Multicore-Project-Report

Abstract

In this project, we compare two parallel programming languages -- OpenMP and Pthreads. We would analyze the frameworks in four aspects: performance, scalability, running overhead and programmability.The experiments would be conducted using six benchmarks: Array accessing, linked list processing, matrix multiplication, LU matrix factorization, merge sort and ray tracing.

1. Introduction

Nowadays, multi-core hardware has been widely applied in high performance computation. With the development of parallel hardware technology, various programming models were developed to fit with different situations. Thus, programmers have a lot of choices of parallel programming languages and often get confused about which one to use.

C is one of the most popular programming languages and many developers use C to develop parallel applications. Therefore, we investigate OpenMP and POSIX threads (Pthreads) which are both shared memory multiprocessing programming API in C by linking libraries.

In this paper, we compare OpenMP and POSIX threads in their performance, scalability, running overhead and programmability. Based on the results of experiments, this paper offers some suggestions about how to choose between these two models.

2. Literature Review

There was a lot of papers and research on the comparison of parallel programming models. We found three papers that were useful [4,5,6]. The focus of these papers was to evaluate performance of some algorithm or program using several parallel programs. However, they did not focus on evaluating the other aspects, such as scalability and overhead of the APIs or not focus on comparing OpenMP against Pthreads. Therefore, we would evaluate the other aspects of OpenMP and Pthreads in this paper.

Searching for comparison of OpenMP against Pthreads, We found 2 papers focused on testing performance and programmability of some algorithms using Pthreads and OpenMP [11,12].

[11] compared Pthreads and OpenMP by paralleling a program called Genehunter. OpenMP and Pthreads achieve similar performance, but they demonstrate that OpenMP is the one that is easier to work with and easier to maintain the code.

[10] also compared Pthreads and OpenMP. They test the performance and programmability of matrix multiplication, Mandelbrot set, and Quick sort. They show that the performance of Quicksort was performing badly due to problems with recursion and parallel execution in OpenMP. However, task is a feature added to version 3.0 of OpenMP, which could deal with the recursion problem, so we tested whether omp task would make the performance better.

Many papers only work on a single aspect, especially the performance. However, programmers have different metrics to decide what is suitable for their applications. Therefore, this paper makes a more comprehensive comparison from several angles including performance, scalability, programmability and running overhead. So that it will offer a relatively complete review about the differences between OpenMP and Pthreads.

3. Proposed Idea

3.1 Performance

Performance is the ultimate concern for programmers because high performance computation is the main target of parallel programming.

In our project, we evaluate the performance by computing the speed-up of parallel programs compared with their sequential versions. To make the comparison valid, the parallel programs in OpenMP and Pthreads are designed with the exact same algorithm.

3.2 Scalability

Scalability refers to the ability of working efficiently with the number of threads growing. So, we evaluate the scalability of these two APIs by calculating the efficiency which is defined as the speed-up per thread.

The project uses the same benchmark programs as in the evaluation of performance. We fixed a relatively big problem size and increased the number of threads. By observing the trend of efficiency, we could learn about the scalability of them in our benchmark programs.

3.3 Running overhead

Parallel overheads arise from creating and destroying threads, synchronization, scheduling, load imbalance, memory access, etc. In this paper, we focus on measuring the overhead of creating threads, synchronization, and scheduling.

Creating a single thread can be a tiny job, so we created threads big enough to measure the time interval.

Overhead due to synchronization and loop scheduling is an important factor in determining the performance of shared memory parallel programs [7]. For a given parallel program, define Tp as the execution time of the parallel program, Ts as the execution time of the sequential version of the same program. To measure the overhead of a directive, the technique we used is to compare the time taken for a section of code executed sequentially, to the time taken for the same code executed in parallel enclosed in a given directive by using the formula Tp-Ts/p (the difference between the parallel execution time and the ideal time given perfect scaling of the sequential program) [7]. We also added some empty loops for the ease of measuring.

When it comes to synchronization, we investigated the overhead of *lock*, *atomic* and *critical*. Since Pthreads doesn't have the same schedule function as OpenMP, we manually assigned the work in the version of Pthreads and made a comparison between the two APIs. The schedule types we worked on contain both dynamic and static.

3.4 Programmability

Programmability is an important metric to programmers about working efficiently. Programming languages with ease of use will have better market acceptance. This paper gives a brief idea about programmability by comparing the lines of code or LOC and the number of characters NOC [3].

Before the statistics, we deleted the minor parts of the codes, such as comments, codes for verification and timer and blank lines, so that the comparison would be more precise.

3.5 Benchmarks Used

To evaluate these two programming models in different aspects, we designed several applications.

Array Access

Array access is an approach used to read and write access patterns to elements or portions of arrays.

Linked list processing

A linked list is a collection of data whose order is not given by their physical placement in memory. It is a data structure consisting of a collection of nodes, each node pointing to the next, which together represent a sequence. In this benchmark, we use OpenMP and Pthreads to traverse the linked list and evaluate them.

