International Journal of Applied Information Systems (IJAIS) – ISSN : 2249-0868
Foundation of Computer Science FCS, Delaware, USA
Volume 13– No. 2, February 2026 – www.ijais.org

Loop Detection and Parallelization for C++ Code using the OpenMP

Shuruq Alsaedi
Department of Computer Science
Faculty of Computing and Information Technology
King Abdulaziz University Jeddah, Saudi Arabia

Amal AlMansour
Department of Computer Science
Faculty of Computing and Information Technology
King Abdulaziz University Jeddah, Saudi Arabia

Lama Al Khzayem
Department of Computer Science
Faculty of Computing and Information Technology
King Abdulaziz University Jeddah, Saudi Arabia

Fathy E. Eassa
Department of Computer Science
Faculty of Computing and Information Technology
King Abdulaziz University Jeddah, Saudi Arabia

Rsha Mirza
Department of Computer Science
Faculty of Computing and Information Technology
King Abdulaziz University Jeddah, Saudi Arabia

ABSTRACT
Finding solutions that increase the speed of the code running process has become a necessity, especially in the current era, which has witnessed the evolution of processor speed techniques and tools. However, in order to fully benefit from such advanced processors, code should be written in a way that can utilize these processors' advantages. In fact, if a sequential code is transformed into parallel code, it would reduce the execution time, resulting in better performance and more efficient utilization of processor resources, especially for large problem sizes. Although automating the code parallelization process would save programmers time and effort, as well as help avoid many programming errors, the parallelization process requires a good knowledge of several important factors such as dependency analysis and identifying parallelizable regions in the code, which can be challenging to handle manually. This study introduces a novel automatic code translation and optimization tool that converts C++ sequential code into parallel code using the OpenMP programming model. To validate the tool, four benchmark programs were tested: a dot product achieved a speedup of up to 5.39× on 12 threads; an array sum reduced execution time from 0.038 seconds in the sequential case to 0.0052 seconds in the parallel case, giving a 9.03× speedup; matrix multiplication of size 1000×1000 improved from 2.91 seconds in sequential execution to 0.39 seconds in parallel execution with a 6.80× speedup; and numerical integration using the trapezoidal rule reached a speedup of 9.66× at 14 threads. These results demonstrate that the proposed approach with OpenMP programming model delivers consistent performance gains across diverse computational workloads.

Keywords
Sequential code ; Single programming model; parallel code, automatic code translation; C++; OpenMP.

1. INTRODUCTION
Our lives depend on computation in almost every single aspect, from the smallest everyday issues, such as checking bus stop times, to larger problems, such as medical surgery techniques or space-related challenges. Traditional computation techniques don’t meet the new requirements that arise from today's needs, such as saving time and effort for both programmers and users. Sequential execution takes longer than parallel execution. In today's life, there is a need for fast computing to execute complicated tasks in a short time. Serial execution performs one task after another, which takes time [1]. Parallel computing means the process that requires running a certain program or performing a computation on a number of processors at one time [2]. In this type of architecture, different problems are split into smaller tasks that can be performed at one time [2]. However, parallel execution requires programmers to have good knowledge and experience in many important parallelization-related factors, such as dependency analysis and identifying parts of the sequential code that can be parallelized. This leads to many challenges and consumes time and effort for programmers. Automating the process of sequential to parallel code translation will reduce programmer effort, save time, and avoid errors that may arise from less experienced programmers in parallelization techniques.
In fact, High-Performance Computing (HPC) systems have a strong impact on the field of computational science, which helpsevo researchers and programmers to tackle complex problems that were previously beyond reach. The evolution of parallel computing has created new challenges for parallel systems developers as well as users [3]. Many tools have significantly contributed to the improvement of parallel programming, such as OpenMP, which is very beneficial for parallelizing code in C++ [4].

HPC systems are mainly designed in order to facilitate the processing of complex calculations that usually are needed at high-speed systems, enabling many processors to be working at the same time on several parts of a certain problem. In fact, such parallel processing ability is not easy for many tasks that demand complex computational processes, such as weather forecasting systems or molecular modeling ones, as well as large-scale simulations in medicine, engineering, and physics. However, balancing the workload distribution among various processors allows HPC systems to reduce the required computation time. Thus, handling big datasets efficiently, Figure 1 illustrates major application domains that rely on high-performance computing.

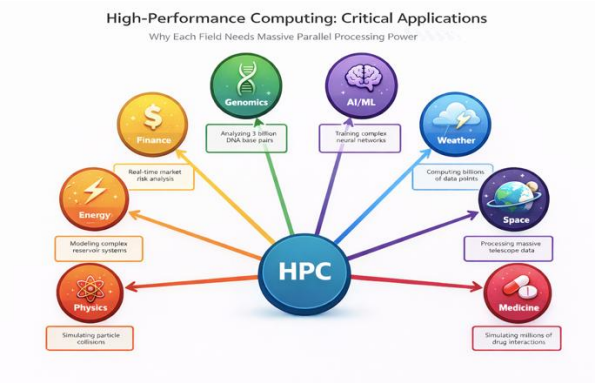


Fig 1: Major application domains that rely on HPC

OpenMP is a popular tool for parallel programming that is designed primarily for shared memory systems, such as multicore computers. OpenMP utilizes compiler directives, which parallel system programmers insert into their code. After the compiler processes these directives, it hands the relevant code blocks to a runtime system that is architecture-specific for parallel computation purposes [5]. A lot of compilers support OpenMP, and many programmers are attracted to it due to its useful parallelization techniques. Fortran and C/C++ languages include OpenMP abstractions. Such abstractions facilitate parallelization by providing different controlling degrees over it. Moreover, directives of OpenMP can be inserted incrementally by the programmers in order to reach the performance they require [5]. OpenMP simplifies parallelism in shared memory systems, in which different processors use a certain memory. OpenMP utilizes the directives for parallelization purposes of loops and other suitable code parts. In this way, sequential programs can be converted into parallel programs without the need of rewriting codes.

C++ provides several features that make it suitable for HPC systems. The C++ programming language is a powerful language and offers flexible and high performance. It combines efficiency and compatibility, which makes it popular in software development [6]. Due to C++ performance, and its low-level and high-level programming supporting, as well as its flexibility, it became a preferred language in HPC systems. With the use of C++, programmers can write optimized, efficient code that simplifies complicated tasks [7][8]. Taking advantages of C++ together with with OpenMP programming model, parallel application development becomes more powerful. C++ features allow creating maintainable, flexible, and modular codes [9][10].

This paper introduces a novel automatic code translation tool which converts C++ sequential code into parallel code using the OpenMP programming model. A number of programs were selected and tested to evaluate the study objectives. The study's findings show improvements in the performance of C++ code and a decrease in execution time, which reduces the effort required to convert from sequential to parallel processing. The main contribution of the current study is introducing our auto code generation tool that applies dependency analysis and converts sequential C++ code to parallel C++ code based on a single programming model (OpenMP).

The rest of this paper is organized as follows. Background and historical view regarding processors and their evolutions are presented in Section 2. Section 3, which is the related work section, discusses a number of existing tools for parallel programming and related programming models. In Section 5, the methodology and proposed system architecture are described. In Section 6, we discuss in detail the achieved results in our different experiments along with the measuring factors and selected attributes for evaluating our proposed translation technique. Finally, conclusions and future work are presented in Section 7.

2. BACKGROUND

In this section, we are going to discuss the background, which goes through a historical view about the processors' evolution from their beginning until how they became today. Moreover, it discusses improvements in parallel programming to efficiently utilize the features and capabilities of processors. Processors are important to High-Performance Computing (HPC) because they facilitate efficient execution, especially in complex calculations. Choosing the suitable architecture is critical for better performance. Figure 2 below shows a simplified HPC architecture which includes many HPC nodes (clusters) with multicore processors.

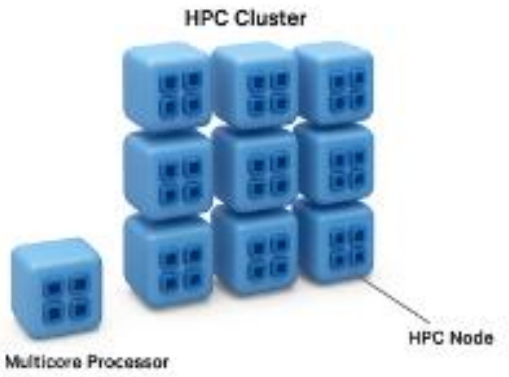


Fig 2: Simplified HPC architecture

Processors have gone through different stages and changes since 1971 till 2024. In fact, there are significant points to consider in processor design, which are: size, speed, power consumption, efficiency, and performance [11]. Earlier processors were coming with a single core and less speed, with less bandwidth and thus high latency, while processors today come with multi-cores and threads, and high speed with less latency [12].

In the information era, which we live in today, microprocessors are considered its icon. In fact, microprocessors were initially developed from transistors, and then from the integrated circuit [13]. We find microprocessors everywhere, whether in powerful computers or in tiny electronic toys [13]. Today, some microprocessors have about one billion transistors in certain heavy devices [13]. Sometimes, the terms 'microprocessor' and 'CPU' are used in an exchangeable way for denoting the same thing. The common variant categories of microprocessors are the Complex Instruction Set Computers (CISC) and the Reduced Instruction Set Computers (RISC), as well as the Very Long Instruction Word Computers (VLIW) and the superscalar processors.

The first dynamic RAM was introduced by Intel in 1970, while the first microprocessor available to the public was Intel 4004, which was a single-chip 4-bit CPU and was basically made to be used in calculators. Intel 4004 gained high popularity among customers and designers as it was the first single-chip microprocessor [13]. The introduced microprocessors in the era

from 1971 to 1973 are known as the first-generation systems, which processed instructions one by one or sequentially. The instruction first was fetched, then decoded, and after that executed. After completing the instruction, the microprocessor repeated the same steps with the next instruction.

