arXiv:submit/6220794 [cs.DC] 21 Feb 2025

What Every Computer Scientist Needs to Know About Parallelization

Temitayo Adefemi

Abstract—Parallelization has become a cornerstone of modern computing, influencing everything from high-performance supercomputers to everyday mobile devices. This paper presents a comprehensive guide on the fundamentals of parallelization that every computer scientist should know, beginning with a historical perspective that traces the evolution from early theoretical models such as PRAM and BSP to today's advanced multicore and heterogeneous architectures. We explore essential theoretical frameworks, practical paradigms, and synchronization mechanisms while discussing implementation strategies using processes, threads, and modern models like the Actor framework. Additionally, we examine how hardware components—including CPUs, caches, memory, and accelerators interact with software to impact performance, scalability, and load balancing. This work demystifies parallel programming by integrating historical context, theoretical underpinnings, and practical case studies. It equips readers with the tools to design, optimize, and troubleshoot parallel applications in an increasingly concurrent computing landscape.

Index Terms—Parallel Computing, Parallelization, Concurrency, Synchronization, Multi-Core Processing, Actor Model, Load Balancing, High-Performance Computing (HPC), Distributed Systems, Memory Hierarchy

1 Introduction

Since IBM's 1958 research memo, parallel programming has been fundamental to computing, yet its perception has often been skewed. Many computer scientists have historically viewed parallelization as an advanced, specialized concept reserved for high-performance computing (HPC) environments [1]. This could not be further from the truth. Today, parallelization is so deeply embedded in modern computing that nearly every system relies on it in some form. From supercomputers to smartphones, cloud platforms to everyday appliances, parallel processing is everywhere [2].

Despite its ubiquity, parallel programming remains underutilized by many developers, often because of the misconception that it requires esoteric knowledge or is only relevant for those working on massive-scale computations. In reality, mastering parallelization is crucial—not just for optimizing HPC workloads but for improving efficiency across all computing domains, including software development, machine learning, and embedded systems [3].

Breaking down the barriers to understanding parallel computing is crucial to bridge this gap. This paper aims to demystify parallel computing, providing a comprehensive understanding of its principles and applications. We will explore the key factors influencing parallel programs, including parallel paradigms, hardware considerations, memory hierarchies, and cache behavior [4]. We will also examine the trade-offs involved, the problems that benefit from parallelization, and the libraries and tools that make it accessible to developers [5]. By the end of this paper, readers will not only grasp the abstract concepts governing parallel computing but also gain the practical knowledge to implement efficient, scalable parallel programs.

• *T. Adefemi is with the University of Edinburgh.*
E-mail: `T.M.Adefemi@sms.ed.ac.uk`

2 What is Parallel Computing

It is crucial to explain parallel computing first to understand it. In serial computing, a problem is broken into a discrete series of instructions executed sequentially—one after another—on a single processor. Only one instruction may be executed at any given moment, meaning tasks are processed strictly linearly. While this model is simple and effective for many applications, it can become inefficient when dealing with large-scale computational problems that require significant processing power [6].

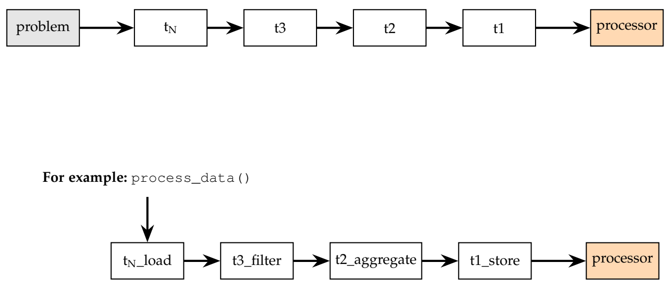

Fig. 1: A schematic showing how a problem is broken down into instructions and executed by a processor. An example function `process_data()` is also shown.

Parallel computing, in contrast, is the simultaneous use of multiple computing resources to solve a computational problem. The problem is divided into discrete parts that can be solved concurrently, and each part is further decomposed into a series of instructions. These instructions are executed simultaneously across different processors, leveraging the computational power of multi-core CPUs, GPUs, or distributed computing systems [7]. An overall control/coordination mechanism maintains accuracy and synchronization, ensuring that each task progresses as intended. However, the complexity of these mechanisms varies based on the problem being solved and the parallelization paradigm used—ranging from shared memory models with synchronization primitives to distributed computing models that use message passing for inter-process communication [8].

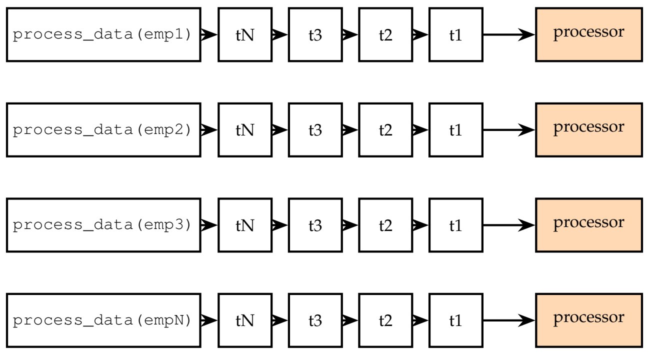

Fig. 2: Parallel execution of `process_data()` for multiple inputs. Each input (e.g., `emp1`) is processed separately in parallel.