Ray tracing

In 3D computer graphics, ray tracing is a method that simulates the physical behavior of light. We cast a ray starting at the viewed point and find what would be reflected to the viewer. After

the ray determines which primitives intersect the ray, the color of that ray, and its corresponding pixel, must be computed. Based on the materials of the objects that are intersected, we can calculate the colors that they reflect or absorb, then we add them together for all the objects that the ray intersects. This is the basis of recursive ray tracing, where the usual ray tracing algorithm is applied to compute the reflection. This can cause further reflections, so we must limit the recursion depth to some constant (the number of light bounces). In this experiment, we set the number to 2, 4, 8, 16, 32, 64.

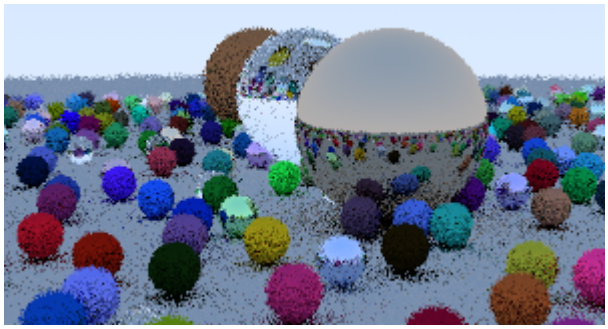


Figure. 3.5.1 Bounce depth 2

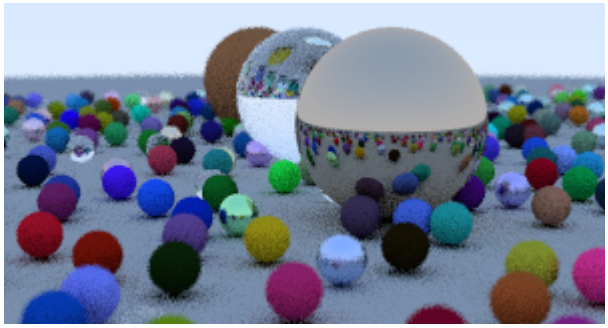


Figure. 3.5.2 Bounce depth 16

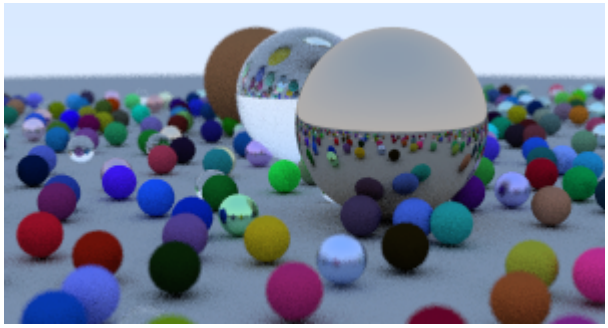


Figure. 3.5.3 Bounce depth 64

Matrix Multiplication

Matrix Multiplication is the process of multiplying two matrices together. The resulting matrix, known as the matrix product, has the number of rows of the first and the number of columns of the second matrix. In this benchmark we multiply two matrices with the same row and column.

LU Matrix Factorization

This application factors a matrix as the product of a lower triangular matrix and an upper triangular matrix. The algorithm is based on Gaussian elimination.

Merge sort

Merge sort is an efficient and general-purpose sorting Divide and Conquer algorithm. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves. In this benchmark, we used task directive to implement the OpenMP version of the application.

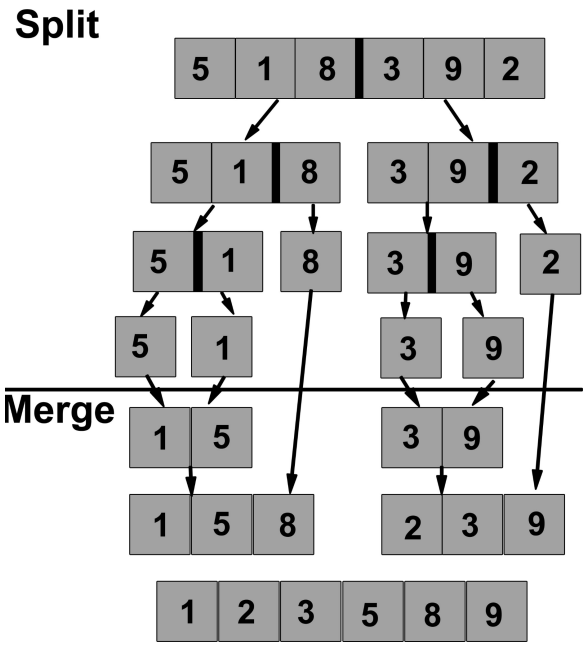


Figure. 3.5.4 merge sort

4. Experimental Setup

All the experiments in the project were done on the NYU computer server (crunchy6.cims.nyu.edu), which is used for CPU and memory intensive processes.
We designed some programs as the benchmarks for the comparison of the two programming models. Our benchmarks include array access, linked list processing, matrix multiplication, LU matrix factorization, merge sort and ray tracing.

