ANALYSIS OF PARELLEL MULTIPROCESSOR EXECUTION OF DIFFERENT PROGRAMMING APPROACHES

Leonid Djinevski
University of Skopje, Skopje, North Macedonia, leonid.djinevski@utms.edu.mk

Abstract: The necessity of parallel programming for leveraging the full potential of modern multicore processors cannot be overstated. One of the fundamental algorithms used extensively across a multitude of fields is the array sum reduction algorithm. This algorithm is pivotal in scenarios requiring the aggregation of array elements, a task that can benefit significantly from parallel processing techniques. Implementing the array sum reduction algorithm on a multiprocessor system can be approached using various parallel programming models.
This paper delves into a comparative analysis of two prominent parallel programming approaches for the array sum reduction algorithm on a shared memory microprocessor environment. We explore the implementation and performance characteristics of C with OpenMP, a well-established and widely adopted parallel programming model in the C programming ecosystem, and C# with Parallel.For, a robust and flexible parallel programming framework within the .NET environment.
Our analysis involves a rigorous empirical evaluation and benchmarking process to assess the execution efficiency, scalability, and ease of implementation associated with each approach. By systematically examining these factors, we aim to elucidate the specific contexts and application scenarios where each parallel programming paradigm excels or may present limitations. Furthermore, we consider the development overhead and learning curve associated with adopting these technologies, providing a holistic view of their practicality in real-world applications.
In addition to performance metrics, we examine the debugging and maintenance complexities that developers may encounter when working with these paradigms. We also assess the extent to which each approach supports advanced parallelism features such as task parallelism and load balancing. Another important aspect of our study is the interoperability of these programming models with other tools and libraries commonly used in high-performance computing.
Through this comparative study, we seek to provide developers and researchers with a comprehensive understanding of the trade-offs involved in selecting between C with OpenMP and C# with Parallel.For. The insights garnered from our evaluation will aid in making informed decisions, ensuring optimal performance and resource utilization in parallel programming applications tailored to modern multicore processors. Ultimately, our findings contribute to the broader discourse on parallel computing best practices and the future directions of parallel algorithm development.
Our results aim to serve as a reference point for future research, highlighting potential areas for enhancement in both programming models. We hope to inspire further innovation in parallel computing techniques, fostering advancements that can lead to more efficient and scalable software solutions.
Keywords: OpenMP, Parallel.For, multiprocessor.

1. INTRODUCTION

Parallel processing is a method of executing multiple tasks concurrently, enabling more efficient use of computational resources and significantly improving the performance of computationally intensive programs. In the realm of programming, leveraging parallelism has become increasingly essential, particularly with the advent of multi-core processors that offer the hardware capabilities necessary to execute parallel tasks effectively. As such, parallel programming (Chandra, 2001) has become a critical skill for developers aiming to optimize the performance of their applications.
OpenMP (Bak et al., 2022) and Parallel.For (Ciccozzi et al., 2022) are two widely used tools for parallel programming in C (Aho, Kernighan, and Weinberger, 2023) and C# (Griffiths, 2022), respectively. OpenMP provides a simple and flexible interface for implementing parallelism in C/C++ programs (Zhu et al., 2015). It allows developers to add parallelism to existing code with minimal changes, making it a popular choice in scientific and engineering applications where performance is paramount. On the other hand, Parallel.For (Okur and Danny, 2012) is part of the Task Parallel Library in .NET (Chi et al., 2021), designed to facilitate easy implementation of parallel loops in C#. This framework is highly intuitive for developers familiar with the .NET ecosystem and offers seamless integration with other .NET features, making it ideal for enterprise and desktop applications.
This paper will explore these frameworks in detail and provide comparative insights to help developers choose the most suitable approach for their specific needs. By presenting a comprehensive analysis, we aim to highlight the strengths and weaknesses of each tool, guiding developers in optimizing their applications for modern multi-core

processors. Furthermore, we will delve into the practical aspects of implementing these frameworks, including common pitfalls and best practices that can help maximize performance and efficiency.