Based on this definition, it becomes evident why a coordination mechanism is essential to ensure the program behaves as intended. These mechanisms operate at multiple levels of the parallel computing stack, from software-level parallel programming models to hardware-level synchronization protocols. At the lowest level, threads use locks, mutexes, and semaphores to prevent simultaneous updates to shared resources, which could lead to race conditions and data inconsistencies [9]. Higher up the stack, message-passing mechanisms—such as those implemented in MPI (Message Passing Interface)—allow distributed processes to communicate effectively and synchronize their execution [10].

Why parallelize a program? The primary motivation is to enhance speed and efficiency. Modern computers typically feature multi-core processors, GPUs, and high-performance distributed systems, making it logical to share computational loads across multiple processing units rather than allowing resources to remain idle. By distributing workloads efficiently, programs can significantly reduce execution times, achieve higher throughput, and better use available hardware resources [11].

3 Historical Perspective

In order to give an overview of parallelization, it is important to understand its roots. Parallelization can be traced back to the early days of computing in the 1950s when researchers first recognized the need to process multiple tasks concurrently, which brought forward the IBM research memo, as discussed earlier. The early pioneers began exploring methods to execute several operations simultaneously, which set the stage for modern parallel computing [12].

As computer architectures advanced in the 1960s and 1970s, so did the ideas surrounding parallel computation. Researchers developed early models such as the Parallel Random Access Machine (PRAM), which provided a simplified abstraction for understanding how multiple processors could work together on a single problem. However, PRAM had its limitations as it did not consider the complexities surrounding parallel computing. During this era, large mainframe systems and specialized supercomputers were built with parallel processing capabilities, albeit within very controlled environments, paving the way for more ambitious applications in scientific and engineering domains [13].

The 1980s witnessed significant progress in hardware design and algorithm development, leading to the advent of dedicated parallel machines. This period saw the emergence of vector processors and early multiprocessor systems that could handle more complex, data-intensive tasks. Academic research and government-funded projects contributed to a deeper understanding of synchronization, load balancing, and the challenges of distributed memory, which are still crucial to parallel computing today [14].

With the advent of microprocessors in the 1990s, parallelization transitioned from specialized supercomputers to more widely available commodity hardware. The introduction of multi-core processors revolutionized the computing landscape, making parallel processing accessible to a broader audience. This shift was accompanied by the development of robust programming models and standards, such as MPI and OpenMP, which allowed developers to exploit parallelism more easily in everyday applications [15].

Today, parallelization is a fundamental aspect of nearly every computing system, from high-performance clusters to smartphones. The historical evolution from theoretical models and expensive hardware to ubiquitous, multi-core devices underscores the transformative impact of parallel computing. Modern computer scientists benefit from decades of research and practical advancements that have made parallel programming a specialized skill and an essential component of practical software development across diverse fields [16].

4 Theories Governing Parallel Computing

4.1 The PRAM Model

One of the earliest and most influential theoretical frameworks in parallel computing is the *Parallel Random Access Machine (PRAM)* model, which was mentioned in the historical perspective section. PRAM provides an idealized abstraction of parallel computation, where multiple processors operate synchronously and share a standard memory. The simplicity of PRAM in capturing parallelism has made it a widely used model for designing parallel algorithms [17]. However, this abstraction also introduces challenges, as full synchronization and shared memory access are costly in practical implementations [18].

- **Exclusive Read Exclusive Write (EREW):** No simultaneous reading or writing of the same memory cell.

- **Concurrent Read Exclusive Write (CREW):** Multiple processors can read the same cell, but only one may write.
- **Concurrent Read Concurrent Write (CRCW):** Both reading and writing can be performed concurrently, with various rules to resolve write conflicts.

While PRAM is primarily a theoretical construct, it has motivated the development of specialized hardware and emulation techniques. Some researchers have attempted to map PRAM models onto modern many-core processors, such as Intel's Single-chip Cloud Computer (SCC), to bridge the gap between theory and real-world implementations [19]. However, practical constraints, such as memory bandwidth limitations and synchronization overheads, limit direct PRAM implementations [20].

Despite its limitations, PRAM remains relevant in algorithm design and analysis, often as a stepping stone for developing practical parallel algorithms [21].

4.2 Bulk Synchronous Parallel (BSP) Model

Developed by Leslie Valiant, the *BSP model* introduces a more realistic abstraction that segments computation into a series of supersteps. Each superstep consists of three phases:

1) **Local Computation:** Processors perform computations using local data.
2) **Communication:** Data is exchanged between processors.
3) **Barrier Synchronization:** Processors wait until all have completed the current superstep before proceeding.

BSP explicitly captures the cost of communication and synchronization, making it a valuable tool for predicting and optimizing performance in real-world parallel systems [22].

4.3 The LogP Model

To further refine our understanding of parallel execution, the *LogP* model introduces four key parameters that account for realistic communication and computation costs:

- **L (Latency):** The delay incurred in communicating a message from one processor to another.
- **o (Overhead):** The time a processor spends sending or receiving a message.
- **g (Gap):** The minimum time interval between consecutive message transmissions.
- **P (Processors):** The number of processors in the system.