We conducted each experiment several times and took the average results to avoid fluctuations.

5. Experiments & Analysis

We do experiments to compare OpenMp and Pthreads in 4 aspects: Performance, Scalability, Running Overhead and Programmability.

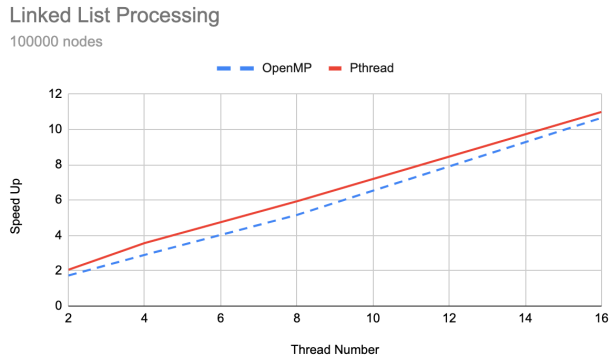
5.1 Performance

We measure speed up and execution time on these 6 benchmarks. For speed up, we fix the problem size and plot speed up as the number of threads increases. For execution time, we fix the number of threads and plot the execution time as the problem size increases.

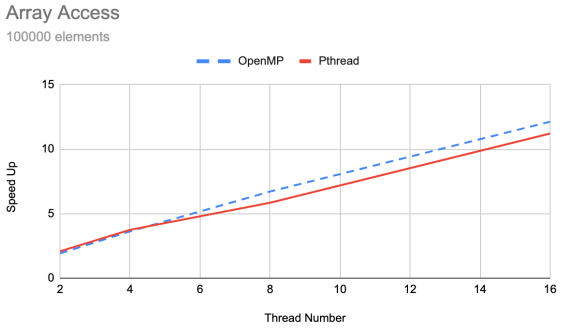
5.1.1 Speed Up

The specific fixed problem size is shown as below:

Benchmark	Fixed Problem Size
Linked List Processing	1000000 nodes
Array Accessing	1000000 elements
Merge Sort	50000000 elements
Matrix Multiplication	1600 * 1600
LU Factorization	2200 * 2200
Ray Tracing	16 per pixel



(a)



(b)

