Analyzing the limitations of parallelism in hardware and software through threaded programming

Chenyi Wang*

Bradley Department of Electrical and Computer Engineering, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, 24061, United States

*Corresponding Author Email: chenyiw@vt.edu

Abstract. Parallel computing is applied across various technology industries for solving complex scientific problems that desire high performance and efficiency while can help utilize the resource to save time, money, and energy. This technique has been developed over decades as the physical limits of chips have been reached. The development of parallel computing is closely bonded with the evolution of hardware and software implementation, and there are many forms of parallelism applied to different systems to improve their performance. This paper focused on threaded programming by applying parallelism to a decision tree program and testing the program performance with variables of thread number and device configuration while aiming to demonstrate the limitation of parallelism from both software and hardware aspects. The experiment's outcome in this paper shows that the overall trend of the performance variation for different devices is similar. Still, an improvement in the performance will be affected by the hardware implementation. In contrast, a device with a better configuration performs better. To summarize this research, parallelism can only be fully utilized when applied to a properly designed program running on a robust machine. Therefore, the innovation of hardware and software is essential to the future development of parallel computing.

Keywords: technology industries, high performance, decision tree, Thread programming

1. Introduction

Parallel computing is a practical technique widely applied in various modern technology industries to solve significant scientific problems requiring high performance. As programming becomes more sophisticated and computationally intensive, this technique improves to fulfill the demands of high efficiency and performance. It allows significant problems to be broken down into smaller pieces and solved simultaneously. It has been continuously developed and upgraded on many different levels over the decades as hardware implementations, such as the design of processors, have evolved and software techniques and applications have developed.

The intuition behind parallel computing is that the utilization of all resources can be maximized. Compared to serial computing, it makes better use of hardware and can solve more complex problems. The use of parallel computing saves time, money, and energy as many resources working together cuts potential costs [1]. It is powerful as it models the real world to let all the events happen together. Still, absolute parallelism is not a natural way of thinking for humans because it is difficult to do multiple tasks simultaneously if all require the same attention [2]. To implement parallelism, it needs some effort from the developer to identify and expose the concurrency in an application while properly leveraging the resources to execute these operations concurrently [3].

To increase the performance and efficiency of the programs, the first approach is to make some evolution at the hardware level. The key components of computers are processor chips, which consist of transistors. The performance and complexity of a processor chip can be roughly estimated by its number of transistors [4]. In addition to increasing the number of transistors, clocking the chip at a higher frequency is another way to speed up the execution of the computer so that the growing market demand for high performance can be better satisfied [5]. This trend was followed for decades till the processor frequency reached a plateau due to the physical limitation. To continue improving system performance, parallel computing has become an essential technique. The history of the development of parallel computing can be traced back to the 1950s. As recorded, IBM researchers John Cocke and Daniel Slotnick discussed using parallelism in numerical calculations for the first time in 1958 [6].

Through the ongoing efforts of numerous researchers over the decades, parallel computing has become more and more mature and widely used nowadays. Along with this development, computer architectures have upgraded continuously to speed up a program by increasing the number of cores. In the past decade, the graphics processing unit (GPU) has become an essential part of the high-performance computing field due to its robust parallel processing power and low cost [7].

There are many methods to realize parallelism. One of the approaches is multicore processors, which partition all transistors on the chip into groups to build multiple, independent processor cores so that the communication of transistors is within the group. A shorter switching time and latency can be maintained [8]. Another way to reach parallelism is to implement multithreaded architectures. This implementation has been applied across the entire computing devices range. Multithreading helps utilize the computational resources of a processor core to be more effective, as a stall in one thread will not result in idle execution resources [9]. On top of these approaches, Intel® has developed a technology called hyper-threading, which is a hardware innovation that let the central processing unit (CPU) expose two execution contexts on each physical core so that each core can work to resemble two "logical cores" and can process different software threads [10]. In addition to the steady increment of performance brought from the parallel hardware, software application development is indispensable to approach an expected growth in application performance and application efficiency [11]. The first step to designing a parallel program is identifying the problem's parallelism. Some popular ways of applying parallelism are functional parallelism, automatic extraction, data parallelism, and request parallelism, which are applicable to different implementations. Overall, the basic concept of a parallel algorithm is to start a single process and fork multiple threads, then execute these tasks simultaneously, and eventually merge the results to obtain the final resolution.