LogP provides a practical framework for analyzing parallel execution costs, particularly in distributed-memory architectures. It addresses PRAM's shortcomings by considering real-world constraints such as network communication overhead and memory access latency [23].

4.4 Scalability and Speedup: Amdahl's Law and Gustafson's Law

4.4.1 *a. Amdahl's Law*

Amdahl's Law is a foundational principle that quantifies the potential speedup of a parallel program. It states that if a fraction f of a task is inherently serial, the maximum speedup S achievable with P processors is limited by:

$$S(P) = \frac{1}{f + \frac{1-f}{P}}$$

This Law underscores a critical limitation: even if most of the computation can be parallelized, the serial portion restricts overall performance. As P approaches infinity, the speedup converges to $\frac{1}{f}$ [17].

This is critical for understanding and analyzing parallel programs and how parallelization can impact their potential performance.

4.4.2 *b. Gustafson's Law*

Gustafson's Law offers a more optimistic view by arguing that as we increase the problem size, the parallelizable portion of the workload grows, potentially mitigating the impact of the serial fraction. Instead of focusing on fixed problem sizes, Gustafson's perspective considers that we often tackle more significant problems, with more processors and the overall efficiency can improve [18].

4.5 Complexity Classes and Parallel Algorithms

Understanding which problems can be efficiently parallelized involves concepts from computational complexity theory.

4.5.1 *a. The NC Class*

The class **NC** (Nick's Class) includes decision problems that can be solved in polylogarithmic time using a polynomial number of processors. These problems are considered "efficiently parallelizable," making them ideal candidates for parallel computation. Examples include parallel sorting algorithms, matrix multiplication, and prefix sum computations [21].

4.5.2 *b. P-Complete Problems*

In contrast, **P-complete** problems are believed to be inherently sequential. While they can be solved in polynomial time, no known efficient parallel algorithm exists. It is widely conjectured that P-complete problems do not belong to NC, placing a fundamental limit on parallelizability [20].

5 Parallel Computing Paradigms

5.1 Processes

In computer science fundamentals, a process is a unit of execution - an 'active' entity, distinct from a program, which is a 'passive' entity. When a program (like a .exe or binary file) is run multiple times, each instance creates a new process. This concept enables parallelization, where a program can divide a problem across multiple processes. For example, consider a 100×100 2D matrix containing 10,000 elements (100 × 100).

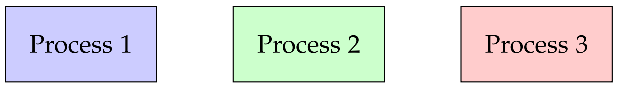

Fig. 3: Processes in an Operating System. Each process runs in its own isolated memory space and maintains separate resources.

If split across 10 processes for parallel computation, each process would handle 1,000 elements [24].

There are multiple ways to distribute elements across processes, such as block-cyclic, block, and cyclic distribution. In block distribution, contiguous chunks of data are assigned to each process. Cyclic distribution alternates elements among processes in a round-robin fashion. Block cyclic combines these approaches by cyclically distributing blocks of elements, balancing communication overhead, and load [25]. This hybrid approach benefits algorithms requiring local data access and regular communication patterns, such as matrix operations in parallel computing. Each of these distribution strategies typically affects the scalability and efficiency of the program. It is essential to understand the problem's core and pick the appropriate strategy based on that. It is crucial to benchmark the program with different distribution strategies when in doubt, as thorough testing is key to ensuring the best performance.

It is not only matrices that can be distributed across processes; matrices were the example given due to their prevalent use in high-performance computing. Processes are isolated, which means each unit of execution runs within its environment with no direct awareness of other processes. The only way to communicate between processes is through message passing, which ensures they can work together to solve problems. Multiple libraries implement message passing, but the most popular and standardized implementation is the Message Passing Interface (MPI). MPI provides a comprehensive set of protocols and routines that enable efficient data exchange and synchronization between parallel processes, allowing robust parallel functionality [26].

5.2 Threads

5.3 Threads in Parallel Computing

A thread refers to a single sequential flow of execution within a process. Threads are typically not as isolated as processes since they exist within the same memory space and share resources like heap memory, file handles, and global variables. They are more lightweight than processes because creating and switching between threads requires less overhead, but there are limits to the number of threads that can be efficiently run on a system. These limitations stem from various factors, including available system resources (memory and CPU cores), operating system constraints, and the overhead of context switching between threads [27], [28].

Threads can be used to solve various parallel computing problems. One example is the adaptive quadrature problem, which can be solved using a First In, First Out (FIFO) approach where threads cooperatively process work items from a shared queue until the desired accuracy threshold is reached; each thread can also have its own queue. However, this is not the only threading-based solution - the same problem can also be solved using higher-level abstractions like tasks, which provide a more structured approach to parallel execution. Modern threading frameworks and libraries have evolved to provide increasingly sophisticated abstractions and primitives, making implementing parallel algorithms efficiently while handling synchronization, load balancing, and thread safety concerns easier [29], [30].