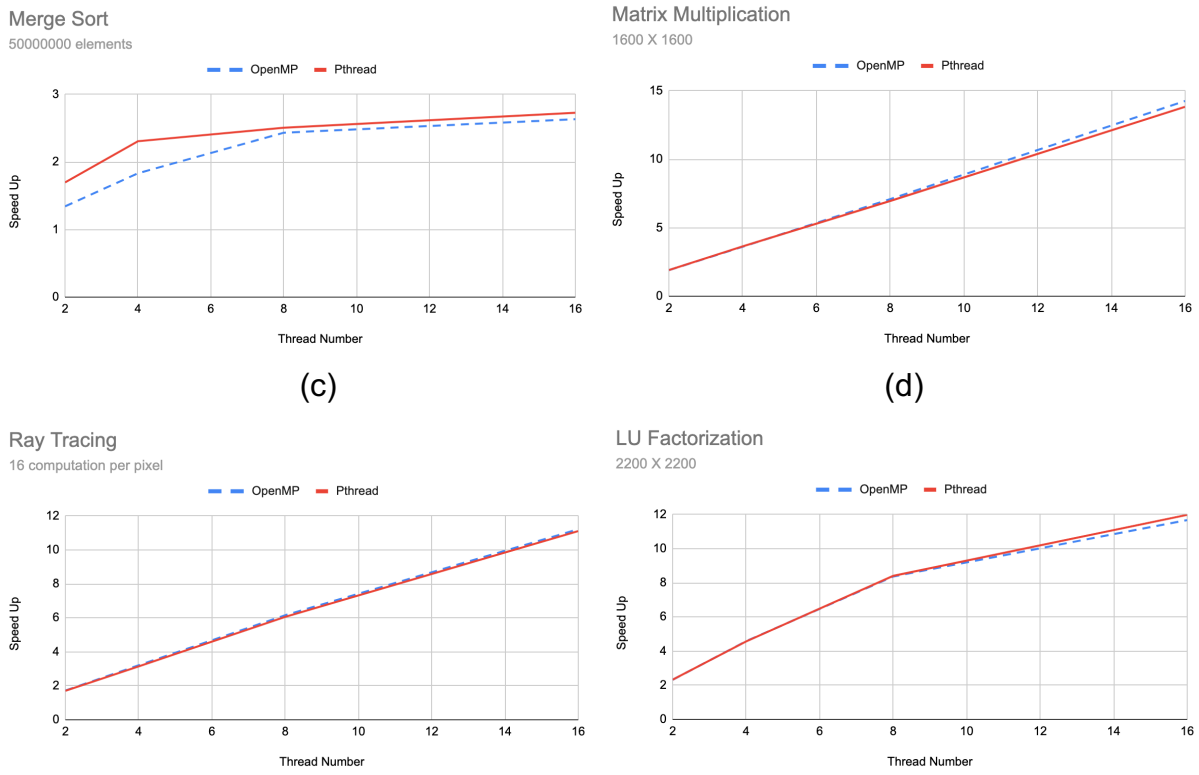


Figure. 5.1.1

As we can see from Figure. 5.1.1(a) and 1(c), Pthreads has a better speedup on linked list processing and merge sort. On linked list processing, the difference between two speedup effects are almost fixed(15%) as the number of threads grows. However, on merge sort, Pthreads' speedup effect is up to 30% more than OpenMP when using less than 6 threads, but it becomes very close to the other when using more and more threads. From(b) and (d), we can know that OpenMP has a better speedup on array accessing and matrix multiplication. As for array accessing, there is almost no difference between them when using less than 8 threads, but after that, OpenMP's speedup is 16% more than Pthreads' speedup. As for matrix multiplication, they are pretty similar until using 14 or more threads, OpenMP's speedup is 3% more(larger as the number of threads grows). Finally, from(e) and (f), we can conclude that their speedup effects are very close to ray tracing and LU Factorization. Besides, their speedup both increases as the number of threads increases on all benchmarks.

5.1.2 Execution time

We use 8 threads to run all the benchmarks. As for merge sort, we use both 2 threads and 8 threads to run.