In addition to performance metrics, we will also consider the ease of use and learning curve associated with each framework, recognizing that developer productivity and maintainability are crucial factors in the real-world application of parallel programming. We will examine case studies and examples to illustrate how each framework can be applied to different types of computational problems, providing a clear understanding of their practical applications. Additionally, we will discuss the community and ecosystem support available for OpenMP and Parallel.For, as these resources can play a significant role in the adoption and successful implementation of these tools.

This paper is organized as follows: Section 2 provides an in-depth overview of parallel processing in C using OpenMP, including its syntax, features, and typical use cases. Section 3 offers a detailed examination of parallel processing with C# using Parallel.For, covering its core concepts, implementation details, and integration within the .NET ecosystem. Section 4 addresses the methodology used for the experiments conducted, detailing the setup, benchmarks, and criteria for evaluation. In Section 5, we present and discuss the results of our empirical analysis, comparing the performance, scalability, and ease of use of OpenMP and Parallel.For. Finally, we conclude in the last section, summarizing our findings and offering recommendations for future work.

Through this detailed comparative study, we aim to equip developers with the knowledge needed to make informed decisions when selecting parallel programming frameworks, ultimately enhancing the performance and efficiency of their computationally intensive tasks.

2. PARALLEL PROCESSING IN C WITH OPENMP

Parallel processing in C with OpenMP offers developers a straightforward approach to introduce concurrency into their programs. OpenMP, which stands for Open Multi-Processing, is an industry-standard API for shared-memory parallel programming in C, C++, and Fortran. It provides a set of compiler directives, runtime library routines, and environment variables that enable developers to parallelize their code easily.

At the heart of OpenMP are compiler directives, which are special comments embedded within the source code to instruct the compiler on how to parallelize certain sections of code. These directives take the form of pragmas, which are recognizable by the compiler and interpreted accordingly. For example, the #pragma omp parallel directive (Torres, Ferrer and Teruel, 2022) initiates a parallel region in the code, indicating that the enclosed block should be executed in parallel by multiple threads.

One of the key features of OpenMP is its simplicity and ease of use. Developers can parallelize loops, sections of code, or even entire functions with minimal changes to the original codebase. For instance, adding parallelism to a loop is as simple as prefixing it with #pragma omp parallel for, instructing the compiler to distribute iterations of the loop among available threads automatically.

OpenMP abstracts many of the low-level details of parallel programming, such as thread creation, synchronization, and load balancing, allowing developers to focus on algorithm design rather than intricacies of thread management. Additionally, OpenMP provides mechanisms for controlling thread behavior, such as specifying the number of threads to use (omp_set_num_threads) or defining thread-private variables (private clause).

Moreover, OpenMP supports a variety of synchronization constructs, such as barriers, atomic operations, and reduction operations, enabling developers to coordinate parallel execution effectively. These constructs ensure that threads synchronize appropriately when necessary, maintaining data consistency and avoiding race conditions.

In summary, OpenMP simplifies parallel programming in C by providing a high-level abstraction that allows developers to harness the power of multi-core processors without delving into complex threading constructs. Its ease of use, combined with robust performance optimizations and widespread compiler support, makes it a popular choice for parallel programming in scientific, engineering, and high-performance computing applications.

3. PARALLEL PROCESSING IN C# WITH PARALLEL.FOR

Parallel processing in C# is facilitated by the Parallel.For loop construct. This powerful feature enables developers to execute iterations of a loop in parallel across multiple threads, exploiting the capabilities of modern multi-core processors.

The Parallel.For loop operates similarly to a traditional for loop but distributes the loop iterations across multiple threads, allowing them to execute concurrently. This concurrent execution can lead to significant performance improvements for computationally intensive tasks, as it effectively utilizes all available CPU cores.

Using Parallel.For is straightforward. Developers specify the range of iterations to be processed and provide a delegate that defines the operation to be performed on each iteration. The TPL then automatically partitions the

iteration space and distributes the work among threads, handling thread creation, synchronization, and load balancing transparently.

One of the advantages of Parallel.For is its simplicity and ease of use. Developers can parallelize existing loop-based algorithms with minimal changes to the codebase, improving performance without introducing complex threading logic. Additionally, Parallel.For abstracts away many of the low-level details of parallel programming, such as thread management and synchronization, allowing developers to focus on algorithm design and application logic.