During the same period, other microprocessors were founded, such as Rockwell International's PPS-4 and National Semiconductor's IMP-16, which were manufactured using technologies like PMOS that produced low output and slow speed [13]. In the second generation of microprocessors, which was in the late 1970s, enough numbers of transistors were provided on the IC. The second generation noticed the initiation of proficient 8-bit microprocessors [13]. Intel 8080, Zilog Z80, and Motorola 6800 were some of the known processors of that generation. In fact, instruction execution in the second generation was defined by step overlapping, which are fetch, decode, and execute overlapping. While one instruction is being processed in the execution phase, another is being decoded, while the next is being fetched. However, the difference between the devices of the first generation and the second one was basically the use of new semiconductor technology to make and design the chips. The newer technology improved instruction speed and execution [13].

The third microprocessor generation was from 1979 to 1980. It was known for Intel's 8086 and Zilog Z8000; both processors were 16-bit. The third generation of microprocessors differed from earlier generations in that many important workstation manufacturers started making RISC architecture microprocessors. Microprocessors at the third age were 16-bit [13].

3. RELATED WORK

The related work section is discussing some existing tools in the parallel programming field and different associated programming models such as MPI, CUDA, and OpenMP. For many years, parallel programming was one of the hot topics that attracted both researchers and programmers, as well as users [14]. Although parallelization has many advantages in terms of execution speed and code optimization, it is not an easy task [14]. Moreover, code parallelization would lead to different errors, so programmers try to overcome this challenge by developing tools that would help in minimizing such errors. Parallel programming can be done manually by programmers or automatically. There are some advantages and disadvantages to both ways [15]. In fact, developing tools that automatically parallelize codes with fewer errors is very important to speed up execution. Nowadays, processors have powerful capabilities, so using sequential code doesn't fully utilize their potential and thus does not achieve optimal performance. It's important to effectively use the processors' features and capabilities by improving code parallelization.

Despite important improvements in computer technology, researchers and developers did not yet efficiently fill the gap in parallelization methods. Current techniques usually do not fully meet performance optimization criteria over different hardware architectures. There are many challenges that need to be resolved, such as data dependencies, efficient resource utilization, and load balancing. There are many current parallelization techniques that have some problems, such as being too specific or too general, which narrows their effectiveness. Moreover, the huge evolution in processor design presented many complexities that exceed the current solutions. Therefore, reaching effective code parallelism techniques in software development maintains its status as a work in progress.

Sardar and Faizabadi in [16] investigated the use of both hand-coded OpenMP and the PLUTO automatic parallelization tool on a set of well-known numeric problems, including matrix multiplication, matrix vector multiplication, matrix addition, Dijkstra's algorithm, and Floyd's algorithm. Their results highlighted that some algorithms, such as matrix multiplication and Floyd's algorithm, benefited significantly from automatic parallelization, while others performed better under manual OpenMP directives.

The parser [17] transforms the file that contains assembler input into program objects that can be processed later. After that, static single assignment is done. Using the data flow graph, the parallelization algorithm distributes instructions across multiple cores. When the sequential code is parallelized, registers are allocated using a linear allocation algorithm, resulting in distributed assembler code across each core. The paper evaluates the speedup achieved in a matrix multiplication example processed by the assembly code parallelizer.

The ParaGraph [18] tool is designed for C code; it doesn't work independently. Instead, it is made to function as an Eclipse plug-in; it can be considered as an extra element of the C/C++ development tool plug-in. ParaGraph requires using another compiler for parallelized code generation by adding suitable OpenMP directives for the applicable loops.

The Autopar tool's purpose [19] is to automatically generate parallelized recursive functions. It performs static analysis. First, it scans a given code to detect the recursive functions. Then, it makes some analysis to gather information related to those recursive functions. The analysis includes gathering required data without doing any changes to the program's design. Finally, it automatically generates the parallelized code by inserting the relevant OpenMP directives in the suitable parts of the program.

In [20], the tool transforms an input C code to a multi-threaded, parallelized C code. This tool enables parallelized multi-level OpenMP constructs. The scheme decomposes the C code into coarse-grain constructs, performs dependency analysis, and then generates an OpenMP parallelized program.

The tool [21] automates the process of converting serial C code into equivalent parallel CUDA code through a two-phase process. The first phase includes the static analysis of the given C code to recognize code parts that can be parallelized. The second phase involves automating the conversion from C code to the corresponding CUDA code. Table 1 summarizes a set of selected attributes [22] in related work.

Table 1. Summary of main points in related work

Title	Approach	Input
A solution for automatic parallelization of sequential assembly code (2013)	Parallelization algorithm	Assembly code
C code parallelization with ParaGraph (2010)	An Eclips Plug- in	C
Autopar: An automatic parallelization tool for recursive calls (2014)	Applying set of Algorithms	ANSI C

Automatic Code Parallelization with OpenMP Task Constructs (2016)	Analysis of decomposed C code into coarse grain tasks	C
Automated Tool to Generate Parallel CUDA code from a Serial C Code (2012)	Analysis of Static C	C
Parallelization and analysis of selected numerical algorithms using OpenMP and Pluto on symmetric multiprocessing machine (2018)	Applying set of Algorithms + Pluto tool	C

Expanding prior approaches, our contribution focuses on the automatic translation of sequential C++ code into parallel equivalents, targeting shared-memory architectures. Our work introduces a single-model framework that leverages OpenMP for shared-memory systems. This support represents an important contribution, enabling broader applicability across heterogeneous environments. Furthermore, our tool incorporates automated dependency analysis generation to ensure correctness and facilitate deeper structural insights. Through evaluation on benchmark programs, we demonstrate the scalability and effectiveness of our approach.

4. MATERIALS AND METHODS

This study aims to develop an automated tool that transforms sequential C++ code into parallel code using a single-model OpenMP directive approach. In fact, this transformation focuses on identifying independent computational regions (independent loops), which can benefit from parallel execution and enhance performance, especially on modern multi-core processors. The methodology section is structured into different phases, each of which plays an important role in the organized transformation of sequential code into a parallelized version.

The translation process starts by accepting sequential C++ code as input and then scanning it to detect potential parallel regions. After that, a detailed dependency analysis is performed to ensure the correctness of the parallelization by detecting statement dependencies within loops. Note that only loops that are independent and free of cross-iteration dependencies can be parallelized using OpenMP directives. This systematic approach ensures that the parallelized code remains functionally equivalent to the original sequential C++ code while achieving better computational efficiency.

Moreover, the methodology not only addresses automating the code transformation process but also emphasizes careful parallelization techniques, ensuring that only suitable loops are parallelized. The following sections describe the phases of the methodology, including their processes, objectives, and outcomes. Figure 3 shows the roadmap of the proposed tool, starting from the input of the sequential C++ code, through scanning, analysis, decision-making, and code generation, and ending with the final parallelized code.

This proposed automated translation tool systematically takes sequential C++ code as input, detects potential regions for parallelization (loops), conducts dependency analysis, and then generates a parallelized version using OpenMP directives where it is suitable.

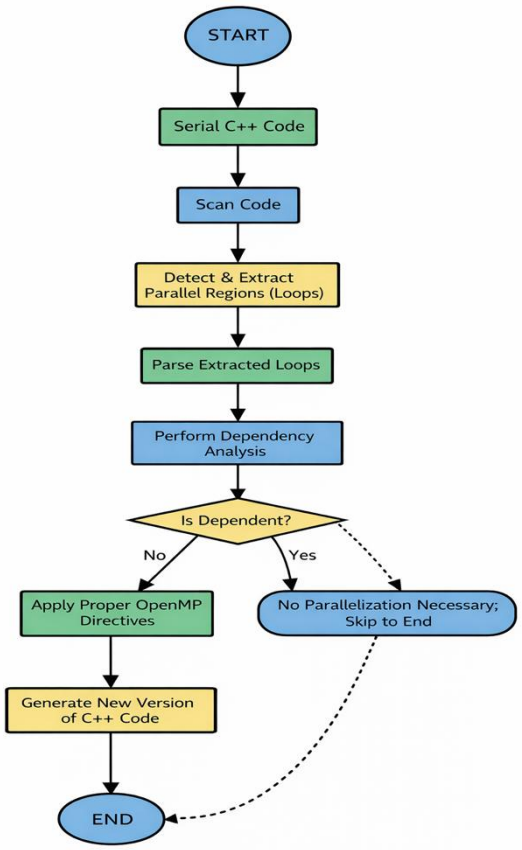


Fig 3: Architecture of the Proposed Translation Tool

Each phase contributes to ensuring that only loops applicable for parallel execution are processed, thus enhancing the sequential C++ code performance through parallelization. The methodology begins with the input phase, where the sequential C++ code is provided to the proposed tool, which works as the base for the following parallelization process. Next, in the scanning and detection phase, the tool analyzes the code to detect and extract the potential parallel regions, particularly loops, for further analysis. The output of this step is the identification of parallel regions inside the C++ code.

Next, in the parsing and dependency analysis phase, the extracted loops are analyzed to determine dependencies between statements. This analysis identifies whether the loops are suitable for parallelization or not. If a certain extracted loop is independent, it proceeds to the parallelization decision phase, where a suitable OpenMP directive is applied for parallel execution. On the other hand, if dependencies are detected, then parallelization is not applicable as it is important to avoid incorrect execution results.