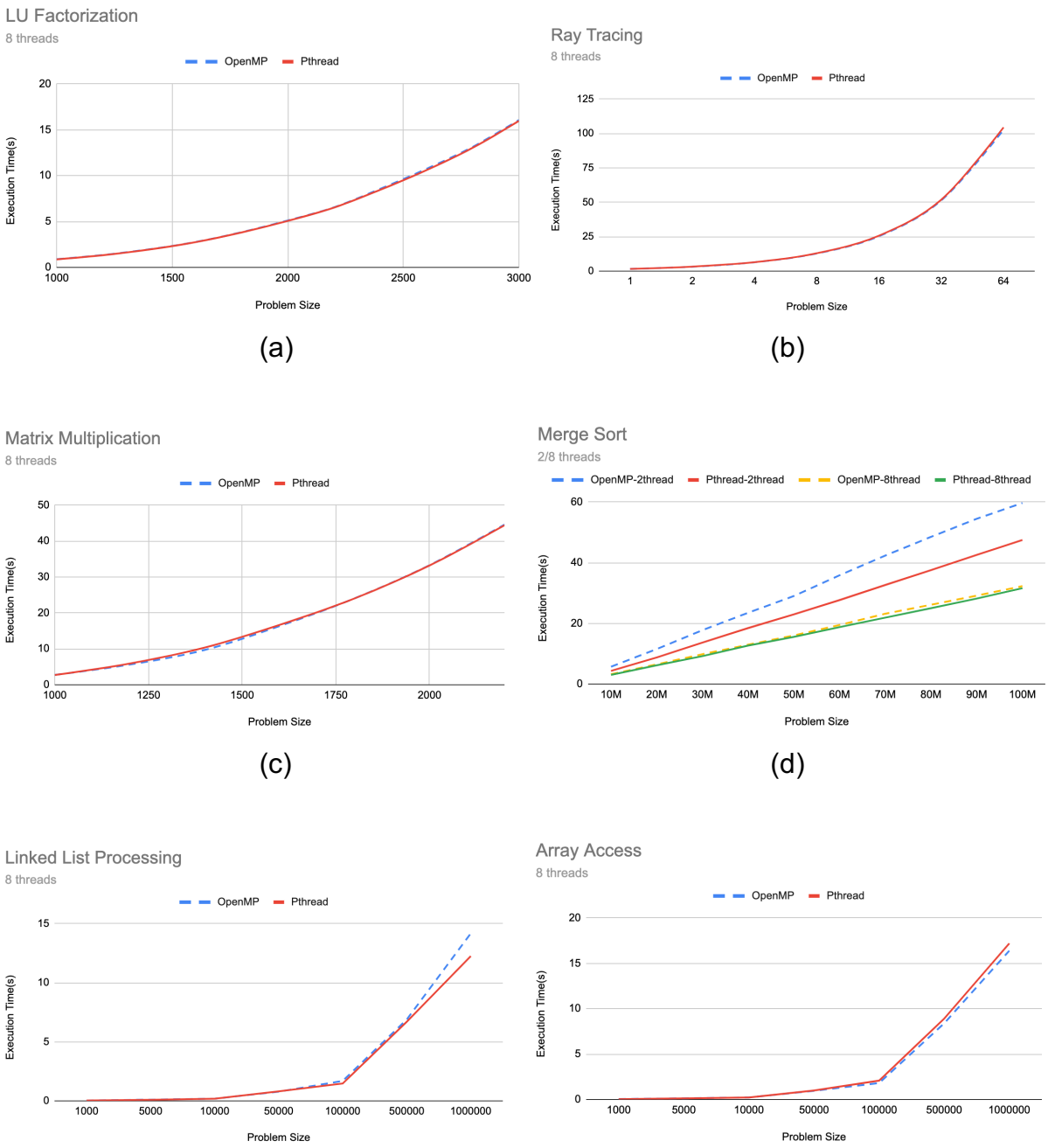


Figure. 5.1.2

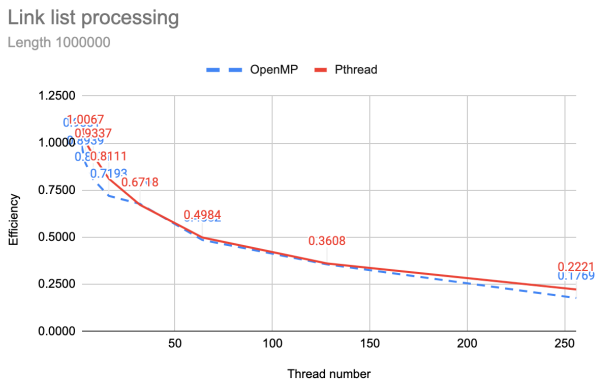
From Figure. 5.1.2(a), (b) and (c), we find OpenMP's and Pthreads' execution time is pretty close on LU factorization, ray tracing and matrix multiplication. From(e) and (f), we can see their execution time is similar when the problem size is small. However, when the problem size is big enough, Pthreads runs faster on linked list processing, OpenMP runs faster on array accessing. As the linked list's problem size continues to increase, the difference between these two becomes larger and larger. But when the array's size continues to increase, the difference is

fixed to around 0.01s. Additionally, as we can see from Figure. 2(d), when the problem size increases, the gap between two times also increases. However, it's no longer the case when using 8 or more threads.

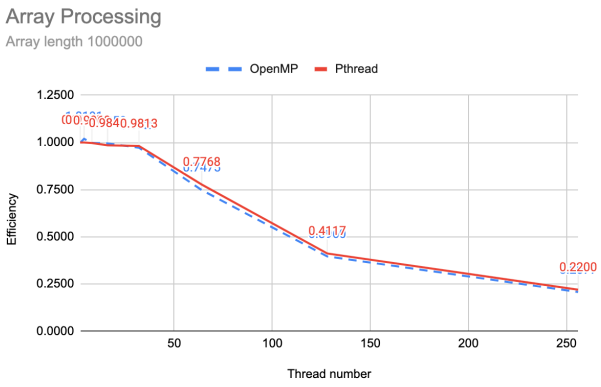
5.2 Scalability

We fix the largest problem size and use larger numbers of threads to test the scalability of OpenMP and Pthread on six benchmarks. The specific fixed problem size is shown as below:

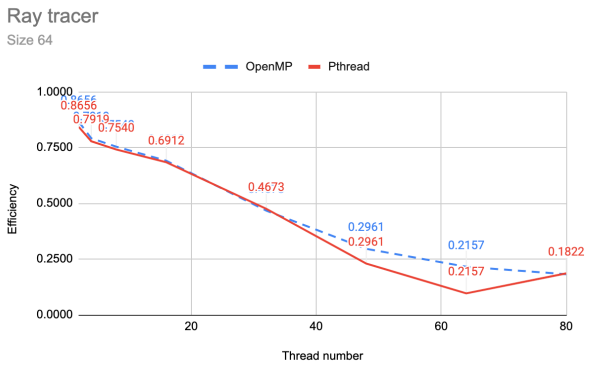
Benchmark	Fixed Problem Size
Linked List Processing	10000000 nodes
Array Accessing	10000000 elements
Merge Sort	100000000 elements
Matrix Multiplication	2200 * 2200
LU Factorization	3000 * 3000
Ray Tracing	64 per pixel

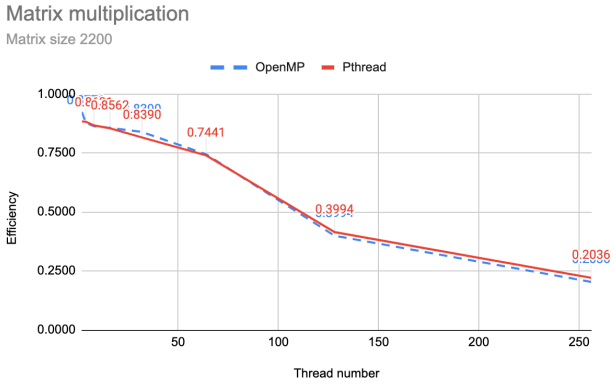


(a)