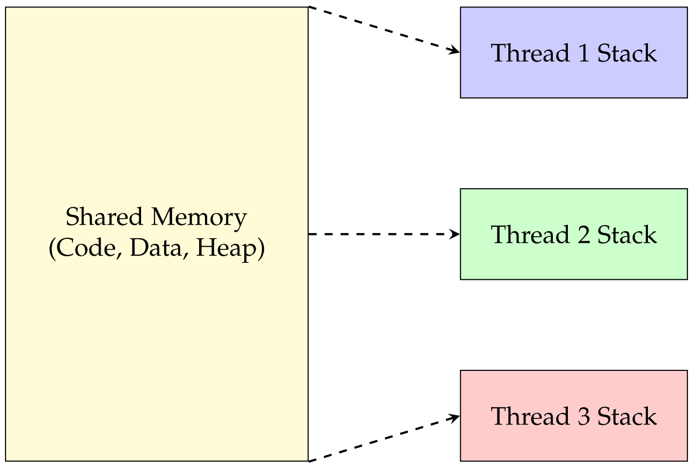

Fig. 4: Threads in an Operating System. Threads share the process's memory (code, data, heap) while each maintains its own stack for execution context.

Several libraries allow threading to occur within the operating system, and a popular standard is the OpenMP standard, which provides coordination and synchronization mechanisms for threads to solve problems accurately while working in unison. There are mechanisms such as locks that prevent race conditions by providing exclusive access to a resource to one thread at a time and other directives that help manage parallel execution, including barriers for synchronization points, critical sections for protecting shared resources, atomic operations for thread-safe updates, and scheduling clauses that control how work is distributed among threads. These features, combined with OpenMP's pragma-based approach, make it easier for developers to parallelize their code while maintaining correctness and achieving better performance [31], [32].

6 Factors Impacting Parallelization

Multiple factors influence the performance, efficiency, and potential for parallelizing parallel programs. This section provides an overview of these critical components and examines their extent of impact. The most crucial factor we must first consider is the nature of the problem itself—specifically, whether it can be parallelized and how well it can be parallelized [33].

6.1 Nature of the Problem

The characteristics of the problem fundamentally determine whether parallelization will be effective and what approach should be taken. Some problems are inherently sequential and resist parallelization, while others naturally divide into independent tasks that can be executed concurrently [34]. Understanding these problem characteristics is essential before attempting to implement any parallel solution, as they directly influence the potential speedup and scalability that can be achieved through parallelization [35].

The first step in parallelization is to find concurrency in the problem. Some problems are embarrassingly parallel, while others cannot be parallelized even with the most robust and innovative methods [36]. An example of an embarrassingly parallel program is 3D video rendering handled by a graphics processing unit, where each frame (forward method) or pixel (ray tracing method) can be handled with no interdependency, allowing the workload to be distributed across multiple processors with minimal overhead [37].

If a problem has a chain of dependencies—meaning that you must compute step i before step $i+1$—then there is little opportunity to run parts of the problem simultaneously [38]. For example, many recursive or iterative processes where the output of one iteration is needed for the next are challenging to parallelize, as the sequential nature of the algorithm forces a strict order of operations that limits concurrency [39].

We have discussed the NC (Nick's Class), which consists of problems that can be solved in polylogarithmic time using a polynomial number of processors [40]. P-complete problems are believed to be inherently sequential because finding an efficient parallel (NC) algorithm for any P-complete problem would imply that P equals NC, a result that most experts doubt [41]. A classic example of a P-complete problem is the Circuit Value Problem (CVP), a benchmark for problems resistant to parallel approaches [40].

There are many problems where concurrency can be found to parallelize the program; problems that do not have a sequential interdependency have massive potential for parallelization [42]. Data structures such as arrays, graphs, queues, stacks, and hash tables often support a wide range of parallel patterns, which will be discussed later in this paper [43]. This inherent ability to divide work among independent subtasks is what makes them attractive targets for parallel processing.

For instance, when processing large datasets, operations can often be performed on different data segments simultaneously without affecting the final result [44]. Similarly, many simulation problems can be divided spatially, with different regions being computed independently before combining results [33]. This division into independent tasks allows for efficient utilization of computing resources, provided that the data can be partitioned in a balanced and effective manner [34].

The challenge lies in identifying these independent components and determining the optimal granularity of parallelization, as too fine-grained parallelism can lead to excessive overhead [38]. At the same time, parallelism that is too coarse-grained might not fully utilize available computing resources. Understanding the nature of these parallelizable problems is essential before applying specific parallel programming patterns and techniques [35]. Illustrating how different data structures can be parallelized.

6.2 Problem Size

Another significant factor influencing the efficiency of parallelization is the size of the problem. For example, if a task operates on a single element and cannot be subdivided into smaller, independent sub-tasks, then parallel processing might offer little to no benefit. In such cases, the workload is too fine-grained to distribute effectively across multiple processing units [45]. This issue arises because the overhead of managing parallel execution, including synchronization and inter-process communication, outweighs any potential speedup [46].