The research in this paper is focused on parallel programming using threads. A thread is an independent collection of sequentially executed programming steps that can help improve the efficiency and performance of a program. This paper will demonstrate the application of multiple threads in a program while presenting the potential limitations of parallelism from both hardware and program implementation aspects.

In this paper, an experiment is created to compare the performance of a parallel implementation in different conditions. This experiment uses a basic chess engine as the execution program. This program is implemented using the minimax tree. At the same time, the parallelism is applied to the first layer under the root node so that the entire tree will be divided into multiple subtrees by the indicated number of threads. The number of lines is the first variable in the test. A computer will run the chess engine program with thread numbers from 2^0 to 2^5 separately and then compare the difference in their performance. The second variable in this research is the hardware configuration. The same test will be repeated on different devices with different configurations, and their results will be compared for further analysis.

The rest of this paper is organized as follows. Section 2 details the model used to implement parallelism and the method used to test the program and variables set for later comparison. Section 3 displays and discusses the results. Section 4 concludes the outcomes and proposes suggestions and prospects for future research.

2. Methods

2.1. Details of the execution program

To visualize the impact of parallelism on a system, the most straightforward approach is to implement it in a program with different constraints and compare the outcomes. The program used in this experiment needs to have a moderate complexity. That is, it cannot be so simple that applying parallelism does not achieve the expected optimization. It cannot be so complicated that it overshadows the importance of testing the parallelism performance in this research. The decision tree is one of the most appropriate models to implement and test parallelism. It has many branches and must be implemented recursively to search for results. Because of the natural behavior of the decision

tree, it is intuitive to apply parallelism as the entire tree can be divided into multiple subtrees and do the computations simultaneously and then merge the results of each subtree back into the top-level node to get the final result as desired.

The execution program with the decision tree algorithm chosen to be used in the experiment of this paper is a basic chess engine because it is simple to implement and easy to understand the computing procedure and the results. This engine works to analyze each move in a chess-playing round and provides the best intelligent next move in a relatively short period. The parallelism is applied to the decision tree implementation using OpenMP. The entire decision tree is divided by the OpenMP instruction at the first layer under the root node into multiple subtrees based on the specified number of threads. Then each thread is executed concurrently to do the calculation for each subtree. After the operation in each thread is finished, their respective results are merged back into the root node to generate the outcome, which is the best next move for the current state in this program.

3. Variable description

Thread number. To observe the influence of parallelism on a program, the first variable set in the test for this experiment is the number of threads. Generally, increases in threads will improve the overall performance; therefore, to test whether this theory is correct in all cases, the program will be run multiple times with a different number of threads. By comparing the execution time of the program with other thread counts, the improvement of the performance with parallelism will be more intuitive.

Device configuration. The second variable set for this research is the device configuration because it can influence the performance of a program to some extent. To observe the effects brought by the hardware setting, several devices with different configurations are used in this test to run the same program in the same manner so that their respective outcomes can be compared objectively.

3.1. Results assumption

For the test based on the number of threads, theoretically, the program will be executed more efficiently with more threads because the task is split into multiple smaller parts. In contrast, each part will be computed much faster to reduce the overall operating time. Therefore, as the number of threads grows, the expected trend of time variation is continuously declining.

For the test based on the device configurations, unless a device is incompatible with running the program, the overall trend of the time variation versus the incrementation of the thread number is expected to be similar. However, considering the impact of different performance configurations on program operation, the time cost for each number of threads is expected to be different.

4. Experimental results and analysis

4.1. Data collection

Several sets of data need to be collected for this experiment better to observe the differences in the results of different variables and to do subsequent analysis.

