Design and Programming for Multicore machines: An Empirical study on time and effort required by programmer

Vinay T R[1*](0000-0001-7038-1943). *Satish E G*[2] and *Megha J*[3]

[1]Ramaiah Institute of Technology, Bengaluru, India, tr.vinay@gmail.com
[2]Nitte Meenakshi Institute of Technology, Bengaluru, India
[3]Ramaiah Institute of Technology, Bengaluru, India.

Abstract. As the demand for high-performance computing continues to surge, harnessing the full potential of multicore architectures has become paramount. This paper explores a pragmatic approach to transition from sequential to parallel programming, capitalizing on the computational prowess of modern hardware systems. Recognizing the challenges of enforcing parallelism in early software development phases, we advocate for a focus on the implementation stage, where architects, designers, and developers can seamlessly introduce parallel constructs while preserving software integrity. To facilitate this paradigm shift, we introduce the "SDLC model with Parallel Constructs," a modified Software Development Life Cycle (SDLC) framework comprising additional phases: "Parallel Constructs" and "Test Parallel Constructs." This model empowers development teams to integrate parallel computing efficiently, enhancing performance and maintaining a structured development process. Our observations reveal intriguing dynamics. Initially, the single-threaded program outperforms its parallel counterpart for smaller datasets, but as data sizes grow, the parallel version demonstrates superior performance. We underscore the pivotal role of available CPU cores and task partitioning in determining efficiency. Our analysis also evaluates the programmer's effort, measured by lines of code, needed for the transition. Leveraging OpenMP constructs streamlines this transition, reducing programming complexity.

Keywords: Multicore machine, Parallel Programming, Software development life cycle, Effort, Parallel Constructs.

1. Introduction

Traditionally, problem-solving through programming has followed a linear path: design a solution, implement it as a sequence of operations or statements executed from START to END. This approach was well-suited to the computing hardware of the past decade, which consisted primarily of single-core machines. These machines executed a single instruction cycle per clock cycle, allowing for single-threaded execution at any given time.

However, as software applications grew in complexity, demanding increased performance and the ability to perform multiple tasks simultaneously, hardware designers and architects faced a significant challenge. They responded by packing more transistors onto a single chip, a trend famously recognized as Moore's Law. Advances in nanotechnology further reduced transistor size, facilitating the integration of even more transistors onto a

*Corresponding author: tr.vinay@gmail.com

© The Authors, published by EDP Sciences. This is an open access article distributed under the terms of the Creative Commons Attribution License 4.0 (https://creativecommons.org/licenses/by/4.0/).

single chip. Nevertheless, a point was reached where adding more transistors became impractical, resulting in excessive power consumption and limited transistor lifespan.

This technological impasse prompted a shift in focus toward a new paradigm: the integration of multiple cores on a single die, giving rise to multicore machines. The last single-core machine was released in 2013, marking the ubiquity of multicore machines across a wide range of electronic devices, from embedded systems to smartphones, workstations, and data servers.

While context switching between application threads was introduced in single-core systems to enhance application performance, it proved insufficient for handling the demands of increasingly complex applications. Single-core machines could execute only one thread at a time, and this limitation became apparent when running large applications. True parallelism was lacking, and other threads had to wait their turn, exacerbated by faster components elsewhere in the system, causing a bottleneck in processing speed. However, it's important to note that single-core processors still offer advantages. They tend to have lower power consumption and manufacturing costs. They are well-suited for embedded systems designed for specific tasks.

To address the performance requirements of modern software applications, such as big data analysis and weather forecasting, innovative approaches are essential. These include designing more efficient algorithms, introducing domain-specific programming languages, and developing domain-specific chips or systems.

In this paper, we delve into strategies for fully harnessing the power of multicore machines from a software development [6] perspective. We recognize that it may not always be practical to force software analysts, designers, and architects to think exclusively in terms of parallel or concurrent execution, as it may lead to suboptimal designs. Instead, we explore the use of compilers, open-source tools, and software libraries that provide directives, APIs, and packages for parallel computation [7]. By leveraging these resources and making minor adjustments to traditional sequential programs, we can optimize program performance [8] and make the most of multicore systems.

2. Related work

In recent, experiments have been carried-out to evaluate the performance of multicore machines and GPU's as well. The measurable variables included are CPU cores, Number of threads, execution time, shared resources of CPU and parallel program constructs itself. The Table 1 gives an overview of some of the studies and summarizes it.

Table 1: Summary of Parallel Methodology adopted by researchers.