Conversely, many problems are inherently large-scale, presenting their challenges regarding parallelization. When a dataset or computational task is huge, simply dividing it into chunks might not be enough. Two major issues can arise:

1) **Memory Constraints:** After partitioning the data, each chunk must fit into the available memory. If the dataset is so vast that the divided portions exceed the memory capacity, the system may resort to disk swapping or other slower memory management techniques [47]. This negates the speed benefits of parallelization and can lead to significant performance bottlenecks. Efficient execution of large-scale parallel tasks under memory constraints requires intelligent memory management techniques, such as active memory scheduling and hierarchical memory models, to maximize utilization while reducing latency [48].
2) **Cache Efficiency:** Modern processors rely heavily on cache memory to speed up data access. Maintaining cache coherence becomes critical when a problem is divided among multiple processors or threads [49]. For huge problems, frequent cache invalidations and the overhead of synchronizing caches across cores can severely degrade performance [45]. Research indicates that cache coherence protocols and optimal cache sizing significantly impact the performance of parallel algorithms, and choosing the proper configuration can lead to substantial improvements in computational efficiency [50]. Additionally, optimizing cache-aware algorithms, such as locality-preserving data structures, can help reduce cache misses and improve speed [51].

In summary, while parallelization can dramatically accelerate computational tasks, its effectiveness depends on the problem size [45]. Tasks that are too small may not be divisible into enough sub-tasks to justify the overhead of parallel processing. On the other hand, huge problems may run into memory limitations and cache inefficiencies, both of which can limit the performance gains [49]. Understanding and addressing these challenges is crucial when designing systems and algorithms for practical parallel computation.

To evaluate how the size of a problem impacts parallelization, we conducted two scaling experiments on the Cirrus supercomputer at the University of Edinburgh for a cellular automaton problem that implements Conway's Game of Life parallelized using Message Passing Interface. The first experiment examined weak scaling by maintaining a fixed number of 5 processors while varying the landscape size from 500×500 to 5000×5000. The second experiment tested strong scaling by fixing the landscape size at 5000×5000 while varying the number of processors from 2 to 16. Each configuration was run 10 times to ensure reliable measurements [52]. The implementation used a decomposition strategy where the grid was divided among processes, with each process responsible for updating its local portion while managing necessary boundary communications with neighboring processes [53].

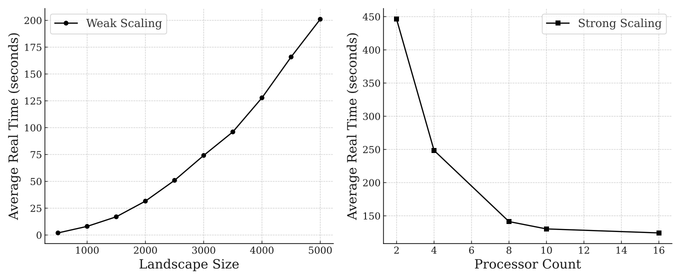

Fig. 5: Scaling Results of a Cellular Automaton parallelised using Message Passing Interface

Our experiments on the Cirrus supercomputer were designed to assess how varying landscape sizes affect the parallelization of a C-based MPI implementation of Conway's Game of Life. The simulation divides the global grid among processors, with each processor handling a subgrid that includes ghost boundaries for halo exchanges [52]. As the overall landscape size increases—from 500×500 to 5000×5000 cells—the computational load per process grows significantly, since the total number of cells (and thus the number of updates) increases quadratically with the grid dimensions [34].

While larger landscapes naturally increase the volume of computations, they also affect the communication overhead inherent in MPI applications [53]. Each process communicates with its neighbours to update ghost cells, and the cost of these communications depends on the perimeter of the subgrid [49]. In larger grids, the relative ratio of the communication boundary (perimeter) to the computational workload (area) tends to decrease, potentially offering better efficiency per process [45]. However, the absolute amount of data exchanged still grows, which can lead to longer overall runtimes despite the improved computation-to-communication ratio [44].

It is clear that increasing the landscape size intensifies the parallel workload, which typically influences the efficiency of parallel programs; this is evident, leading to a marked rise in computational effort reflected in the observed execution times [52]. Although larger local subgrids can mitigate the relative impact of communication overhead, growth in the number of cells ultimately results in increased runtimes [34]. This analysis underscores the need to balance computational workload and communication efficiency when scaling parallel MPI applications for larger problem sizes [53]. The pseudocode of the program is available in the Appendix.

6.3 Parallel Patterns

The parallelization pattern typically affects the parallel program's scalability, speedup, load balancing, and overhead. It is important to state that there are constructs of parallel patterns and parallel patterns themselves. There are many parallel patterns, some popular ones including geometric decomposition, actor pattern, pipeline pattern and recursive data pattern. Selecting the appropriate pattern dictates how efficiently a program scales with additional processors and influences the overall execution speed by managing overhead and balancing loads across computing units [54]. For instance, geometric decomposition divides a problem's domain into smaller regions that can be solved concurrently, often improving scalability and speedup [55]. At the same time, the actor pattern emphasizes independent entities communicating via messages to maintain balanced work distribution [56]. Similarly, the pipeline pattern organizes tasks into a sequence of processing stages that can operate in parallel, and the recursive data pattern leverages divide-and-conquer strategies to handle complex tasks by breaking them into simpler subproblems [57]. Each pattern relies on underlying constructs provided by modern programming languages and frameworks, reinforcing that low-level tools and high-level design patterns are essential for developing efficient and robust parallel programs.