To improve the collection efficiency while ensuring the comparability and coverage of the collected data, the number of threads will be increased from 2^0 to 2^5, which is only six sets of data. The number of threads stops incrementing at 32 because this program's parallelism is applied to the first layer under the root node of the decision tree. The number of threads cannot exceed the number of branches in this layer. By testing the chess engine in general, the maximum number of branches is 37; therefore, the test on increasing the number of threads stops at 32. For each number of threads, the device will run the program in the same manner 20 times to obtain the average cost time because the running time fluctuates sometimes, and to acquire a more objective result, running the program multiple times is a better option.

For the variable of different devices, each device will run the program, in the same manner, with a different number of threads so that the only variable in this test is the device configurations. This test is a control experiment conducted on six other devices, which are Intel® i5-8250U, Intel® i7-10700, Intel® i9-9880H, Intel® Xeon® with eight cores, Intel® Xeon® with 12 cores, and AMD Ryzen 7. Each device chosen has a different CPU performance, which provides an excellent comparison of the impact of devices on the execution speedup. After the test is completed, a similar form of data sets will be obtained for each device.

4.2. Results and Discussion

By testing the program with different numbers of threads on an Intel® i5 machine, it is obvious that the overall performance of the program is improved with the increment of thread number as expected with the decrease of the execution time (Table 1).

Table 1. Performance comparison with different numbers of threads

Thread counts (N)	**1**	**2**	**4**
Time(s)	5.713	3.690	2.480
Thread counts (N)	**8**	**16**	**32**
Time(s)	1.928	1.713	1.708

When the thread increases from 1 to 4, the time cost is reduced dramatically; however, after the thread number reaches 8, the execution time reduction is not considerable, which indicates that the thread increment to 8 is the threshold performance of the parallelism applied to this algorithm.

This phenomenon may be caused by the hardware configuration of the testing machine. The Intel® i5 device in this test has four cores and eight logical processors. After the thread number exceeds the number of logical processors, those different threads are virtual threads created programmatically and do not perform as well as the threads that come with the device itself. The same test procedure was carried out on all six machines, and the results are shown in Figure 1.

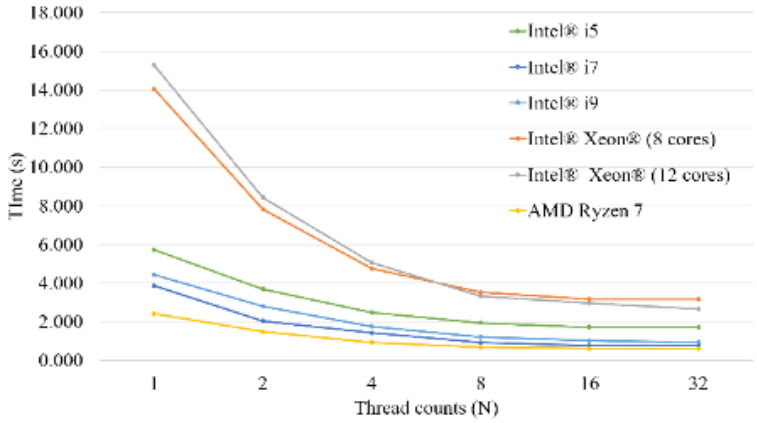


Figure 1. Performance comparison of different devices with different numbers

The overall trend of the execution time variation for all six devices is similar, as expected, which is declining as the number of threads grows. It is also apparent that none of the two devices have the same performance; the time cost is different in general, and this outcome also matches the expectation.

Looking more closely at the resulting graph, the overall performance trend of two Intel® Xeon® machines is slightly different from the other four devices, as they seem to cost more time during the execution. Still, the improvement corresponding to the thread increment is more dramatic. This is caused by the implementation differences of the Intel® Xeon® processors, as they support higher memory speeds and capacity and are designed to accelerate the workload. They avoid the time cost of data reloading to increase operational efficiency by retaining data in persistent memory during restart [12]. This design helps reduce the amount of time the thread takes to fetch the data, while consequently resulting in a better improvement of the overall performance as the thread number rises.