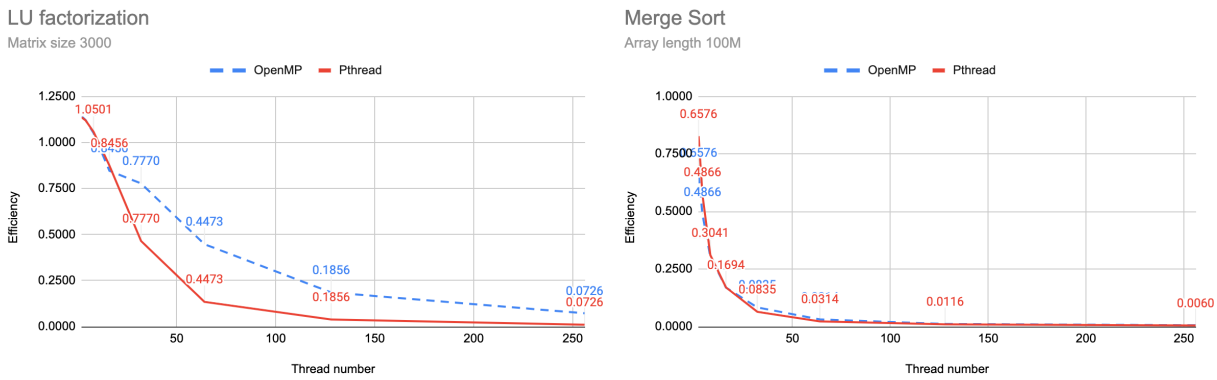


(b)





(c) (d)



(e) (f)

Figure. 5.2

From Figure. 5.2, we conclude that in most of our benchmarks, the scalability of OpenMP and Pthreads is similar. In Ray tracer and LU factorization, it seems that OpenMP has a bit better efficiency with the number of threads increasing.

Both of their scalability is relatively weak since the efficiency always decreases significantly when the number of threads is huge. The efficiency of LU factorization falls faster than other benchmarks may be due to they have more complicated synchronization requirements and the reason for Merge sort may be that it needs many creations and destructions of temporary arrays in recursion.

5.3 Running Overhead

We do experiments on 3 factors of the running overhead: create & destroy threads, scheduling and synchronization.

5.3.1 Create & Destroy Threads

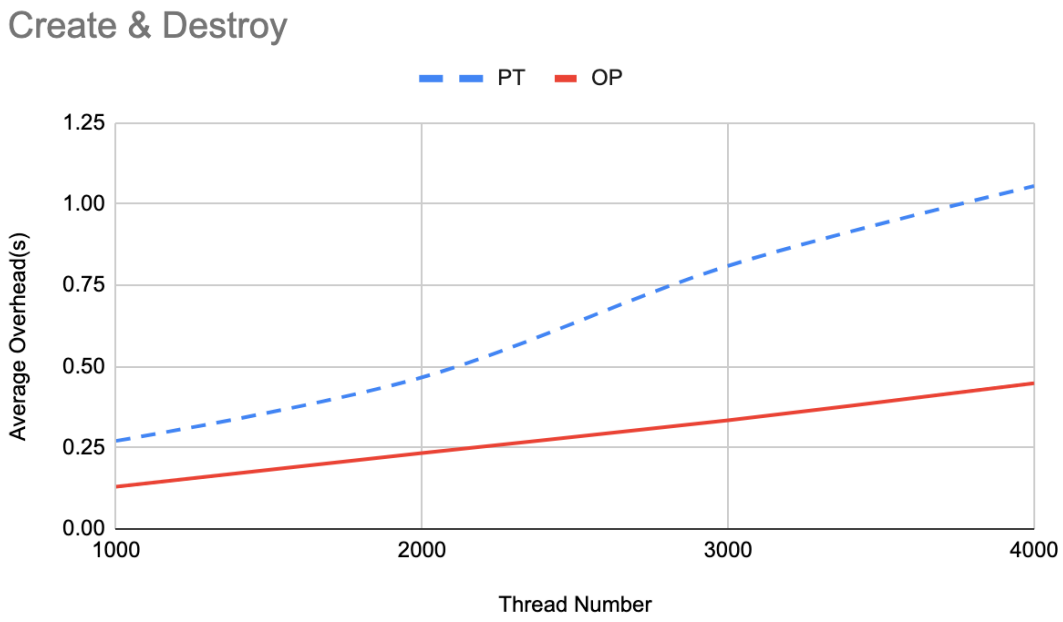


Figure. 5.3.1

As we can see from Figure. 5.3.1, Pthreads has a much larger creating & destroying overhead than OpenMP. Pthreads overhead is more than 2 times OpenMP's overhead when the number of threads reaches 3000 or more.