Effective load balancing is another critical factor influenced by the chosen pattern. Techniques such as the Actor model encapsulate state and behavior into independent units communicating through message passing. This results in a more dynamic distribution of tasks across processors and reduces the risk of some cores idling while others are overburdened [56]. However, every pattern brings its overheads; for instance, the pipeline pattern, which organizes computation into sequential stages, may suffer from inefficiencies if one stage processes data slower than the others, causing subsequent stages to wait [58].

It is also essential to distinguish between the constructs of parallel patterns and the patterns themselves. Constructs refer to the fundamental building blocks provided by programming languages or libraries—such as threads, tasks, futures, or message-passing mechanisms—that enable parallel execution [59]. In contrast, parallel patterns are higher-level, reusable solutions that encapsulate best practices and design strategies for common parallel programming challenges. By leveraging these patterns, developers can create efficient parallel programs without reinventing the wheel for each new problem [54].

6.4 Programming Language & Libraries

Each programming language is suitable for specific applications. Programming languages are typically classified into high-level and low-level programming languages [43]. The choice of language significantly impacts the performance of the parallel program; low-level languages are usually preferred for parallel programs as they are closer to the hardware with limited abstractions [44]. That does not mean that high-level languages cannot write parallel programs, but if speed and efficiency are paramount, it is more appropriate to write a parallel program in a low-level language, just as if you want an efficient serial program, it is crucial to write the program in a language closer to hardware, the same principles apply to parallel programs [36].

Libraries used for parallel programming also play a crucial role in an application's performance [38]. Some libraries are meticulously optimized to extract every bit of efficiency from each line of code—employing techniques like low-level synchronization, vectorization, and loop unrolling—to maximize speed on specific hardware [49]. In contrast, while generally efficient, other libraries may not be as finely tuned. Moreover, even a well-optimized library can exhibit varying performance across standard implementations due to differences in system architectures, compiler optimizations, or algorithmic choices [34]. Therefore, selecting the appropriate library involves balancing factors such as raw performance, compatibility with target hardware, and overall ease of integration with the existing codebase [47].

We implemented Conway's Game of Life as a parallel cellular automaton using MPI in Python and C to compare different programming languages' performance and parallelization capabilities. We ran these implementations on the Cirrus supercomputer at the University of Edinburgh to analyze how each language affects program performance. The program was parallelized using 4 processes.

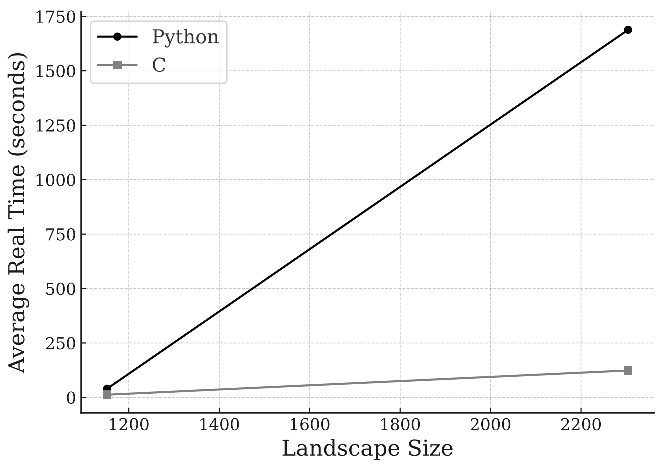

Fig. 6: Timing Results of a Cellular Automaton parallelized using Message Passing Interface

Based on the figure, it is evident that C is far more efficient than Python in writing MPI programs. Our experiments were conducted on two landscape sizes, namely 1152 and 2304, and the results clearly indicate that the efficiency gap between the two languages grows as the problem size increases.

One fundamental reason for this disparity is the intrinsic difference in how the two languages are executed. C is a compiled language, which means that its code is translated directly into machine language before execution. This allows for low-level optimizations, efficient memory management, and direct access to hardware resources. Such features are crucial in high-performance computing environments, where every millisecond counts [60]. In contrast, Python is an interpreted language that introduces an additional layer of abstraction. This results in runtime overhead due to dynamic type checking, garbage collection, and the interpretation process itself. As the problem size scales up, these overheads become increasingly significant, thereby exacerbating the performance gap [61].

Furthermore, when it comes to MPI programming, the efficiency of communication between processes is paramount. C-based MPI implementations are typically optimized to leverage the full capabilities of the underlying hardware, including low-latency networks and high-throughput interconnects. Python, although supported by libraries such as `mpi4py`, adds an extra layer of abstraction over the native MPI calls. This extra layer can introduce latency and additional computational cost, which in turn diminishes its performance, especially in larger-scale problems where communication costs dominate [62].

Additionally, as the landscape size doubles, the amount of data to be processed and communicated increases significantly. This not only increases the computational load but also magnifies any inefficiencies in the programming model. In C, the low-level control and static typing allow the program to scale more gracefully under increased load. Python, on the other hand, may suffer from scaling issues due to its inherent overhead, making it less suitable for very large problem sizes in MPI contexts [63].