Sl. No.	**Author and Title**	**Methodology/ variables studied**	**Observation**
1	Woochang Shin: “Performance Comparison of Parallel Programming Frameworks in Digital Image Transformation (DIT)” [1]	Considered DIT problem and applied OpenMP, PPL, OpenCL and CUDA parallel programming frameworks and measured execution-time of all the methods.	The author’s conclusion is based on only specific DIT problem. Effort by the software programmer is calculated only by measuring LoC.
2	Bob Kuhn et.all, “OpenMP versus Threading in C/C++”, White paper. [2]	The authors considered Genehunter application , analyzed, applied and evaluated OpenMP and p-threads. They concluded OpenMP is better than p-threads.	The author’s conclusion is specific to Genehunter application and cannot be generalized.
3	Ruidong Gu et. All., “A Comparative Study of Parallel Programming Frameworks for Distributed GPU Applications”. [3]	The authors studied memory utilization, synchronization support, execution model and GPU support for MPI, charm++ and other parallel programming frameworks.	The observation from their study points out that different applications perform better with different programming frameworks.
4	Jascha Schewtschenko, “Implementation of Parallelization OpenMP, P-Threads and MPI”. [4]	Here the study on Parallelization of tasks by employing different parallel constructs is carried-out. Different features available in OpenMP, P-threads and MPI are demonstrated by taking a suitable example.	The study is carried out purely in terms of implementation phase.
5	Henrik Swahn, “Pthreads and OpenMP A performance and productivity study”. [5]	Study on employing P-threads and openMP on Matrix, Quicksort is carried-out. The performance of these programs on multicore machine is observed.	Observed that while employing recursions in implementation part. It is expensive.

3. Proposed Methodology

During the software development phase, the primary emphasis lies on comprehending the problem, collecting requirements, and subsequently, the design phase. At this juncture, it's often impractical to mandate architects and designers to immediately embrace parallel computation concepts. Instead, we advocate for shifting this focus to a later stage: the implementation phase. To facilitate this transition, we've redefined the traditional Software Development Life Cycle (SDLC) model, introducing two additional phases—namely, "Parallel Constructs" and "Test Parallel Constructs." This modified model is referred to as the "SDLC model with Parallel Constructs," and Figure 1 below illustrates this updated SDLC framework.

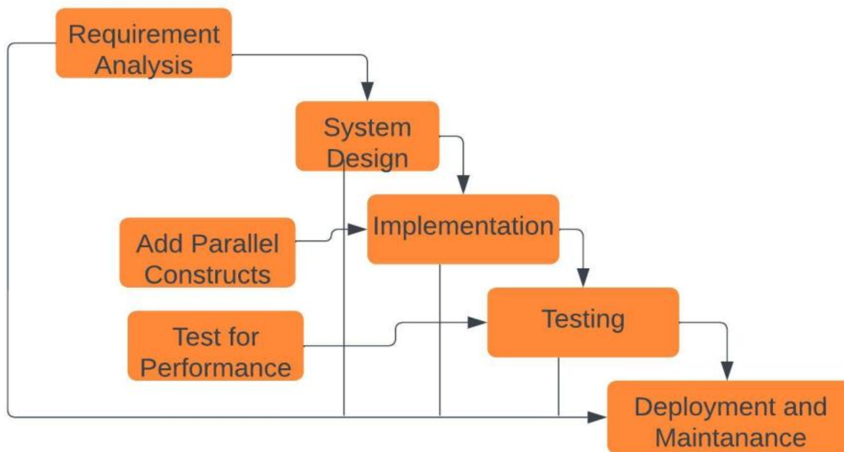

Figure 1: SDLC model with Parallel constructs.

Under the adapted SDLC model, software development models and methodologies can be suitably revised to incorporate parallel constructs within the implementation and testing phases. During the testing phase, our objectives expand beyond functional testing. We aim to measure and quantify the efficiency gains resulting from the integration of parallel constructs. Additionally, we rigorously assess for any potential memory leaks that may arise.

Measuring the effort required by software engineers to transition from a sequential algorithmic design to a parallel execution paradigm is paramount. Two key metrics for this assessment are the Number of Lines of Code (LOC) and the time spent by individuals on inserting parallel constructs, considered a complex and low-level language task.

3.1 Methodology Steps

The overarching steps of this methodology are presented below:

I. For any software project, adhere to the standard SDLC process, encompassing problem understanding, requirement collection, and the design of the solution.

II. Document the corresponding algorithm as the proposed solution for the given requirements.

III. During the implementation phase, integrate parallel constructs tailored to specific domains or those with which software developers are proficient. These constructs can be sourced from various organizations and open-source communities.

IV. Conduct comprehensive performance assessments to gauge the effectiveness of the application's parallelized components.

V. It's crucial to note that, within this methodology, modifications are made exclusively during the implementation phase. This ensures that the correctness and completeness of the software solution are maintained throughout the development cycle.

By following these methodological steps, software development teams can efficiently incorporate parallel computing concepts into their projects, optimizing application performance while preserving software integrity and completeness.