For loops which are suitable for parallelization, the code generation phase generates new C++ code, incorporating the appropriate OpenMP directives. The final output is a parallelized version of the input C++ code, including OpenMP directives that enable efficient parallel execution, resulting in a performance improvement over the initial sequential version and time-saving. The methodology is divided into several

phases as described below.

4.1 Parsing and Dependency Analysis

In this phase, the extracted loops are parsed, and a dependency analysis is conducted. The analysis aims to determine if there are dependencies between statements of the loops, which may prevent safe parallelization. Note that the dependency analysis procedure is a critical step in determining whether a loop can be safely parallelized without compromising the correctness of the program. The goal is to analyze each loop statement to identify any dependencies that might inhibit parallel execution. The output of this phase is the dependency analysis result, which indicates whether the loops have dependencies or if they are independent and thus can be parallelized.

4.2 Parallelization Decision and Directive Application

In this phase, if the extracted loop is identified as independent, a suitable OpenMP directive is applied to parallelize the loop. The selection of the appropriate directive is based on the dependency analysis carried out in the previous phase. However, if the loop is found to contain dependencies, then parallelization is not applicable due to the risk of incorrect results from concurrent execution.

4.3 Parallelized Code Generation Phase

For loops that are independent and suitable for parallelization, this phase generates new C++ code, incorporating the suitable OpenMP directives into the original sequential code. The output is the generated code that contains the necessary constructs for parallel execution. The final output of the proposed tool is a parallelized version of the original input code, providing a performance improvement over the initial sequential version through parallel execution of independent loops.

5. IMPLEMENTATION

In this section, we describe the chosen platform for the experiment, along with the programs chosen to apply the experiment. Additionally, the computation metrics used for evaluating the proposed translation tool are described. These metrics include computing performance (execution time). A comparison between sequential and parallel execution is provided to enhance understanding of the code execution and the experiment.

5.1. Experimental Platform and Measuring Factors

To evaluate our proposed translation tool using a dual programming model, we conducted experiments on four different programs, all of which demonstrated the expected results from the proposed tool, proving its effectiveness and success in meeting the outlined goals, as explained next:

5.1.1 Dot product program

The program computes the dot product of vectors; the corresponding elements are multiplied, then results are summed. Different vector sizes are tested, ranging from 10,000 to 10,000,000 elements, with 1 to 16 threads. The goal is to evaluate how parallel processing with multiple threads enhances performance and reduces execution time compared to sequential execution.

5.1.2 Array Sum

The goal of this program is calculating the sum of arrays. Seven different arrays are generated with up to a billion elements. The different array sizes are: 1,000; 10,000; 100,000; 1,000,000; 10,000,000; 100,000,000; and 1,000,000,000 elements. There are 16 threads used in the parallel approach. Execution times are measured for parallel and sequential methods.

5.1.3 Array multiplication

The array multiplication program computes the dot product of a number of different matrix sizes, which are (10×10, 100×100, and 1000×1000). Both parallel and sequential methods are computed. It uses multiple threads (1 to 16) to evaluate the performance achieved through parallel processing compared to sequential computation.

All of the experiments were performed on a personal computer; the processor comes with 8 cores and 16 threads.

5.1.4 Numerical integration of x^2 on [0,1], using the trapezoidal rule

This program computes the definite integral of x^2 over a given interval [a, b] by using the trapezoidal rule. It finds the area under a curve by dividing it into trapezoids, then summing the resulting areas. It computes execution times for various threads (1 to 16), focusing on how the performance varies when changing the number of threads and the number of trapezoids. After that, it compares these with a serial computation.

6. RESULTS AND DISCUSSION

In our experiments, we measured the execution time for four different programs using our proposed translation tool, which is based on a dual programming language approach. For each program, we compared parallel execution against sequential execution. Additionally, in each program, we tested various values to gain a better understanding of the performance. Each program's test experiment is described in detail as follows:

6.1 Dot Product

The dot product program tests performance using different numbers of threads and vector sizes. The essential metrics here are the number of threads (1 to 16), the size of the vectors (1,000; 10,000; 100,000; 1,000,000; 10,000,000), the resulting dot product, and the execution time (secs).

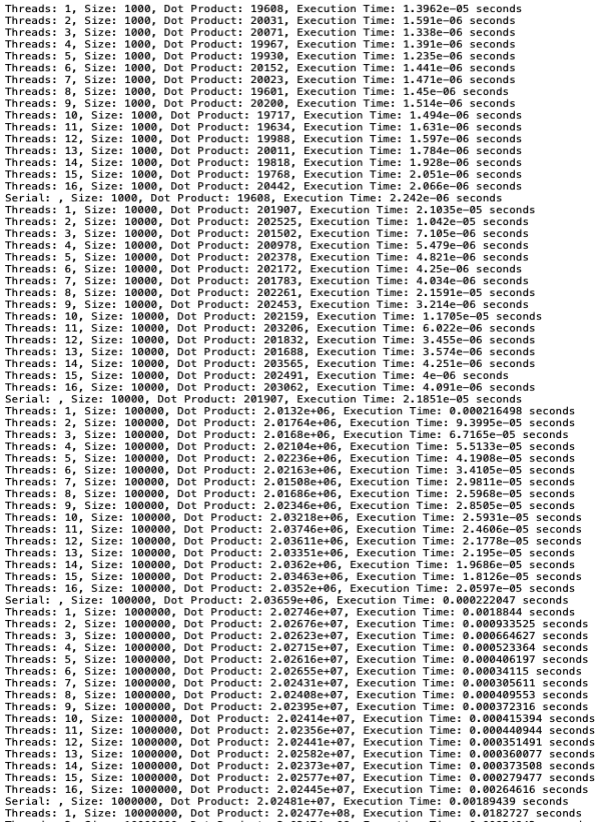


Fig 4: Results of computation of the dot product of vectors of different sizes

The data of the expermit as showed in Figure 4 and table 2,3,4,5 and 6 are number of threads (1 to 16) , vector size (1000 to 10000000) , resulted dot product, parallel execution time (Secs), and sequential execution time.

Table 2. Dot product of vector size 1000

Thread	Size	Dot Product	Parallel Execution Time (seconds)	Serial Execution Time (seconds)
1	1,000	19,608	1.3962e-05	2.242e-06
2	1,000	20,031	1.591e-06	2.242e-06
3	1,000	20,071	1.338e-06	2.242e-06
4	1,000	19,967	1.391e-06	2.242e-06
5	1,000	19,930	1.235e-06	2.242e-06
6	1,000	20,152	1.441e-06	2.242e-06
7	1,000	20,023	1.471e-06	2.242e-06
8	1,000	19,601	1.450e-06	2.242e-06
9	1,000	20,200	1.514e-06	2.242e-06
10	1,000	19,717	1.494e-06	2.242e-06
11	1,000	19,634	1.631e-06	2.242e-06
12	1,000	19,988	1.597e-06	2.242e-06
13	1,000	20,011	1.784e-06	2.242e-06
14	1,000	19,818	1.928e-06	2.242e-06
15	1,000	19,768	2.051e-06	2.242e-06
16	1,000	20,442	2.066e-06	2.242e-06

Table 3. Dot product of vector size 10000

Thread	Size	Dot Product	Parallel Execution Time (seconds)	Serial Execution Time (seconds)
1	10,000	201,907	2.1035e-05	2.1851e-05
2	10,000	202,525	1.042e-05	2.1851e-05
3	10,000	201,502	7.105e-06	2.1851e-05
4	10,000	200,978	5.479e-06	2.1851e-05
5	10,000	202,378	4.821e-06	2.1851e-05
6	10,000	202,172	4.250e-06	2.1851e-05
7	10,000	201,783	4.034e-06	2.1851e-05
8	10,000	202,261	2.1591e-05	2.1851e-05
9	10,000	202,453	3.214e-06	2.1851e-05
10	10,000	202,159	1.1705e-05	2.1851e-05
11	10,000	203,206	6.022e-06	2.1851e-05
12	10,000	201,832	3.455e-06	2.1851e-05
13	10,000	201,688	3.574e-06	2.1851e-05
14	10,000	203,565	4.251e-06	2.1851e-05
15	10,000	202,491	4.000e-06	2.1851e-05
16	10,000	203,062	4.091e-06	2.1851e-05

Table 4. Dot product of vector size 100000

Thread	Size	Dot Product	Parallel Execution Time (seconds)	Serial Execution Time (seconds)
1	100,000	2,013,200	0.000216498	0.000222047
2	100,000	2,017,640	9.3995e-05	0.000222047
3	100,000	2,016,800	6.7165e-05	0.000222047
4	100,000	2,021,040	5.5133e-05	0.000222047
5	100,000	2,022,360	4.1908e-05	0.000222047
6	100,000	2,021,630	3.4105e-05	0.000222047
7	100,000	2,015,080	2.9811e-05	0.000222047
8	100,000	2,016,860	2.5968e-05	0.000222047
9	100,000	2,023,460	2.8505e-05	0.000222047
10	100,000	2,032,180	2.5931e-05	0.000222047
11	100,000	2,037,460	2.4606e-05	0.000222047
12	100,000	2,036,110	2.1778e-05	0.000222047
13	100,000	2,033,510	2.1950e-05	0.000222047
14	100,000	2,036,200	1.9686e-05	0.000222047
15	100,000	2,034,630	1.8126e-05	0.000222047
16	100,000	2,035,200	2.0597e-05	0.000222047

