete the job

As shown in the table 1, fig.2, table2 and fig.3, the threads creation and destruction time for OpenMP is 0.0027ms on average, while processes creation and destruction time for MPI is around 92.9ms, which is 34000 times longer. The time for creating and deleting a single thread stays almost the same for OpenMP, while MPI takes more time to create a single process when the total number is less. This is probably because MPI runtime itself has more overhead.

5.2 Reduction

Table 3 Reduction Benchmark (4 Threads/Processes)

Array Length	OpenMP Time	MPI Time
1,000,000	0.045	0.056
10,000,000	0.437	0.490
50,000,000	2.237	2.500
100,000,000	4.431	5.076

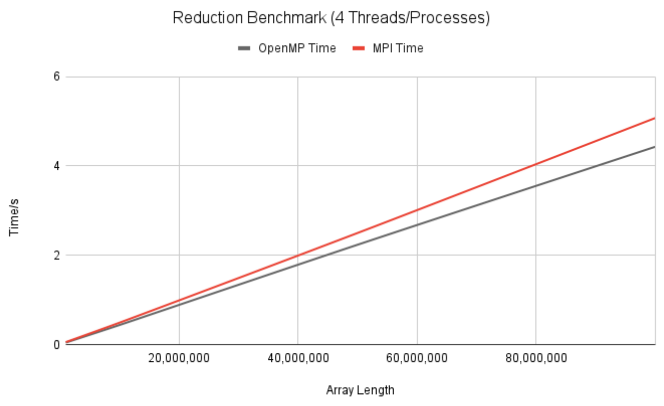

Fig. 4 Reduction Benchmark (4 Threads/Processes): the x-axis is the size of the array, the y-axis is the time it takes to complete the reduction sum process.

Table 4 Reduction Benchmark (32 Threads/Processes)

Array Length	OpenMP Time	MPI Time
1,000,000	0.010	0.022
10,000,000	0.083	0.166
50,000,000	0.348	0.728
100,000,000	0.684	1.443

OpenMP consistently outperforms MPI in terms of execution time for both 4 and 32 threads/processes. The difference in execution time can be attributed to the overheads associated with the message-passing model in MPI. When increasing the number of threads/processes from 4 to 32, both OpenMP and MPI show significant improvements in execution time across all array lengths. This indicates that both parallelization approaches can effectively utilize more computational resources to solve larger problems faster. The performance improvement from 4 to 32 threads/processes is more pronounced for OpenMP than for MPI. This is likely due to OpenMP's lower thread creation overhead and more straightforward parallelization approach.

5.3 Matrix Multiplication

The result of this part experiment is shown at Fig. 5-6, Table. 4-5. In this part, we measured the performance of OpenMP and MPI, considering the impact of the number of parallel workers and problem size on the speedup provided by the parallelization. Our experiments showed that the speedup provided by the extra workers is high when the total number is low, but the speedup converges to zero when the number of workers is too high. Specifically, MPI has a lower speed when problem size is 500 and processes number is 64, compared to the number of 32 condition. It is due to the higher overhead

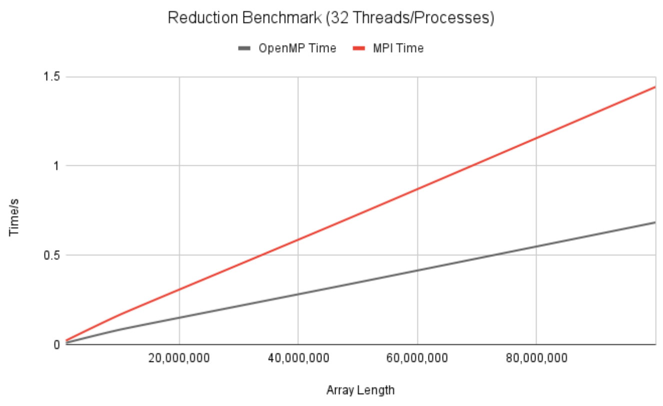

Fig. 5 Reduction Benchmark (32 Threads/Processes): the x-axis is the size of the array, the y-axis is the time it takes to complete the reduction sum process.

Table 5 Matrix Multiplication Benchmark (Matrix Dim = 500)

Number of threads/processes	OpenMP Time	MPI Time
2	1.572	1.471
4	0.842	0.949
8	0.442	0.440
16	0.229	0.250
32	0.157	0.190
64	0.124	0.912

Table 6 Matrix Multiplication Benchmark (Matrix Dim = 2000)

Number of threads/processes	OpenMP Time	MPI Time
2	107.678	93.080
4	57.116	48.406
8	29.781	25.826
16	15.851	13.783
32	8.629	7.544
64	6.555	8.732

creating a process compared to creating a thread, which is the conclusion of the first experiment.