5.3.2 Scheduling

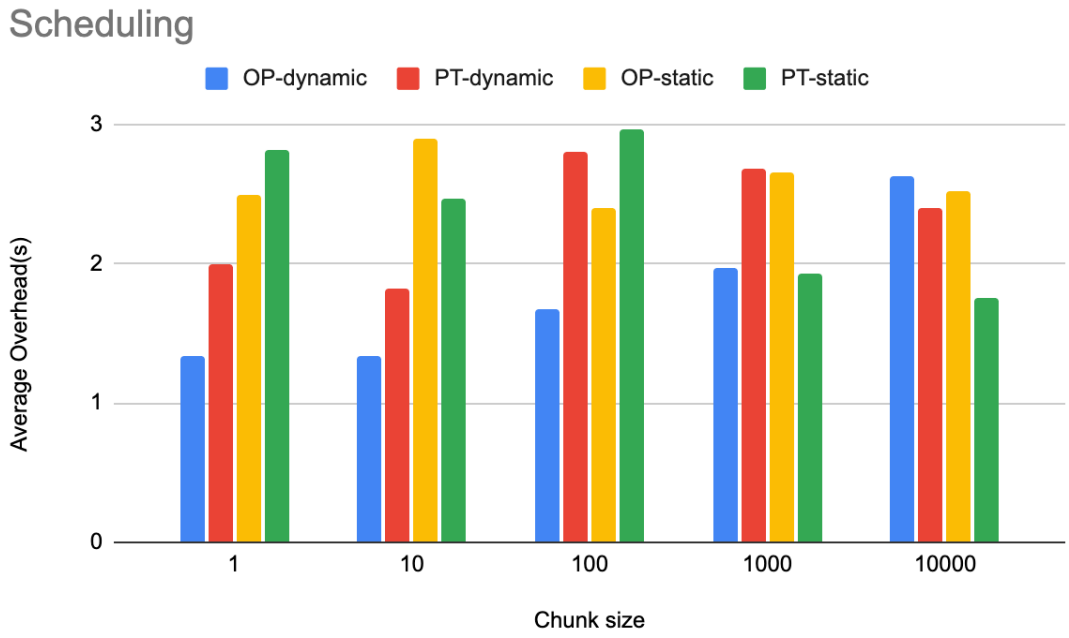


Figure. 5.3.2

From Figure. 5.3.2, as for dynamic overhead, Pthreads is 30%-40% larger than OpenMP when the chunk size is less than 10000. When the chunk size is too large, OpenMP's overhead is larger. As for static overhead, OpenMP's overhead is generally larger. But when the chunk size is 100, Pthreads is larger.

5.3.3 Synchronizing

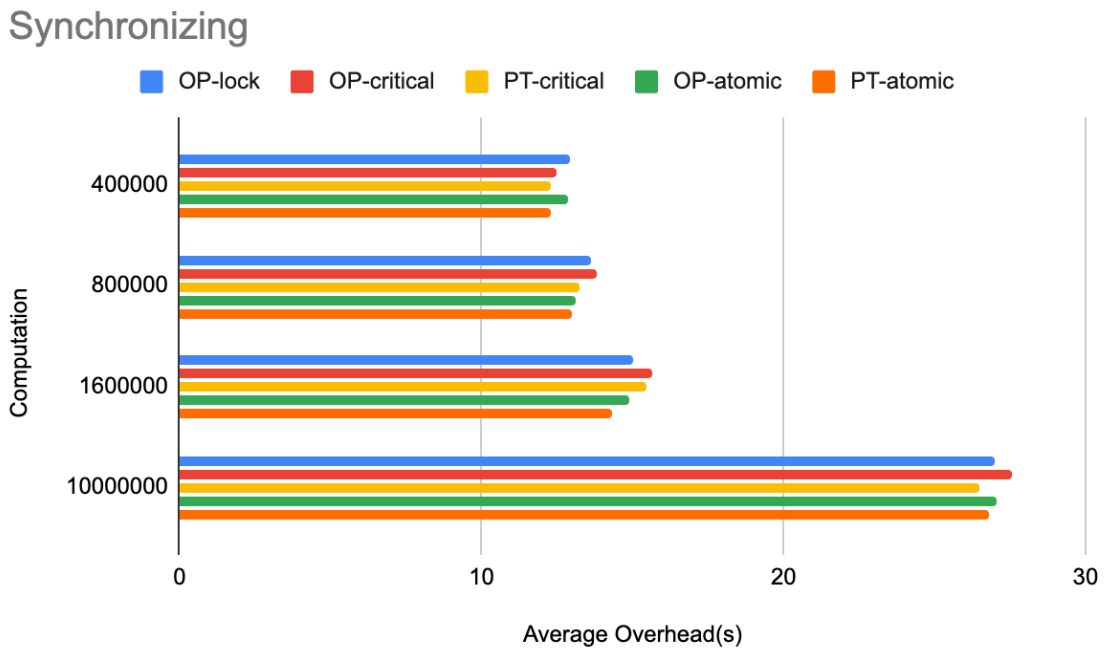


Figure. 5.3.3

From Figure. 5.3.3, the synchronizing overhead becomes larger as computation increases. And Atomic has the smallest overhead from these locking types, critical has the largest overhead. That's because atomic doesn't use an explicit lock, it's much lighter than critical. As for OpenMP and Pthreads, their overheads are very similar.

6. Programmability

Table 1 and 2 show the LOC (lines of codes) and NOC (number of characters) for our benchmarks for both OpenMP and Pthreads. Evidently, OpenMP is much simpler than Pthreads in programmability. Basically, programmers need to put in 120% effort on the OpenMP version than the sequential code and more than 200% effort on the Pthreads version.