Table 5. Dot product of vector size 1000000

Thread	Size	Dot Product	Parallel Execution Time (seconds)	Serial Execution Time (seconds)
1	1,000,000	20,274,600	0.0018844	0.00189439
2	1,000,000	20,267,600	0.000933525	0.00189439
3	1,000,000	20,262,300	0.000664627	0.00189439
4	1,000,000	20,271,500	0.000523364	0.00189439
5	1,000,000	20,261,600	0.000406197	0.00189439
6	1,000,000	20,265,500	0.000341150	0.00189439
7	1,000,000	20,243,100	0.000305611	0.00189439
8	1,000,000	20,240,800	0.000409553	0.00189439
9	1,000,000	20,239,500	0.000372316	0.00189439
10	1,000,000	20,241,400	0.000415394	0.00189439
11	1,000,000	20,235,600	0.000440944	0.00189439
12	1,000,000	20,244,100	0.000351491	0.00189439
13	1,000,000	20,258,200	0.000360077	0.00189439
14	1,000,000	20,237,300	0.000373508	0.00189439
15	1,000,000	20,257,700	0.000279477	0.00189439
16	1,000,000	20,244,500	0.00264616	0.00189439

Table 6. Dot product of vector size 10000000

Thread	Size	Dot Product	Parallel Execution Time (seconds)	Serial Execution Time (seconds)
1	10,000,000	202,477,000	0.0182727	0.0183282
2	10,000,000	202,474,000	0.00954843	0.0183282
3	10,000,000	202,628,000	0.00629823	0.0183282
4	10,000,000	202,413,000	0.00475025	0.0183282
5	10,000,000	202,538,000	0.00422083	0.0183282
6	10,000,000	202,537,000	0.00366860	0.0183282
7	10,000,000	202,513,000	0.00293110	0.0183282
8	10,000,000	202,485,000	0.00310606	0.0183282
9	10,000,000	202,496,000	0.00295158	0.0183282
10	10,000,000	202,491,000	0.00286786	0.0183282
11	10,000,000	202,462,000	0.00270819	0.0183282
12	10,000,000	202,403,000	0.00262760	0.0183282
13	10,000,000	202,493,000	0.00252521	0.0183282
14	10,000,000	202,495,000	0.00274522	0.0183282
15	10,000,000	202,546,000	0.00246188	0.0183282
16	10,000,000	202,556,000	0.00826185	0.0183282

As the above tables show, execution times are computed in seconds and represent the time taken to compute the dot product either in parallel (with the specified number of threads) or in sequential methods. However, the sequential execution time doesn't change because it is executed sequentially regardless of the number of threads. Note that the dot product values may vary slightly due to floating-point precision, but they are generally close to the expected result.

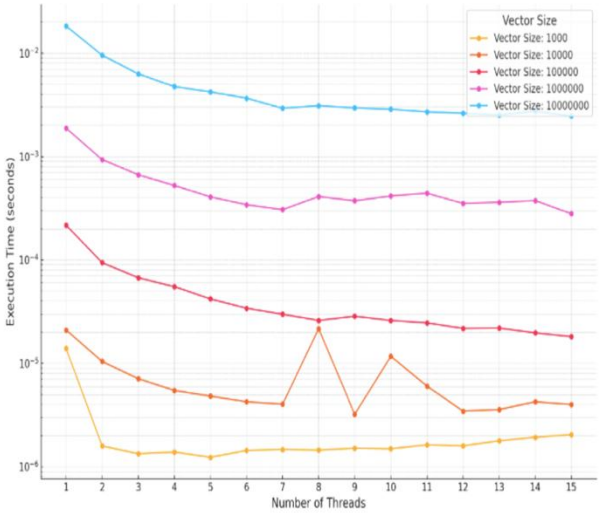
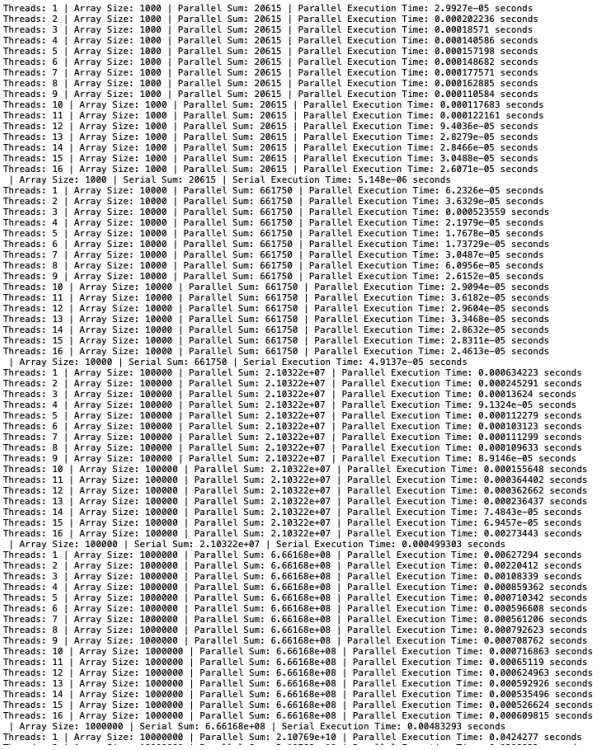


Fig 5: Execution time vs different number of threads for number of vector sizes

Figure 5 chart depicts the execution time against the number of threads for various vector sizes. As shown in Figure 4, the chart illustrates that execution time generally decreases as the number of threads increases, particularly in larger vector sizes. This indicates better performance with more parallelism. However, large vector sizes naturally result in longer execution times, but there is a noticeable reduction in execution time as the number of threads increases. This shows the impact of parallel processing. In fact, results show good scalability with the different number of threads.