4. Case study:

An empirical study was conducted to investigate the impact of integrating parallel constructs into a sequential program. The selected program for this study is scalar multiplication, a classic example involving sequential array operations and multiplication.

```
// Scalar Multiplication Program
#include<stdio.h>
#include <time.h>
#define n 10000000
int main() {
long long int i;
float a = 2.0;
static float x[n];
static float y[n];
clock_t start, end;
for (i = 0; i < n; i++) {
       x[i] = 1.0;
       y[i] = 2.0;
}
start = clock();
for (i = 0; i < n; i++) {
       y[i] = a * x[i] + y[i];
}
end = clock();
double duration = ((double)end - start)/CLOCKS_PER_SEC;
printf("Time taken to execute in seconds : %f",
duration);
}
```

Figure 2: code snippet of scalar multiplication.

1. The selected hardware machine, with the following configuration—model name: Intel(R) Core(TM) i9-10900X CPU @ 3.70GHz with 20 cores—was used for execution.

2. The program depicted in Figure 2 was executed as a single-threaded program, and the execution time was recorded. This process was repeated while varying the size of the array in each run.

3. Subsequently, for the program shown in Figure 2, we introduced the parallel construct-OpenMP, and executed it with arrays of different sizes. This modified program is presented in Figure 3. It was executed as a parallel-threaded program using OpenMP constructs, with the number of threads restricted to twenty due to our hardware system's limitation of having only twenty cores.

```
// Scalar Multiplication Program
#include <omp.h>
#include<stdio.h>
#define n 1000000
int main() {
long long int i;
float a = 2.0;
static float x[n];
static float y[n];
double start_time, end_time;
for (i = 0; i < n; i++) {
       x[i] = 1.0;
       y[i] = 2.0;
}
start_time = omp_get_wtime();
#pragma omp parallel for private(i)
for (i = 0; i < n; i++) {
       y[i] = a * x[i] + y[i];
}
end_time = omp_get_wtime();
printf("SAXPY Time: %f\n", end_time - start_time);
}
```

Figure 3: code snippet of scalar multiplication including OpenMP construct.

The execution times for different array sizes are recorded and presented in Table 2.

Table 2: Execution time of sequential and parallel program using OpenMP

Sl. No.	Array size (n)	Execution time T(O) in secs. of original program	Execution time T(N) in secs. of modified program with OpenMP construct
1	100	0.000003	0.000745
2	1000	0.000008	0.000830
3	10000	0.000061	0.000868
4	100000	0.000590	0.000875
5	1000000	0.004166	0.001526
6	10000000	0.016257	0.008155
7	100000000	0.159479	0.066576

Figure 4 displays the data from Table 2 in graphical form. The scale factor is le8.

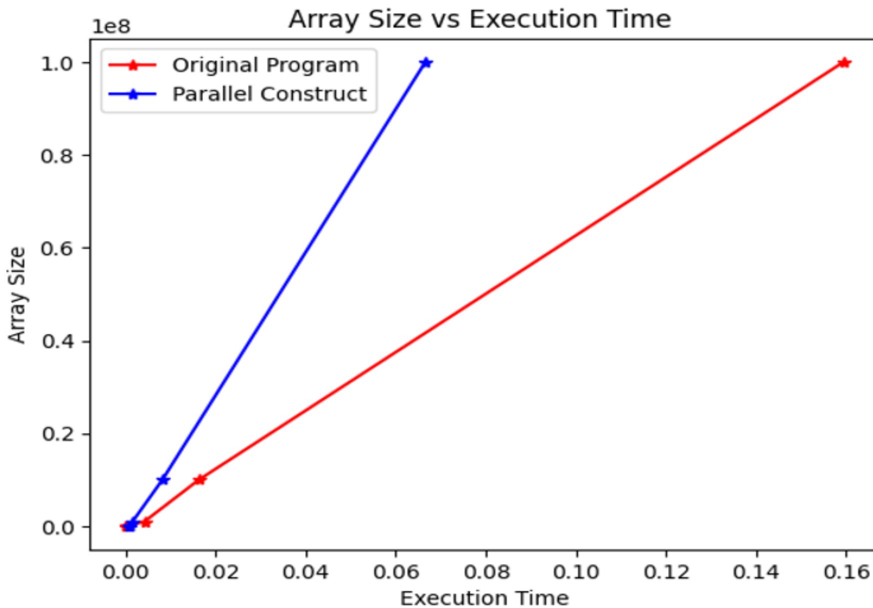

Figure 4: Comparison of Execution Times: Original Program vs. Parallel Construct Program.

The speed-up in each case is calculated using Amdahl's law and presented in Table 3. Speed-up is determined by dividing the time taken by the original program (T(O)) by the time taken when OpenMP is added to the original program (T(N)).

Table 3: Speedup achieved by varying the array sizes.