The combination of compile-time optimizations, efficient memory management, and direct hardware interfacing gives C a substantial advantage over Python for parallel programming. These advantages become even more pronounced as the problem size increases, leading to a widening efficiency gap between the two languages [64].

6.5 Hardware

This section discusses the impact that hardware has on parallel computing. Hardware is broad and overarching, so this section is divided into different components, including the cache, memory, CPU, accelerator, architecture, and interconnects. Each of these elements plays a critical role in ensuring efficient parallel computation. The cache minimizes data access latency by storing frequently used data close to the processor, thereby improving execution speed [65]. The memory subsystem provides the bandwidth and capacity to manage large volumes of data efficiently, which is crucial for high-performance computing (HPC) applications [66]. The CPU is the central orchestrator, coordinating multiple threads and processes to maximize computational throughput [67].

Accelerators, such as GPUs and FPGAs, enhance performance by offloading specialized tasks and enabling massive parallelism. Adopting accelerators in HPC systems has become increasingly common due to their ability to execute data-parallel workloads efficiently [68]. In the subsequent sections, we delve deeper into these components, exploring their architectures, interactions, and how they collectively contribute to the performance and scalability of parallel computing systems.

6.5.1 CPU

The CPU (Central Processing Unit) is the primary computing engine responsible for executing instructions and managing data flow. Modern CPUs often feature multiple cores, enabling the simultaneous execution of multiple threads and enhancing computational efficiency [69]. Advanced techniques like pipelining, branch prediction, and out-of-order execution further optimize instruction throughput by minimizing stalls and ensuring efficient resource utilization [70].

In homogeneous architectures, where all CPU cores are identical, parallel workloads benefit from predictable performance scaling and simplified task scheduling. Uniform core performance ensures efficient load balancing, making such designs ideal for applications like scientific simulations, financial modeling, and numerical computing, where tasks can be evenly distributed across cores [71]. However, homogeneous CPUs struggle with diverse workloads, lacking the architectural flexibility to optimize specialized tasks such as deep learning inference or high-speed cryptographic processing.

Conversely, heterogeneous CPU architectures combine general-purpose cores with specialized cores or accelerators. This hybrid approach is increasingly popular in modern computing systems, mobile processors, and AI-driven architectures, where high-performance cores manage control flow while energy-efficient cores or domain-specific units handle specialized tasks [68].

For efficient parallel computing, CPU architectures must support high-speed inter-core communication, low-latency memory access, and dynamic task scheduling. In multi-core CPUs, cache coherence mechanisms (e.g., the MESI protocol) ensure consistency across cores but can introduce performance overhead due to increased synchronization traffic. Optimizations like NUMA-aware memory placement and thread affinity help reduce cache contention and improve locality, directly enhancing parallel performance [66].

6.5.2 Cache

Caches are small, high-speed memory units situated close to the CPU cores. They store frequently accessed data and instructions to reduce latency and improve performance. Typically, modern processors employ a multilevel cache hierarchy (L1, L2, and sometimes L3), each balancing speed and capacity [66].

In parallel computing, efficient cache utilization is critical as multiple cores require fast access to shared data. Poor cache management can lead to contention and increased memory access latencies, reducing system efficiency [65].

A key challenge in multi-core and many-core systems is cache coherence, which ensures that all processor cores see a consistent view of memory. Cache coherence protocols, such as MESI (Modified, Exclusive, Shared, Invalid), are crucial in synchronizing shared data across multiple caches. However, maintaining coherence introduces performance overhead, as frequent invalidations and updates can increase memory traffic, affecting parallel efficiency [69]. To mitigate this, modern architectures incorporate non-uniform cache architectures (NUCA) and directory-based coherence mechanisms to balance latency and bandwidth.

6.5.3 Memory

Memory, commonly implemented as DRAM, is the primary workspace for data and instructions during program execution. The performance of the memory subsystem—in terms of capacity, bandwidth, and latency—directly influences the efficiency of parallel applications. For instance, distributed memory architectures require efficient data communication strategies to optimize performance [72].

In heterogeneous systems, managing distinct memory spaces for different types of processing units introduces complexities in data coherence and efficient access. Consequently, optimizing memory access patterns is key to improving performance [73]. Coordination and synchronization are crucial for optimal performance, as the limited uniformity of the architecture might cause unpredictability in the performance of parallel programs.

One of the major bottlenecks in parallel computing is memory bandwidth limitations. As the number of processing cores increases, the demand for memory access grows, potentially leading to memory contention and bottlenecks in shared-memory architectures. High-bandwidth memory (HBM) and DDR5 are designed to mitigate these issues by offering increased bandwidth and reduced latency [68].

6.5.4 Accelerator

Accelerators have become prevalent in parallel computing due to their ability to handle specialized tasks that demand high computational throughput and parallel data processing. Unlike traditional CPUs, which prioritize general-purpose computing and sequential task execution, accelerators exploit task and data parallelism by leveraging architectures optimized for specific workloads. For instance, GPUs (Graphics Processing Units) utilize a Single Instruction Multiple Thread (SIMT) execution model, making them particularly effective for matrix operations, convolutional computations in deep learning, and large-scale simulations in computational physics and molecular dynamics [68].