6.2 Array Sum

In the array sum program we aim to calculate the sum of arrays of different sizes (1,000, 10,000, 100,000, 1,000,000, 10,000,000, and 1,000,000,000) with both parallel and sequential execution. The parallel execution is tested with up to 16 threads. Both parallel and sequential execution times are recorded. Note that execution time changes with increasing thread count and array size.



```
Threads: 1 | Array Size: 1000 | Parallel Sum: 20615 | Parallel Execution Time: 2.9927e-05 seconds
Threads: 2 | Array Size: 1000 | Parallel Sum: 20615 | Parallel Execution Time: 0.000202236 seconds
Threads: 3 | Array Size: 1000 | Parallel Sum: 20615 | Parallel Execution Time: 0.00018571 seconds
Threads: 4 | Array Size: 1000 | Parallel Sum: 20615 | Parallel Execution Time: 0.000140586 seconds
Threads: 5 | Array Size: 1000 | Parallel Sum: 20615 | Parallel Execution Time: 0.000157198 seconds
Threads: 6 | Array Size: 1000 | Parallel Sum: 20615 | Parallel Execution Time: 0.000148682 seconds
Threads: 7 | Array Size: 1000 | Parallel Sum: 20615 | Parallel Execution Time: 0.000177571 seconds
Threads: 8 | Array Size: 1000 | Parallel Sum: 20615 | Parallel Execution Time: 0.000162885 seconds
Threads: 9 | Array Size: 1000 | Parallel Sum: 20615 | Parallel Execution Time: 0.000110584 seconds
Threads: 10 | Array Size: 1000 | Parallel Sum: 20615 | Parallel Execution Time: 0.000117683 seconds
Threads: 11 | Array Size: 1000 | Parallel Sum: 20615 | Parallel Execution Time: 0.000122161 seconds
Threads: 12 | Array Size: 1000 | Parallel Sum: 20615 | Parallel Execution Time: 9.4036e-05 seconds
Threads: 13 | Array Size: 1000 | Parallel Sum: 20615 | Parallel Execution Time: 2.8279e-05 seconds
Threads: 14 | Array Size: 1000 | Parallel Sum: 20615 | Parallel Execution Time: 2.8466e-05 seconds
Threads: 15 | Array Size: 1000 | Parallel Sum: 20615 | Parallel Execution Time: 3.0488e-05 seconds
Threads: 16 | Array Size: 1000 | Parallel Sum: 20615 | Parallel Execution Time: 2.6071e-05 seconds
 | Array Size: 1000 | Serial Sum: 20615 | Serial Execution Time: 5.148e-06 seconds
Threads: 1 | Array Size: 10000 | Parallel Sum: 661750 | Parallel Execution Time: 6.2326e-05 seconds
Threads: 2 | Array Size: 10000 | Parallel Sum: 661750 | Parallel Execution Time: 3.6329e-05 seconds
Threads: 3 | Array Size: 10000 | Parallel Sum: 661750 | Parallel Execution Time: 0.000523559 seconds
Threads: 4 | Array Size: 10000 | Parallel Sum: 661750 | Parallel Execution Time: 2.1979e-05 seconds
Threads: 5 | Array Size: 10000 | Parallel Sum: 661750 | Parallel Execution Time: 1.7678e-05 seconds
Threads: 6 | Array Size: 10000 | Parallel Sum: 661750 | Parallel Execution Time: 1.73729e-05 seconds
Threads: 7 | Array Size: 10000 | Parallel Sum: 661750 | Parallel Execution Time: 3.0487e-05 seconds
Threads: 8 | Array Size: 10000 | Parallel Sum: 661750 | Parallel Execution Time: 6.0956e-05 seconds
Threads: 9 | Array Size: 10000 | Parallel Sum: 661750 | Parallel Execution Time: 2.6152e-05 seconds
Threads: 10 | Array Size: 10000 | Parallel Sum: 661750 | Parallel Execution Time: 2.9094e-05 seconds
Threads: 11 | Array Size: 10000 | Parallel Sum: 661750 | Parallel Execution Time: 3.6182e-05 seconds
Threads: 12 | Array Size: 10000 | Parallel Sum: 661750 | Parallel Execution Time: 2.9604e-05 seconds
Threads: 13 | Array Size: 10000 | Parallel Sum: 661750 | Parallel Execution Time: 3.3468e-05 seconds
Threads: 14 | Array Size: 10000 | Parallel Sum: 661750 | Parallel Execution Time: 2.8632e-05 seconds
Threads: 15 | Array Size: 10000 | Parallel Sum: 661750 | Parallel Execution Time: 2.8311e-05 seconds
Threads: 16 | Array Size: 10000 | Parallel Sum: 661750 | Parallel Execution Time: 2.4613e-05 seconds
 | Array Size: 10000 | Serial Sum: 661750 | Serial Execution Time: 4.9137e-05 seconds
Threads: 1 | Array Size: 100000 | Parallel Sum: 2.10322e+07 | Parallel Execution Time: 0.000634223 seconds
Threads: 2 | Array Size: 100000 | Parallel Sum: 2.10322e+07 | Parallel Execution Time: 0.000245291 seconds
Threads: 3 | Array Size: 100000 | Parallel Sum: 2.10322e+07 | Parallel Execution Time: 0.00013624 seconds
Threads: 4 | Array Size: 100000 | Parallel Sum: 2.10322e+07 | Parallel Execution Time: 9.1324e-05 seconds
Threads: 5 | Array Size: 100000 | Parallel Sum: 2.10322e+07 | Parallel Execution Time: 0.000112279 seconds
Threads: 6 | Array Size: 100000 | Parallel Sum: 2.10322e+07 | Parallel Execution Time: 0.000103123 seconds
Threads: 7 | Array Size: 100000 | Parallel Sum: 2.10322e+07 | Parallel Execution Time: 0.000111299 seconds
Threads: 8 | Array Size: 100000 | Parallel Sum: 2.10322e+07 | Parallel Execution Time: 0.000109633 seconds
Threads: 9 | Array Size: 100000 | Parallel Sum: 2.10322e+07 | Parallel Execution Time: 8.9146e-05 seconds
Threads: 10 | Array Size: 100000 | Parallel Sum: 2.10322e+07 | Parallel Execution Time: 0.000155648 seconds
Threads: 11 | Array Size: 100000 | Parallel Sum: 2.10322e+07 | Parallel Execution Time: 0.000364402 seconds
Threads: 12 | Array Size: 100000 | Parallel Sum: 2.10322e+07 | Parallel Execution Time: 0.000362662 seconds
Threads: 13 | Array Size: 100000 | Parallel Sum: 2.10322e+07 | Parallel Execution Time: 0.000236437 seconds
Threads: 14 | Array Size: 100000 | Parallel Sum: 2.10322e+07 | Parallel Execution Time: 7.4843e-05 seconds
Threads: 15 | Array Size: 100000 | Parallel Sum: 2.10322e+07 | Parallel Execution Time: 6.9457e-05 seconds
Threads: 16 | Array Size: 100000 | Parallel Sum: 2.10322e+07 | Parallel Execution Time: 0.00273443 seconds
 | Array Size: 100000 | Serial Sum: 2.10322e+07 | Serial Execution Time: 0.000499303 seconds
Threads: 1 | Array Size: 1000000 | Parallel Sum: 6.66168e+08 | Parallel Execution Time: 0.00627294 seconds
Threads: 2 | Array Size: 1000000 | Parallel Sum: 6.66168e+08 | Parallel Execution Time: 0.00220412 seconds
Threads: 3 | Array Size: 1000000 | Parallel Sum: 6.66168e+08 | Parallel Execution Time: 0.00108339 seconds
Threads: 4 | Array Size: 1000000 | Parallel Sum: 6.66168e+08 | Parallel Execution Time: 0.000859362 seconds
Threads: 5 | Array Size: 1000000 | Parallel Sum: 6.66168e+08 | Parallel Execution Time: 0.000710342 seconds
Threads: 6 | Array Size: 1000000 | Parallel Sum: 6.66168e+08 | Parallel Execution Time: 0.000596608 seconds
Threads: 7 | Array Size: 1000000 | Parallel Sum: 6.66168e+08 | Parallel Execution Time: 0.000561206 seconds
Threads: 8 | Array Size: 1000000 | Parallel Sum: 6.66168e+08 | Parallel Execution Time: 0.000792623 seconds
Threads: 9 | Array Size: 1000000 | Parallel Sum: 6.66168e+08 | Parallel Execution Time: 0.000708762 seconds
Threads: 10 | Array Size: 1000000 | Parallel Sum: 6.66168e+08 | Parallel Execution Time: 0.000716863 seconds
Threads: 11 | Array Size: 1000000 | Parallel Sum: 6.66168e+08 | Parallel Execution Time: 0.00065119 seconds
Threads: 12 | Array Size: 1000000 | Parallel Sum: 6.66168e+08 | Parallel Execution Time: 0.000624963 seconds
Threads: 13 | Array Size: 1000000 | Parallel Sum: 6.66168e+08 | Parallel Execution Time: 0.000592926 seconds
Threads: 14 | Array Size: 1000000 | Parallel Sum: 6.66168e+08 | Parallel Execution Time: 0.000535496 seconds
Threads: 15 | Array Size: 1000000 | Parallel Sum: 6.66168e+08 | Parallel Execution Time: 0.000526624 seconds
Threads: 16 | Array Size: 1000000 | Parallel Sum: 6.66168e+08 | Parallel Execution Time: 0.000609815 seconds
 | Array Size: 1000000 | Serial Sum: 6.66168e+08 | Serial Execution Time: 0.00483293 seconds
Threads: 1 | Array Size: 10000000 | Parallel Sum: 2.10769e+10 | Parallel Execution Time: 0.0424277 seconds
```

Fig 6: Sample of array sum program result exported in TXT file

As Figure 6 and Figure 7 show, execution time decreases when thread counts increase. On the other hand, the dashed line represents the sequential execution to compare it with the parallel execution. Obviously, parallel processing with our proposed tool reduces execution time, especially for arrays of large sizes.

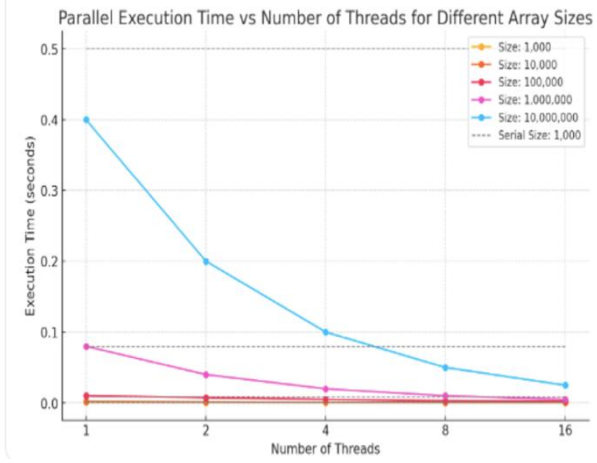


Fig 7: Parallel execution for different arrays

Take, for example, the array sizes of 1,000,000 and 10,000,000 elements; the parallel execution demonstrates a notable enhancement in performance. For the array of size 1,000,000, parallel execution with 16 threads reduces the execution time significantly from 0.0048 seconds (sequential) to 0.0006 seconds. Additionally, for the array of size 10,000,000, parallel execution decreases the time from 0.038 seconds (sequential) to 0.0052 seconds. This shows the effectiveness of parallel processing, especially in reducing execution time when the array size increases.

Table 7. Parallel execution time vs. sequential execution time over different threads number and array sizes

Thread	1,000	10,000	100,000	1,000,000	10,000,000	100,000,000	1,000,000,000
1	2.9927e-05	6.2236e-05	6.3423e-04	6.2729e-03	4.24277e-02	3.97386e-01	3.94621
2	2.02236e-04	3.6329e-05	2.45291e-04	2.24012e-03	2.08822e-02	2.05694e-01	2.02626
3	1.8571e-04	5.23559e-04	1.3624e-04	1.08339e-03	1.39409e-02	1.38824e-01	1.39303
4	1.40586e-04	2.1979e-05	9.1324e-05	8.59362e-04	1.04482e-02	1.05842e-01	1.05491
5	1.57198e-04	1.1678e-05	1.12279e-04	7.10342e-04	9.18543e-03	8.8748e-02	0.875533
6	1.48682e-04	1.7372e-05	1.03123e-04	5.96608e-04	7.43769e-03	7.56613e-02	0.742523
7	1.77571e-04	3.0487e-05	1.11299e-04	5.61206e-04	7.71035e-03	6.59139e-02	0.646288
8	1.62885e-04	6.0956e-05	1.09633e-04	7.92623e-04	5.83405e-03	5.73363e-02	0.582568
9	1.10584e-04	2.6152e-05	8.9146e-05	7.08762e-04	8.48685e-03	7.99671e-02	0.659842
10	1.17683e-04	2.9094e-05	1.55648e-04	7.16863e-04	7.96895e-03	7.68812e-02	0.658611
11	1.22161e-04	3.6182e-05	3.64402e-04	6.5119e-04	7.24048e-03	7.19435e-02	0.602151
12	9.4036e-05	2.9604e-05	3.62662e-04	6.24963e-04	6.67201e-03	6.6291e-02	0.58751
13	2.8279e-05	3.3468e-05	2.36437e-04	5.92926e-04	6.17585e-03	6.33318e-02	0.577433
14	2.8466e-05	2.8632e-05	7.4843e-05	5.35496e-04	5.78414e-03	5.78742e-02	0.553081
15	3.0488e-05	2.8311e-05	6.9457e-05	5.26624e-04	5.42363e-03	5.49328e-02	0.542912
16	2.6071e-05	2.4613e-05	2.73443e-05	6.09815e-04	5.19552e-03	5.29346e-02	0.533089
serial	5.148e-05	4.9137e-05	4.99305e-04	4.83293e-03	3.79641e-02	3.66333e-01	3.65057

Table [7] presents a comparison between parallel and sequential execution times across different numbers of threads and varying array sizes. It highlights the impact of increasing thread counts on execution time and demonstrates the performance benefits of parallel execution over sequential execution for larger problem sizes.