It is because in OpenMP programmers can use a few lines of annotations to parallelize the sequential code, manage synchronization, scheduling, data access, etc. While in Pthreads, programmers must design functions for parallelization and declare threads and arguments variables manually.

Consequently, OpenMP is a more developer-friendly API compared with Pthreads. However, Pthreads would be a better choice if developers hope to customize more details of parallelization.

		SEQ	OpenMP	OpenMP Effort(%)	Pthreads	Pthreads Effort(%)
Array access	#LOC	37	43	16.22	88	137.84
	#NOC	543	640	17.86	1515	179.01
Linked list processing	#LOC	41	67	63.41	69	68.29
	#NOC	451	851	88.69	1157	156.54
LU matrix factorization	#LOC	45	50	11.11	90	100.00
	#NOC	806	939	16.50	1585	96.65
Matrix multiplicatio n	#LOC	45	53	17.78	96	113.33
	#NOC	973	1150	18.19	1971	102.57
Merge sort	#LOC	75	83	10.67	180	140.00
	#NOC	965	1196	23.94	2844	194.72

Table.1

		SEQ	Pthreads	Pthreads Effort(%)
Array access	#LOC	37	88	137.84
	#NOC	543	1515	179.01
Linked list processing	#LOC	41	69	68.29
	#NOC	451	1157	156.54
LU matrix	#LOC	45	90	100.00

factorization	#NOC	806	1585	96.65
Matrix multiplication	#LOC	45	96	113.33
	#NOC	973	1971	102.57
Merge sort	#LOC	75	180	140.00
	#NOC	965	2844	194.72

Table.2

7. Conclusions

Based on all the experiments above, we arrive at the conclusion that OpenMP is close to Pthreads in scalability in general, but OpenMP performs slightly better when the number of threads is bigger than 16.

As for performance, Pthreads is slightly better at processing linked lists, while they achieve similar performance at accessing arrays. But in general, they achieve similar performance in many of our applications. In recursion problems like merge sort, Pthreads performs better when we run the program with 2 threads. However, as the thread number increases, like to 8 threads, they achieve similar performance.

In overhead, Pthreads has more overhead in creating and destroying threads than OpenMP and the overhead of synchronization and scheduling is close.

In programmability, OpenMP is more friendly to developers because OpenMP uses simple annotations for parallelization while developers must declare variables and start-routine functions for executing threads in Pthreads. However, Pthreads give us more flexibility to assign the threads.

In this project, we made great efforts to refine our codes of the Pthreads version to make its implementation as similar as to the OpenMP version. Besides, we also designed the scheduling method and linked the other atomic library in the Pthreads version to make a fair comparison with OpenMP.

In our future work, we will explore some other methods to get more precise data for comparison. We can also continue to refine and supplement our benchmarks for more general results.

References

1. Göran Angelo Kaldéren and Anton From, Thesis title: A comparative analysis between parallel models in C/C++ and C#/Java : A quantitative comparison between different programming models on how they implement parallelism, 2013 KTH Information and Communication Technology

2. Konstantina Panagiotopoulou, Thesis title: Assessing Modern Parallel Programming Languages, 2013 Computer Science School of Mathematical and Computer Sciences

3. F. Cantonnet, Y. Yao, M. Zahran, T. El-Ghazawi, Productivity analysis of the UPC language, 2004 IEEE

4. Solmaz Salehian, Jiawen Liu, Yonghong Yan, Comparison of Threading Programming Models, 2017 IEEE

5. Alexandros Stamatakis, Michael Ott, Exploiting Fine-Grained Parallelism in the Phylogenetic Likelihood Function with MPI, Pthreads, and OpenMP: A Performance Study, 2008 SpringerLink

6. Panagiotis D.Michailidis, Konstantinos G.Margaritis, Scientific computations on multi-core systems using different programming frameworks, 2016 ScienceDirect

7. J. M. Bull, Measuring Synchronisation and Scheduling Overheads in OpenMP, Proceedings of the First European Workshop on OpenMP, Lund, Sweden, 1999, pp 99–105

8. J. M. Bull and D. O'Neill, A microbenchmark suite for OpenMP 2.0, SIGARCH Comput. Archit. News, vol. 29, no. 5, pp. 41–48, 2001

9. J. M. Bull, F. Reid and N. McDonnell, A microbenchmark suite for OpenMP tasks, in Proceedings of the 8th international conference on OpenMP in a Heterogeneous World (IWOMP '12) pp. 271-274, 2012

10. Henrik Swahn, Pthreads and OpenMP A performance and productivity study, 2016 Blekinge Institute of Technology

11. Bob Kuhn, Paul Petersen, OpenMP versus Threading in C/C++, 2000 dblp

12. Wei Zhong, Gulsah Altun, Xinmin Tian, Robert Harrison, Phang C. Tai, Yi Pan, Parallel protein secondary structure prediction schemes using Pthread and OpenMP over hyper-threading technology, 2007 google scholar