Another notable feature of Parallel.For is its integration with the .NET ecosystem. As part of the TPL, Parallel.For seamlessly integrates with other asynchronous programming constructs in C#, such as asynchronous methods and the async/await pattern. This integration enables developers to combine parallel processing with asynchronous I/O operations, enhancing overall application performance and responsiveness.

Furthermore, Parallel.For provides options for controlling the degree of parallelism, allowing developers to specify the maximum number of threads to use or let the TPL dynamically adjust the parallelism based on the system's capabilities. This flexibility enables developers to fine-tune performance and resource utilization according to the requirements of their applications.

In summary, Parallel.For in C# offers a convenient and efficient way to introduce parallel processing into applications, leveraging multi-core processors to enhance high performance computing. Its simplicity, integration with the .NET ecosystem (Cvijić and Ranilović, 2024), and flexibility in controlling parallelism make it a valuable tool for developers seeking to optimize their C# applications for modern hardware architectures.

4. METHODOLOGY

Reduction, a prevalent parallel computing pattern, involves condensing a collection of values into a single result through an associative operation, such as summation or multiplication (Karstadt, 2020). This section illustrates the implementation of a reduction algorithm utilizing OpenMP in C and Parallel.For in C#.

The reduction algorithm operates by partitioning the dataset into smaller segments, processing each segment independently, and subsequently combining the intermediate results to obtain the final output. The associative nature of the reduction operation facilitates parallelization, allowing concurrent processing of different segments by multiple threads.

Given the inputs and memory configurations, we evaluate the parallel algorithm execution speed and, in the end, calculate and analyze the speedup ratio according to (1).

SpeedUp = Parallel Execution Time / Sequential Execution Time (1)

The experiments involve varying dataset sizes and computational complexities to assess the scalability and performance characteristics of each approach. The data sets scale as 50K, 100K, 200K, 500K, 1M and 2M items.

5. RESULTS

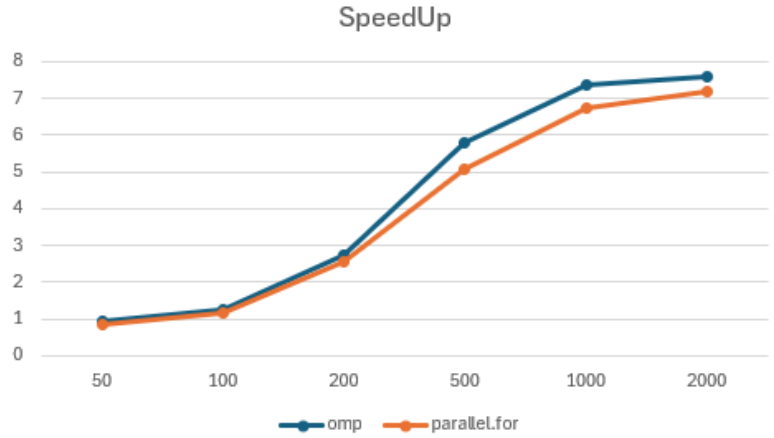
The implementations of the reduction algorithm, all sequential, parallel with OpenMP and parallel with Parallel.For were executed on the following hardware configuration:

1. *CPU: Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz*.
2. *RAM*: *Ramaxel* 8GB 2400MHz DDR4 PC4-19200 non-ECC Unbuffered SoDimm OEM Laptop Memory (Model: RMSA3260NA78HAF-2400)
3. Motherboard: HP ENVY x360 Convertible 15-dr0xxx

The choice of hardware was made to provide a baseline for performance evaluation across the implemented algorithms. The Intel Core i7-8565U processor, featuring four cores and eight threads, offers sufficient computational power for conducting parallel processing experiments. The 8GB RAM configuration ensures adequate memory resources for the execution of the algorithms under study.

What we can see in Figure 1 are the results from the execution of reduction sequentially, using OpenMP and using Parallel.For. What we can analyze in the Figure 1 are the execution times for all the scaling factors mentioned above.