6.3 Array Multiplication

In this program, we measure the execution time for multiplying arrays of sizes (10×10, 100×100, and 1000×1000). The parallel execution is tested with threads (1 to 16), and sequential execution times are recorded. In fact, for smaller array sizes (10×10 and 100×100), the execution times decrease as the number of threads increases, and for larger sizes (1000×1000), parallel execution significantly reduces the time. The following Tables 8, 9, and 10, along with Figure 9, show the results in detail.

TABLE 8. Array multiplication of size 10

Threads	Size	Execution Time (seconds)
1	10x10	2.46E-05
2	10x10	1.53E-05
3	10x10	1.43E-05
4	10x10	1.43E-05
5	10x10	1.52E-05
6	10x10	1.47E-05
7	10x10	1.49E-05
8	10x10	1.46E-05
9	10x10	1.71E-05
10	10x10	2.24E-05
11	10x10	2.12E-05
12	10x10	2.10E-05
13	10x10	3.57E-06
14	10x10	4.29E-06
15	10x10	3.58E-06
16	10x10	4.40E-06
Serial	10x10	3.56E-06

TABLE 9. Array multiplication of size 100

Threads	Size	Execution Time (seconds)
1	100x100	0.0033855
2	100x100	0.00163879
3	100x100	0.00103605
4	100x100	0.00078082
5	100x100	0.00062257
6	100x100	0.00080495
7	100x100	0.00080241
8	100x100	0.00061728
9	100x100	0.00078882
10	100x100	0.00069623
11	100x100	0.00064704
12	100x100	0.00056349
13	100x100	0.00044522
14	100x100	0.00041892
15	100x100	0.00044042
16	100x100	0.00038547
Serial	100x100	0.00038692

TABLE 10. Array multiplication of size 1000

Threads	Size	Execution Time (seconds)
1	1000x1000	2.90626
2	1000x1000	1.46513
3	1000x1000	0.997864
4	1000x1000	0.752417
5	1000x1000	0.616824
6	1000x1000	0.525539
7	1000x1000	0.448733
8	1000x1000	0.394929
9	1000x1000	0.377947
10	1000x1000	0.482062
11	1000x1000	0.446682
12	1000x1000	0.428345
13	1000x1000	0.439807
14	1000x1000	0.462733
15	1000x1000	0.431919
16	1000x1000	0.477553
Serial	1000x1000	2.91394

```
Threads: 1, Size: 10x10, Execution Time: 2.4595e-05 seconds
Threads: 2, Size: 10x10, Execution Time: 1.528e-05 seconds
Threads: 3, Size: 10x10, Execution Time: 1.4327e-05 seconds
Threads: 4, Size: 10x10, Execution Time: 1.4321e-05 seconds
Threads: 5, Size: 10x10, Execution Time: 1.5213e-05 seconds
Threads: 6, Size: 10x10, Execution Time: 1.4715e-05 seconds
Threads: 7, Size: 10x10, Execution Time: 1.4929e-05 seconds
Threads: 8, Size: 10x10, Execution Time: 1.4573e-05 seconds
Threads: 9, Size: 10x10, Execution Time: 1.7059e-05 seconds
Threads: 10, Size: 10x10, Execution Time: 2.2427e-05 seconds
Threads: 11, Size: 10x10, Execution Time: 2.1154e-05 seconds
Threads: 12, Size: 10x10, Execution Time: 2.1007e-05 seconds
Threads: 13, Size: 10x10, Execution Time: 3.573e-06 seconds
Threads: 14, Size: 10x10, Execution Time: 4.294e-06 seconds
Threads: 15, Size: 10x10, Execution Time: 3.577e-06 seconds
Threads: 16, Size: 10x10, Execution Time: 4.445e-06 seconds
Serial, Size: 10x10, Execution Time: 3.559e-06 seconds
Threads: 1, Size: 100x100, Execution Time: 0.00333855 seconds
Threads: 2, Size: 100x100, Execution Time: 0.00163879 seconds
Threads: 3, Size: 100x100, Execution Time: 0.00103605 seconds
Threads: 4, Size: 100x100, Execution Time: 0.000780815 seconds
Threads: 5, Size: 100x100, Execution Time: 0.000622566 seconds
Threads: 6, Size: 100x100, Execution Time: 0.000804946 seconds
Threads: 7, Size: 100x100, Execution Time: 0.00080241 seconds
Threads: 8, Size: 100x100, Execution Time: 0.000617282 seconds
Threads: 9, Size: 100x100, Execution Time: 0.000788821 seconds
Threads: 10, Size: 100x100, Execution Time: 0.000696227 seconds
Threads: 11, Size: 100x100, Execution Time: 0.000647038 seconds
Threads: 12, Size: 100x100, Execution Time: 0.000563486 seconds
Threads: 13, Size: 100x100, Execution Time: 0.00044522 seconds
Threads: 14, Size: 100x100, Execution Time: 0.000418922 seconds
Threads: 15, Size: 100x100, Execution Time: 0.000403417 seconds
Threads: 16, Size: 100x100, Execution Time: 0.000385474 seconds
Serial, Size: 100x100, Execution Time: 0.00340814 seconds
Threads: 1, Size: 1000x1000, Execution Time: 2.90626 seconds
Threads: 2, Size: 1000x1000, Execution Time: 1.46513 seconds
Threads: 3, Size: 1000x1000, Execution Time: 0.997864 seconds
Threads: 4, Size: 1000x1000, Execution Time: 0.752417 seconds
Threads: 5, Size: 1000x1000, Execution Time: 0.616824 seconds
Threads: 6, Size: 1000x1000, Execution Time: 0.525539 seconds
Threads: 7, Size: 1000x1000, Execution Time: 0.448733 seconds
Threads: 8, Size: 1000x1000, Execution Time: 0.394929 seconds
Threads: 9, Size: 1000x1000, Execution Time: 0.577947 seconds
Threads: 10, Size: 1000x1000, Execution Time: 0.482062 seconds
Threads: 11, Size: 1000x1000, Execution Time: 0.446682 seconds
Threads: 12, Size: 1000x1000, Execution Time: 0.428345 seconds
Threads: 13, Size: 1000x1000, Execution Time: 0.439807 seconds
Threads: 14, Size: 1000x1000, Execution Time: 0.462733 seconds
Threads: 15, Size: 1000x1000, Execution Time: 0.431919 seconds
Threads: 16, Size: 1000x1000, Execution Time: 0.477553 seconds
Serial, Size: 1000x1000, Execution Time: 2.91394 seconds
```

Fig 8: Sample of array multiplication program result exported in TXT file

The results introduce the execution times for parallel processing using a number of threads and three different array sizes (10×10, 100×100, and 1000×1000). The execution times are measured in seconds, and a sequential execution time is provided for each array size. For the 10×10 array, the sequential execution time was 3.559e-06 seconds, while the parallel execution time decreased as the number of threads increased. The best performance was at 16 threads, which is 4.445e-06 seconds.

In the 100×100 array, sequential execution took 0.00340814 seconds, while parallel execution improved significantly, with execution times decreasing from 0.00333855 seconds with 1 thread to 0.000385474 seconds with 16 threads, which shows noticeable scalability.

Finally, in the 1000×1000 array, the sequential execution time is 2.91394 seconds, and parallel execution significantly reduced the execution time; it reached 0.394929 seconds using 8 threads. This shows optimal performance when a larger number of threads are used.