Sl. No.	Array Size (n)	T(O) /T(N)	Remarks
1	100	0.00402	When the array size is small, the original program exhibits shorter execution times compared to its parallel counterpart.
2	1000	0.00963	
3	10000	0.0702	
4	100000	0.674	
5	1000000	2.73	As the array size increases, the program with parallel constructs demonstrates improved execution time.
6	10000000	1.993	
7	100000000	2.395	

Table 4: Comparison of Lines of Code

Sl. No.	Original Program's: Lines of Code	Parallel Construct Program's: Lines of Code	Programmer's effort
1	21	22	Here in this case, the programmer has to add only one line as there is only one for-loop.

Observations:

I. Initially, the execution time of the single-threaded program outperforms the parallel version when the array size is smaller. However, as the array size increases, the parallel construct of the program exhibits better performance.

II. Generally, when the array size is substantially large, the parallel version of the program demonstrates a significant speedup, being around fifty times (according to 7th case where n = 100000000) faster than the single-threaded program.

III. The number of parallel threads executed is constrained by the number of available CPU cores in the hardware system.

IV. The efficiency of the parallel execution also relies on how the programmer partitions the task into multiple sub-tasks and assigns them to parallel threads. When sub-tasks have uniform workloads, all threads may complete their tasks simultaneously, fully utilizing the available CPU cores. However, imbalanced workloads can lead to some threads finishing earlier, leaving certain cores underutilized.

V. The parallel version of the program effectively leverages all available CPU cores within the hardware system, in contrast to the single-threaded program.

VI. The additional effort required by the programmer is quantified by comparing the lines of code between the single-threaded and parallel-threaded implementations.

VII. In this case, the OpenMP construct is employed to transition from a single-threaded execution model to a multi-threaded parallel execution model. This approach minimizes the programming effort required, streamlining the parallelization process.

These observations provide valuable insights into the performance characteristics and trade-offs between the sequential and parallel implementations of your program. Including specific speedup factors and details about the hardware configuration would enhance the completeness of your findings.

5. Conclusion

In this paper, we embarked on a journey to harness the untapped power of multicore computing architectures from a software development perspective. As software applications continue to grow in complexity and demand heightened performance, the need to embrace parallel computing methodologies becomes increasingly critical. However, transitioning from sequential to parallel programming is not without its challenges. To facilitate this paradigm shift, we introduced the "SDLC model with Parallel Constructs," a modified Software Development Life Cycle model. This model introduces two additional phases: "Parallel Constructs" and "Test Parallel Constructs." By doing so, it enables software development teams to seamlessly integrate parallel computing into their projects while maintaining a structured and well-defined development process by following the methodologies and insights presented in this paper, software development teams can unlock the latent potential of multicore systems, achieving enhanced performance, efficiency, and competitiveness in a rapidly evolving computing landscape.

References

[1] Woochang Shin et all, “Performance Comparison of Parallel Programming Frameworks in Digital Image Transformation (DIT)”. https://www.earticle.net/Article/A361091. (2019).

[2] Bob Kuhn et.all, “OpenMP versus Threading in C/C++”, 사Concurrency Practice and Experience 12(12):1165-1176, DOI:10.1002/1096-9128(200010)12:12<1165::AID-CPE529>3.0.CO;2-L

[3] Ruidong Gu et. All., “A Comparative Study of Parallel Programming Frameworks, (2000), for Distributed GPU Applications”, CF '19: Proceedings of the 16th ACM International Conference on Computing FrontiersApril 2019Pages 268 273, https://doi.org/10.1145/3310273.3323071, (2019)

[4] Jascha Schewtschenko, “Implementation of Parallelization OpenMP, PThreads and MPI” http://icg.port.ac.uk/~schewtsj/hpc2018/lecture-implemention.pdf. (2018).

[5] Henrik Swahn, “Pthreads and OpenMP A performance and productivity study” (2016), http://www.diva-portal.org/smash/record.jsf?pid=diva2%3A944063&dswid=3598

[6]Vinay T R and A. A. Chikkamannur, "A methodology for migration of software from single-core to multi-core machine," 2016 International Conference on Computation System and Information Technology for Sustainable Solutions (CSITSS), Bengaluru, India, 2016, pp. 367-369, doi: 10.1109/CSITSS.2016.7779388, (2016).

[7]Vinay, T.R., Chikkamannur, A.A. A Novel Methodology to Restructure Legacy Application onto Micro-Service-Based Architecture System. Emerging Research in Computing, Information, Communication and Applications. Lecture Notes in Electrical Engineering, vol 790. Springer, Singapore. https://doi.org/10.1007/978-981-16-1342-5_39, (2022).

[8] Vinay T R, Chikkamannur A A, A Semi-Automatic Transformational Technique for Transforming Single Threaded Program Into Multi-Threaded Program. https://ijarcce.com/wp-content/uploads/2022/01/IJARCCE.2021.101251.pdf, (2022).