5.4 Programmability

From the result shown in Table 7, our OpenMP benchmarks have less compiler instructions than the MPI ones. However, this cannot indicate that OpenMP has more programmability. MPI gives programmers more control of the processors, such as

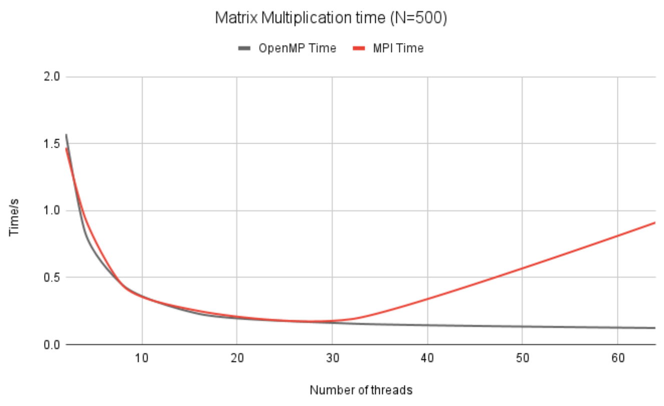

Fig. 6 Matrix Multiplication time (N=500): the x-axis is the number of parallel workers, the y-axis is the time it takes to complete the job, the matrix dimension is 500

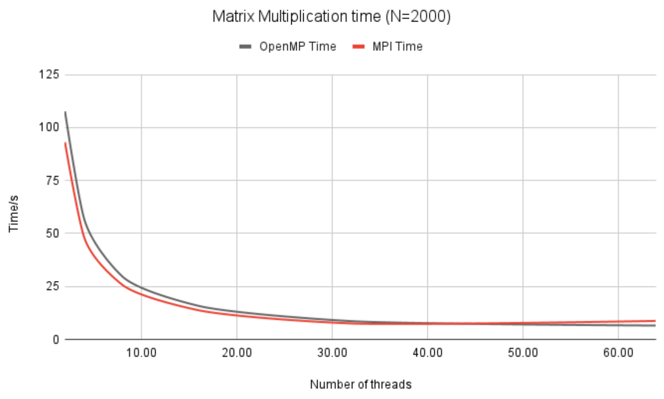

Fig. 7 Matrix Multiplication time (N=2000): the x-axis is the number of parallel workers, the y-axis is the time it takes to complete the job, the matrix dimension is 2000

explicit communication among processes, which means programmers have to write more codes to manage the processes. In contrast, OpenMP provides a series of runtime implementations of parallelization, which makes it easier but less flexible to program with.

6 Conclusion

Base on our benchmark results, both OpenMP and MPI demonstrate good scalability and performance improvements when increasing the number of threads/processes. However, OpenMP consistently outperforms MPI in terms of execution time, likely due to its lower overhead and more straightforward parallelization approach.

Table 7 the length of the benchmark programs compared with the length of parallel instructions

	OpenMP			MPI		
	Total	Extra	Percentage	Total	Extra	Percentage
matrix multiplication	56	1	1.8%	71	7	9.9%
Reduction	33	2	6%	56	8	14.3%

In conclusion, the memory system is the first thing to consider when we decide which parallel programming model to use. For distributed memory system, MPI is the best choice to accelerate computations. For shared memory system, OpenMP is more light-weight and easier to use. The problem size is also an important factor to consider. For scientific computations and deep learning, where data set cannot fit into a single shared memory, MPI should be the better choice. For other lighter computations, OpenMP is more efficient and can provide more speedup.

References

[1] Dubash, M.: Moore's law is dead, says gordon moore. Techworld. https://www.techworld.com/news/tech-innovation/moores-law-is-dead-says-gordon-moore-3571681/

[2] Openmp architecture review board (arb) members. https://www.openmp.org/about/members/

[3] Openmp fortran interpretations version 1.0. https://www.openmp.org/wp-content/uploads/finterp10.html

[4] DW, W.: Standards for Message-passing in a Distributed Memory Environment, (August 1992). Oak Ridge National Lab., TN (United States), Center for Research on Parallel Computing (CRPC). p. 25. OSTI 10170156. ORNL/TM-12147. Retrieved 2019-08-18. https://technicalreports.ornl.gov/1992/3445603661204.pdf

[5] Board, O.A.R.: Openmp application program interface, (2008). http://www.openmp.org/mp-documents/spec30.pdf

[6] Execution Model. https://www.openmp.org/spec-html/5.1/openmpse3.html

[7] J. Diaz, C. Muñoz-Caro and A. Niño, A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era, in IEEE Transactions on Parallel and Distributed Systems, vol. 23, no. 8, pp. 1369-1386, Aug. 2012, doi: 10.1109/TPDS.2011.308.

[8] S. J. Kang, S.Y.L., Lee., K.M.: Performance comparison of openmp, mpi, and mapreduce in practical problems. Advances in Multimedia (2015)

[9] Lorin Hochstein, V.R.B.: A preliminary empirical study to compare MPI and OpenMP. ISI-TR-676 (December 2011)

[10] Bova, S.W., Breshears, C.P., Gabb, H., Kuhn, B., Magro, B., Eigenmann, R., Gaertner, G., Salvini, S., Scott, H.: Parallel programming with message passing and directives (2001). https://doi.org/10.1109/5992.947105