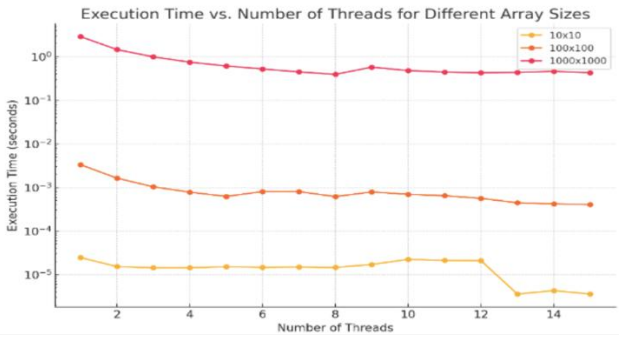


Fig 9: Execution time vs. number of threads for different array sizes

6.4 Numerical integration of x^2 on [0,1]

The numerical integration program uses the trapezoidal rule to calculate the integral of x^2 over the interval [0,1]. Execution times across different thread counts (from 1 to 16) are measured. It shows how performance change as the number of threads and trapezoids changes. For performance evaluation, these results are compared to a serial computation.

```
Threads: 1 | Number of Trapezoids: 10 | Integral: 0.335 | Parallel Execution Time: 2.32196e-06 seconds
Threads: 2 | Number of Trapezoids: 10 | Integral: 0.335 | Parallel Execution Time: 2.9203e-05 seconds
Threads: 3 | Number of Trapezoids: 10 | Integral: 0.335 | Parallel Execution Time: 3.873e-05 seconds
Threads: 4 | Number of Trapezoids: 10 | Integral: 0.335 | Parallel Execution Time: 2.3069e-05 seconds
Threads: 5 | Number of Trapezoids: 10 | Integral: 0.335 | Parallel Execution Time: 2.038e-05 seconds
Threads: 6 | Number of Trapezoids: 10 | Integral: 0.335 | Parallel Execution Time: 2.1027e-05 seconds
Threads: 7 | Number of Trapezoids: 10 | Integral: 0.335 | Parallel Execution Time: 2.2731e-05 seconds
Threads: 8 | Number of Trapezoids: 10 | Integral: 0.335 | Parallel Execution Time: 2.2965e-05 seconds
Threads: 9 | Number of Trapezoids: 10 | Integral: 0.335 | Parallel Execution Time: 2.1096e-05 seconds
Threads: 10 | Number of Trapezoids: 10 | Integral: 0.335 | Parallel Execution Time: 2.108e-05 seconds
Threads: 11 | Number of Trapezoids: 10 | Integral: 0.335 | Parallel Execution Time: 2.1144e-05 seconds
Threads: 12 | Number of Trapezoids: 10 | Integral: 0.335 | Parallel Execution Time: 2.6741e-05 seconds
Threads: 13 | Number of Trapezoids: 10 | Integral: 0.335 | Parallel Execution Time: 2.4251e-05 seconds
Threads: 14 | Number of Trapezoids: 10 | Integral: 0.335 | Parallel Execution Time: 2.768e-05 seconds
Threads: 15 | Number of Trapezoids: 10 | Integral: 0.335 | Parallel Execution Time: 2.2885e-05 seconds
Threads: 16 | Number of Trapezoids: 10 | Integral: 0.335 | Parallel Execution Time: 3.4576e-05 seconds
Serial:  | Number of Trapezoids: 10 | Integral: 0.335 | Serial Execution Time: 2.66999e-07 seconds
Threads: 1 | Number of Trapezoids: 100 | Integral: 0.33335 | Parallel Execution Time: 3.26405e-06 seconds
Threads: 2 | Number of Trapezoids: 100 | Integral: 0.33335 | Parallel Execution Time: 1.996e-06 seconds
Threads: 3 | Number of Trapezoids: 100 | Integral: 0.33335 | Parallel Execution Time: 0.000134063 seconds
Threads: 4 | Number of Trapezoids: 100 | Integral: 0.33335 | Parallel Execution Time: 2.9047e-05 seconds
Threads: 5 | Number of Trapezoids: 100 | Integral: 0.33335 | Parallel Execution Time: 1.0372e-05 seconds
Threads: 6 | Number of Trapezoids: 100 | Integral: 0.33335 | Parallel Execution Time: 1.0129e-05 seconds
Threads: 7 | Number of Trapezoids: 100 | Integral: 0.33335 | Parallel Execution Time: 9.68203e-06 seconds
Threads: 8 | Number of Trapezoids: 100 | Integral: 0.33335 | Parallel Execution Time: 3.2387e-05 seconds
Threads: 9 | Number of Trapezoids: 100 | Integral: 0.33335 | Parallel Execution Time: 1.9447e-05 seconds
Threads: 10 | Number of Trapezoids: 100 | Integral: 0.33335 | Parallel Execution Time: 1.9648e-05 seconds
Threads: 11 | Number of Trapezoids: 100 | Integral: 0.33335 | Parallel Execution Time: 2.1814e-05 seconds
Threads: 12 | Number of Trapezoids: 100 | Integral: 0.33335 | Parallel Execution Time: 2.3986e-05 seconds
Threads: 13 | Number of Trapezoids: 100 | Integral: 0.33335 | Parallel Execution Time: 2.0949e-05 seconds
Threads: 14 | Number of Trapezoids: 100 | Integral: 0.33335 | Parallel Execution Time: 2.057e-05 seconds
Threads: 15 | Number of Trapezoids: 100 | Integral: 0.33335 | Parallel Execution Time: 2.0869e-05 seconds
Threads: 16 | Number of Trapezoids: 100 | Integral: 0.33335 | Parallel Execution Time: 0.00856642 seconds
Serial:  | Number of Trapezoids: 100 | Integral: 0.33335 | Serial Execution Time: 2.06899e-06 seconds
Threads: 1 | Number of Trapezoids: 1000 | Integral: 0.333333 | Parallel Execution Time: 2.8155e-05 seconds
Threads: 2 | Number of Trapezoids: 1000 | Integral: 0.333333 | Parallel Execution Time: 0.000298802 seconds
Threads: 3 | Number of Trapezoids: 1000 | Integral: 0.333334 | Parallel Execution Time: 8.549e-05 seconds
Threads: 4 | Number of Trapezoids: 1000 | Integral: 0.333333 | Parallel Execution Time: 3.1242e-05 seconds
Threads: 5 | Number of Trapezoids: 1000 | Integral: 0.333333 | Parallel Execution Time: 1.6377e-05 seconds
Threads: 6 | Number of Trapezoids: 1000 | Integral: 0.333334 | Parallel Execution Time: 2.2873e-05 seconds
Threads: 7 | Number of Trapezoids: 1000 | Integral: 0.333334 | Parallel Execution Time: 4.26371e-05 seconds
Threads: 8 | Number of Trapezoids: 1000 | Integral: 0.333333 | Parallel Execution Time: 1.8573e-05 seconds
Threads: 9 | Number of Trapezoids: 1000 | Integral: 0.333334 | Parallel Execution Time: 1.537e-05 seconds
Threads: 10 | Number of Trapezoids: 1000 | Integral: 0.333334 | Parallel Execution Time: 1.5066e-05 seconds
Threads: 11 | Number of Trapezoids: 1000 | Integral: 0.333334 | Parallel Execution Time: 3.2232e-05 seconds
Threads: 12 | Number of Trapezoids: 1000 | Integral: 0.333334 | Parallel Execution Time: 2.2788e-05 seconds
Threads: 13 | Number of Trapezoids: 1000 | Integral: 0.333334 | Parallel Execution Time: 2.261e-05 seconds
Threads: 14 | Number of Trapezoids: 1000 | Integral: 0.333334 | Parallel Execution Time: 3.3812e-05 seconds
Threads: 15 | Number of Trapezoids: 1000 | Integral: 0.333333 | Parallel Execution Time: 0.00014703 seconds
Threads: 16 | Number of Trapezoids: 1000 | Integral: 0.333334 | Parallel Execution Time: 0.00390729 seconds
Serial:  | Number of Trapezoids: 1000 | Integral: 0.333333 | Serial Execution Time: 1.9309e-05 seconds
Threads: 1 | Number of Trapezoids: 10000 | Integral: 0.333333 | Parallel Execution Time: 0.000192107 seconds
Threads: 2 | Number of Trapezoids: 10000 | Integral: 0.333333 | Parallel Execution Time: 9.5586e-05 seconds
Threads: 3 | Number of Trapezoids: 10000 | Integral: 0.333333 | Parallel Execution Time: 0.000113049 seconds
Threads: 4 | Number of Trapezoids: 10000 | Integral: 0.333333 | Parallel Execution Time: 5.4919e-05 seconds
Threads: 5 | Number of Trapezoids: 10000 | Integral: 0.333333 | Parallel Execution Time: 4.046e-05 seconds
Threads: 6 | Number of Trapezoids: 10000 | Integral: 0.333333 | Parallel Execution Time: 3.7455e-05 seconds
Threads: 7 | Number of Trapezoids: 10000 | Integral: 0.333333 | Parallel Execution Time: 4.3432e-05 seconds
Threads: 8 | Number of Trapezoids: 10000 | Integral: 0.333333 | Parallel Execution Time: 3.9222e-05 seconds
Threads: 9 | Number of Trapezoids: 10000 | Integral: 0.333333 | Parallel Execution Time: 4.3933e-05 seconds
Threads: 10 | Number of Trapezoids: 10000 | Integral: 0.333333 | Parallel Execution Time: 4.1777e-05 seconds
Threads: 11 | Number of Trapezoids: 10000 | Integral: 0.333333 | Parallel Execution Time: 4.0396e-05 seconds
Threads: 12 | Number of Trapezoids: 10000 | Integral: 0.333333 | Parallel Execution Time: 4.3024e-05 seconds
Threads: 13 | Number of Trapezoids: 10000 | Integral: 0.333333 | Parallel Execution Time: 3.7596e-05 seconds
Threads: 14 | Number of Trapezoids: 10000 | Integral: 0.333333 | Parallel Execution Time: 4.0428e-05 seconds
Threads: 15 | Number of Trapezoids: 10000 | Integral: 0.333333 | Parallel Execution Time: 3.5853e-05 seconds
Threads: 16 | Number of Trapezoids: 10000 | Integral: 0.333333 | Parallel Execution Time: 3.5209e-05 seconds
Serial:  | Number of Trapezoids: 10000 | Integral: 0.333333 | Serial Execution Time: 0.000192519 seconds
Threads: 1 | Number of Trapezoids: 100000 | Integral: 0.333333 | Parallel Execution Time: 0.00191792 seconds
```

Fig 10: A sample of numerical integration results

As Figure 10 and Figure 11 show, the execution time of the program through different numbers of threads (1 to 16) and numbers of trapezoids (10, 100, 1,000, 1,000,000, 10,000,000, 100,000,000).

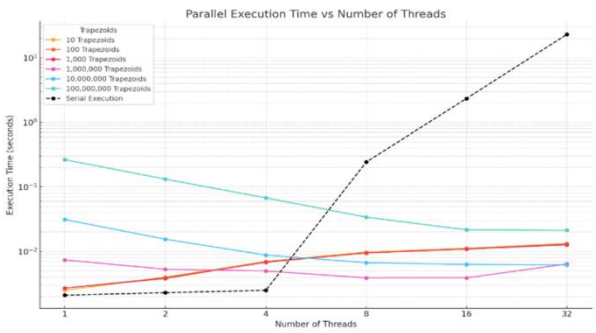


Fig 11: The numerical integration program computing the integral of x^2 over the interval [0,1] execution times across different thread counts

It's clear that as the number of threads increases, the execution time generally decreases, which means better performance with our proposed translation tool. Sequential execution takes longer time, as expected. As the number of trapezoids increases, longer execution time is taken. We can see that for larger problem sizes, more threads lead to better performance. The results show that in all cases, parallel execution surpasses serial execution, especially when the problem size increases. However, the difference is most noticeable for large numbers of trapezoids (e.g., 100,000 and more), where parallel execution can be clearly faster than serial execution. Table 11 shows speedup; we found higher speedup with a higher number of threads because, in such values, the effect becomes clearer.