In addition to the distinction in the general performance trend, different machines result in different cost times. A more detailed comparison indicates that the higher the device configuration, the faster it runs the program. Since only one program of relatively low complexity, was used in this experiment, the differences in terms of runtime speedups between individual devices due to configuration differences were not reflected a great extent. Therefore, in future research, several more different models of varying complexity can be used, while conducting more tests and adding more threads to explore further potential limitations.

5. Conclusion

This paper presented research on program execution performance comparison with the parallelism implementation with threads under different conditions, aiming to identify the limits of parallelism application from hardware and software aspects.

The outcomes of the experiment in this paper demonstrate that the improvement of a program's performance has limitations that depend on both executing program and machine hardware implementation. The configuration of the running device also impacts the program's performance to different extents. By analyzing the results, the performance of a program is closely related to how it is implemented and where it is being executed. The program's implementation can directly decide the limitation of its performance with sufficient support from the compatible device, which can be easily adjusted by changing the algorithms. However, the limitation from the hardware perspective is more significant and harder to adjust. As the results are shown in this paper, the program performance improvement will be affected by the physical implementation of the devices. In addition, devices with higher configurations will consistently achieve better overall performance than devices with lower configurations under the same conditions.

To conclude the results of this research, software development and hardware evolution are equally crucial to implementing parallelism. Parallel computing is a powerful technique that can help improve the performance and efficiency of a program to a great extent while saving time, money, and energy. The technique has been developed for many decades, along with the innovation of computer architecture. However, the power of parallelism will not be utilized entirely on a robust machine without a carefully designed parallel program, and vice versa. Therefore, it is crucial to focus on both software and hardware aspects when making progress for parallel computing.

In future work, more models with different levels of complexity will be used as a variable to test the limits of parallelism from the software perspective. More sophisticated programs can be used to apply parallelism so that the performance improvement will be more significant, and further tight relationships between software and hardware when applying parallelism can be potentially discovered.

References

[1] “Introduction to Parallel Computing.” GeeksforGeeks, June 4, 2021. Retrieved August 21, 2022, from https://www.geeksforgeeks.org/introduction-to-parallel-computing/

[2] Kriha, Walter. “Why Is Parallel Programming so Hard to Express?” Computer Science Blog, October 24, 2016. Retrieved August 21, 2022, from https://blog.mi.hdm-stuttgart.de/index.php/2016/10/24/why-is-parallel-programming-so-hard-to-express/

[3] Robey, Robert, and Yuliana Zamora. “Chapter 1 Why Parallel Computing?” Essay. In Parallel and High Performance Computing. Manning, 2021.

[4] Rauber, Thomas, and Rünger Gudula. “2.1 Processor Architecture and Technology Trends.” Essay. In Parallel Programming for Multicore and Cluster Systems, p.7–10. Springer, 2013.

[5] Venu, Balaji. "Multi-core processors-an overview." arXiv preprint arXiv:1110.3535 (2011).

[6] Wilson, Gregory V. “The History of the Development of Parallel Computing,” October 28, 1994. Retrieved August 21, 2022, from https://webdocs.cs.ualberta.ca/~paullu/C681/parallel.timeline.html

[7] Navarro, Cristobal A., Nancy Hitschfeld-Kahler, and Luis Mateu. "A survey on parallel computing and its applications in data-parallel problems using GPU architectures." Communications in Computational Physics 15, no. 2 (2014): 285-329.

[8] Wang, Lizhe, Jie Tao, Gregor von Laszewski, and Holger Marten. "Multicores in Cloud Computing: Research Challenges for Applications." J. Comput. 5, no. 6 (2010): 958-964.

[9] Nemirovsky, Mario, and Dean M. Tullsen. "Multithreading architecture." Synthesis Lectures on Computer Architecture 8, no. 1 (2013): 1-109.

[10] "What Is Hyper-Threading?" Intel. Retrieved September 17, 2022, from https://www.intel.com/content/www/us/en/gaming/resources/hyper-threading.html.

[11] Pacheco, Peter. "2.4 Parallel Software" An introduction to parallel programming, p.47. Elsevier, 2011.

[12] "Intel® Optane™ Persistent Memory 200 Series Brief." Intel. Retrieved September 17, 2022, from https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-persistent-memory-200-series-brief.pdf.