Figure 1: Spead of the parallel execution of OpenMP and Parallel.For



6. CONCLUSION

OpenMP presents a straightforward solution for programmers who are already familiar with C/C++, requiring minimal modifications to existing codebases. Its extensive support in scientific and engineering applications further reinforces its suitability for these domains, thanks to its seamless integration with C/C++. On the other hand, Parallel.For in C# is highly intuitive, especially for developers who are accustomed to the .NET framework and lambda expressions. Its seamless integration with the .NET ecosystem makes it particularly ideal for enterprise and desktop applications.

Both frameworks offer substantial performance enhancements for parallelizable tasks, each excelling in different aspects. OpenMP tends to deliver superior performance for finely tuned applications, benefiting from the detailed control it provides over parallel execution. This control allows developers to optimize performance based on specific hardware characteristics, making it highly effective for applications where performance is critical and fine-tuning is feasible. In contrast, Parallel.For emphasizes simplicity and ease of use, enabling rapid development and reducing the complexity of writing parallel code. This abstraction makes it an excellent choice for developers who prioritize ease of use and quick deployment over granular performance optimizations.

The choice between OpenMP and Parallel.For largely depends on the specific requirements of the application and the development environment. Understanding the strengths and limitations of each framework is crucial for making an informed decision. OpenMP's control and performance optimization capabilities make it ideal for high-performance computing and applications requiring detailed hardware-specific tuning. Parallel.For, with its user-friendly interface and integration within the .NET ecosystem, suits scenarios where development speed and ease of maintenance are more critical.

Ultimately, the decision should be guided by the nature of the task at hand, the existing codebase, the target environment, and the specific performance requirements. By leveraging the strengths of each framework appropriately, developers can effectively optimize their applications for modern multicore processors, achieving the desired balance between performance, ease of use, and development efficiency. Future work could explore hybrid approaches that combine the best of both frameworks, potentially offering even greater flexibility and performance. Additionally, investigating other emerging parallel programming models could provide further insights into optimizing multicore processor utilization.

Moreover, ongoing advancements in processor architectures and parallel programming techniques will continue to shape the landscape, making it essential for developers to stay informed about new developments. Engaging with the broader developer community through forums, conferences, and publications can also provide valuable perspectives and innovative solutions. By staying adaptable and continually evaluating new tools and methodologies, developers can ensure their applications remain efficient and competitive in an ever-evolving technological landscape.

LITERATURE

Aho, A. V., Kernighan, B. W., & Weinberger, P. J. (2023). *The AWK programming language*. Addison-Wesley Professional.

Bak, S., Bertoni, C., Boehm, S., Budiardja, R., Chapman, B. M., Doerfert, J., ... & Yeung, P. K. (2022). OpenMP application experiences: Porting to accelerated nodes. *Parallel Computing*, 109, 102856.

Chandra, R. (2001). *Parallel programming in OpenMP*. Morgan Kaufmann.

Chi, Y., Guo, L., Lau, J., Choi, Y. K., Wang, J., & Cong, J. (2021, May). Extending high-level synthesis for task-parallel programs. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (pp. 204-213). IEEE.

Ciccozzi, F., Addazi, L., Asadollah, S. A., Lisper, B., Masud, A. N., & Mubeen, S. (2022). A comprehensive exploration of languages for parallel computing. *ACM Computing Surveys (CSUR)*, 55(2), 1-39.

Cvijić, B., & Ranilović, P. (2024). From. NET Core to. NET 8: A Comprehensive Analysis of Performance, Features, and Migration Pathways. *JITA-APEIRON*, 27(1), 69-77.

Griffiths, I. (2022). *Programming C# 10*. " O'Reilly Media, Inc.".

Karstadt, E., & Schwartz, O. (2020). Matrix multiplication, a little faster. *Journal of the ACM (JACM)*, 67(1), 1-31.

Okur, Semih, and Danny Dig. "*How do developers use parallel libraries?*." Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. 2012.

Torres, R., Ferrer, R., & Teruel, X. (2022, May). A novel set of directives for multi-device programming with openmp. In 2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) (pp. 401-410). IEEE.

Zhu, X., Whitehead Jr, E. J., Sadowski, C., & Song, Q. (2015). *An analysis of programming language statement frequency in C, C++, and Java source code*. Software: practice and experience, 45(11), 1479-1495.