Speedup =Serial Execution Time/parallel Execution Time

Table 11. Speed up at size of 1000,000

Program	Threads	Speed Up
Dot Product	10	4.560
	12	5.391
	14	5.074
Array Sum	10	6.745
	12	7.730
	14	9.032
Array Multiplication	10	6.046
	12	6.803
	14	6.300
Numerical integration of x^2 on [0,1]	10	6.97
	12	8.29
	14	9.66

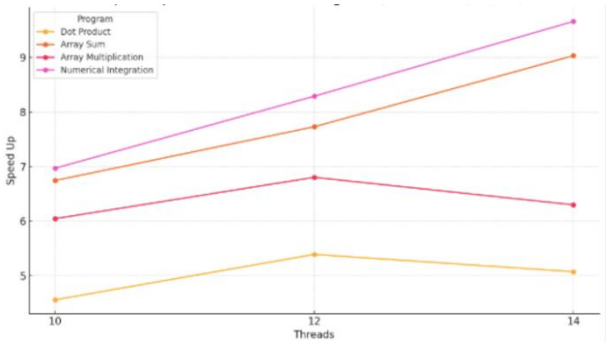


Fig 12: Speed up for threads 10,12,14 in the different four programs

In some cases, we notice that speedup drops when there are more threads, as shown in Figure 12. This is due to overheads such as thread management, computation, and synchronization. Although this drop occurs, parallelism is still efficient because it still minimizes execution time when compared to sequential execution. In addition to the experiments demonstrating the efficiency of our tool in reducing execution time and enhancing performance. After obtaining these results, it is necessary to relate them to previous studies in the field. One notable example is the work of Sardar & Faizabadi [16], who examined the performance of manual OpenMP coding versus the Pluto auto-parallelization tool on a set of five numeric and graph algorithms. While their study provided useful insights into the relative strengths of hand-tuned and automated approaches, our work differs in scope and methodology. Specifically, our approach emphasizes a fully automated single-model translation (OpenMP) and provides quantitative performance evaluations on larger datasets and reduction-based problems. Table 12 presents a comparison between our work and that of Sardar & Faizabadi [16]. As a common benchmark of matrix multiplication, a comparison shows that for the 1000×1000 matrix case the authors' method [16] achieved ~2.115× speedup, while our approach reached ~6.086× as shown in Figure 13.

Table 12. Comparison between our work and Sardar & Faizabadi (2019)

Aspect	Our work	Sardar & Faizabadi (2019, OpenMP + PLUTO)
Benchmarks	Dot Product, Array Sum, Matrix Multiplication, Trapezoidal Integration	Matrix Multiplication, Matrix Vector Multiplication, Matrix Addition, Algorithms
Programming Language	C++ (sequential translated automatically with OpenMP)	C (coded sequentially, then parallelized manually and via Pluto
Parallelization Approach	Fully automatic translation of sequential C++ to OpenMP	Two paths: (1) Hand parallelization using OpenMP, (2) Auto parallelization using Pluto

Datasets	Large-scale: vectors up to 10M elements, matrices up to 1000×1000, trapezoid integration up to 100M intervals	Randomly generated datasets with sizes (matrices: 200–4000, vectors: 2000–20000, graphs: n=10–1000)
Output	Quantitative results (execution times and speedups up to 9.66×)	Manual analysis : Decision Table showing where OpenMP or Pluto performed better

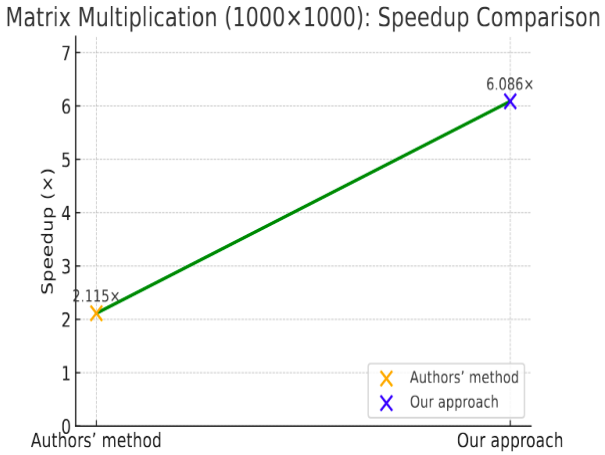


Fig 13: Speedup comparison for 1000×1000 matrix multiplication

7. CONCLUSION

In this study, we presented a novel tool for the automatic transformation of sequential C++ code into parallel code, addressing the increasing need for efficient code execution in the era of rapid processor advancements. By automating the parallelization process, this tool effectively reduces the burden on programmers, mitigates the potential for errors, and ensures better utilization of modern multi-core processors, which saves time and effort. The inclusion of dependency analysis and identification of parallelizable regions ensures that only suitable sections of code are parallelized, leading to optimized execution. The results of testing demonstrate significant improvements in code performance and reduced implementation time, validating the efficacy of the proposed approach. Overall, the proposed tool contributes to enhancing computational efficiency and saves time for both users and developers.

For future work, extending the tool's capability to support other common and widely used programming languages, such as Python or Java, would be very useful. Additionally, further work could also focus on integrating support for additional parallel programming models, such as CUDA for GPU acceleration.

8. REFERENCES

[1] Anurag, A. V. P., Chauhan, A., and Yadav, M. K. 2022. Performance analysis of serial computing vs. parallel computing in MPI environment. International Journal of Advanced Research in Science, Communication and Technology (IJARSCT), 2(7).

[2] Rastogi, S. and Zaheer, H. 2016. Significance of parallel computation over serial computation. In Proceedings of the 2016 International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT). 57–61.

[3] Gabriel, E., et al. 2004. Open MPI: Goals, concept, and design of a next generation MPI implementation. In Recent Advances in Parallel Virtual Machine and Message Passing Interface.

[4] Kranzlmüller, D., Kacsuk, P., and Dongarra, J., Eds. 2004. Lecture Notes in Computer Science, Vol. 3241. Springer, Berlin, Germany.

[5] Gropp, W., Lusk, E., Doss, N., and Skjellum, A. 1996. A high-performance, portable implementation of the MPI message passing interface standard. Parallel Computing 22, 6, 789–828.

[6] Mehtre, V. V. and Kumar, V. 2019. Review paper on C++. IRE Journal 3, 6, 48.

[7] Aljas, J. M., Amores, H. A., and Lincopinis, D. R. 2023. An overview on C++ programming language. Western Mindanao State University, Pagadian City, Philippines.

[8] Guan, W. 2023. Research on how to optimize data structures with C++ language. Academic Journal of Computing and Information Science 6, 1, 52–56.

[9] Bazoukis, K. 2019. Comparing maintainability, reliability and efficiency quality attributes on C, classic C++, and modern C++ languages. Presented at Conference, Feb. 2019.

[10] Singh, I. 2013. Review on parallel and distributed computing. Scholars Journal of Engineering and Technology (SJET) 1, 4, 218–225.

[11] Muddukrishna, A. 2016. Improving OpenMP productivity with data locality optimizations and high-resolution performance analysis. PhD Dissertation. Department of ICT, KTH Royal Institute of Technology, Stockholm, Sweden.

[12] Nikolic, G., Dimitrijević, B., Nikolic, T., and Stojcev, M. 2022. Fifty years of microprocessor evolution: From single CPU to multicore and manycore systems. Facta Universitatis Series: Electronics and Energetics 35, 2, 155–186.

[13] Kumar, R., Rao, K. V. R., Dakshinamurthy, A., and Chatterjee, D. 2013. Evolution of processors and its implication in data computation capability. In Proceedings of the 10th Biennial International Conference and Exposition. 281.

[14] Shima, M. 2005. The birth, evolution and future of the microprocessor. In Proceedings of the 5th International Conference on Computer and Information Technology (CIT'05), Shanghai, China.

[15] Herhut, S., Hudson, R. L., Shpeisman, T., and Sreeram, J. 2012. Parallel programming for the web. In Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism.

[16] Sardar, S. and Faizabadi, S. 2019. Parallelization and

analysis of selected numerical algorithms using OpenMP and Pluto on symmetric multiprocessing machine. Data Technologies and Applications 53, 1, 90–106.

[17] Kovačević, D., Stanojević, M., Marinković, V., and Popović, M. 2013. A solution for automatic parallelization of sequential assembly code. Serbian Journal of Electrical Engineering 10, 1, 91–101.

[18] Bluemke, I. and Fugas, J. 2010. C code parallelization with ParaGraph. In Proceedings of Conference. 163–166.

[19] Kalender, M. E., Mergenci, C., and Ozturk, O. 2014. AutopaR: An automatic parallelization tool for recursive calls. In Proceedings of Conference, Bilkent University, Ankara, Turkey.

[20] Mathews, M. and Abraham, J. P. 2016. Automatic code parallelization with OpenMP task constructs. Department of Computer Science and Engineering, Mar Athanasius College of Engineering, Kochi, India.

[21] Jindal, A., Jindal, N., and Sethia, D. 2012. Automated tool to generate parallel CUDA code from a serial C code. International Journal of Computer Applications 50, 8, 15–22.

[22] Algarni, A., Alofi, A., and Eassa, F. 2021. Parallelization technique using hybrid programming model. International Journal of Advanced Computer Science and Applications (IJACSA) 12, 2.