Similarly, FPGAs (Field-Programmable Gate Arrays) offer customizable hardware acceleration, allowing direct circuit-level optimization of specific tasks. Unlike fixed-function GPUs, FPGAs provide fine-grained parallelism and low-latency execution, making them suitable for real-time data processing, signal processing, and cryptographic applications [73].

On the other hand, ASICs (Application-Specific Integrated Circuits) provide the highest efficiency level for targeted applications, such as AI inference, blockchain computations, and high-frequency trading. By being hardwired for specific functions, ASICs eliminate overhead associated with general-purpose processing, leading to unmatched power efficiency and computational density [74].

While accelerators significantly enhance parallel computing capabilities, their integration into heterogeneous architectures introduces multiple challenges. Efficient memory management becomes critical, as different processing units operate on separate memory spaces. Unified Memory Architecture (UMA), zero-copy memory transfers, and direct memory access (DMA) aim to bridge these gaps, ensuring minimal data transfer overhead [66].

6.5.5 *Architecture*

The architecture of a computing system defines the overall design and organization of its hardware components, including the CPU, cache, memory, accelerators, and interconnects. The choice of architecture directly influences parallel computing performance by determining computational efficiency, scalability, energy consumption, and ease of programming. A key distinction in modern architectures is between homogeneous and heterogeneous designs, which have unique implications for parallel computing performance [71].

Homogeneous architectures provide predictable performance scaling and efficient task scheduling, while heterogeneous architectures combine different processing units, such as general-purpose CPUs with specialized GPUs, FPGAs, or TPUs, to leverage their strengths [68].

6.5.6 *Interconnects*

Interconnects are the communication backbone linking various hardware components within a computing system, such as CPUs, memory, and accelerators. Their performance is crucial in parallel computing, as they directly impact data transfer speed, latency, and overall system efficiency [70].

One of the primary challenges in parallel computing is minimizing communication overhead while maximizing data locality and concurrency. High-speed, low-latency interconnects—such as PCIe, NVLink, and Intel's Compute Express Link (CXL)—are designed to facilitate high-throughput data transfer between CPUs, GPUs, and accelerators [69].

7 Tooling and Ecosystem

Numerous tools aid parallel programming, and the number has been growing since its inception. These tools are enveloped within a rich ecosystem that helps write, debug, and optimize concurrent applications [2], [75]. In this section, we explore the various components of this ecosystem, from foundational programming frameworks to performance analysis tools, simulation platforms, and community resources that together empower developers to build robust parallel systems.

7.1 Programming Frameworks and Libraries

A wide array of programming frameworks and libraries forms the spine of parallel development. Widely adopted frameworks such as MPI (Message Passing Interface) and OpenMP provide standardized approaches to parallelism on distributed-memory and shared-memory systems, which have been discussed earlier. There are specialized libraries like CUDA and OpenCL, which haven't been mentioned, that target the massive parallelism offered by GPUs, enabling developers to harness the power of heterogeneous architectures for compute-intensive tasks fully [52], [76].

These frameworks offer the fundamental constructs for process synchronization, task distribution, and communication and integrate with modern programming languages like C++, Python, and Rust to provide a high degree of abstraction. These abstractions enable developers to focus on algorithmic design rather than low-level details while achieving efficient execution across multi-core and many-core environments.

7.2 Performance Analysis and Debugging Tools

Identifying bottlenecks and ensuring the correctness of parallel applications requires a robust suite of profiling and debugging tools. Tools such as Intel VTune and NVIDIA Nsight offer deep insights into applications' runtime behavior, including CPU and GPU utilization, memory bandwidth usage, and communication overheads [77], [78]. These tools are essential for fine-tuning performance, as they help pinpoint areas where parallel efficiency may be improved.

In parallel environments, debugging challenges such as race conditions, deadlocks, and synchronization issues become more pronounced, including root cause analysis; when parallelizing a serial program, it moves the number of instances your program is running from singular to multiple, which makes it more complicated in trying to identify where the issue is when a bug arises. Specialized debugging tools like TotalView and Allinea DDT are designed to handle the complexity of multi-threaded and multi-process applications. They allow developers to step through concurrent executions, inspect thread states, and monitor inter-process communication, ensuring that parallel programs run efficiently and correctly [79].

7.3 Simulation and Visualization Platforms

Simulation and visualization tools play a crucial role in the design and analysis of parallel systems. These platforms allow developers to model and predict the behavior of complex systems under various workloads, providing a sandbox environment for testing theoretical models like PRAM, BSP, and LogP [23], [80]. This helps them understand the dynamics of their parallel programs before even writing code. By simulating parallel architectures and workloads, these tools provide a robust environment for analyzing the scalability and synchronization overhead before deployment on actual hardware.

Visualization also plays a critical role; graphing tools allow the process to assist in interpreting performance data, which can be critical in benchmarking and understanding implementation dynamics to make better decisions about parallel programs, and it also makes it easier to understand complex interactions within parallel systems. Graphical representations of thread activity, memory usage, and interconnect traffic can highlight inefficiencies that will not be visible simply by compiling the program and executing the code. Visualization has become very critical and valuable to the parallel computing ecosystem because of the insights that lead to more efficient and resilient parallel systems [81].

7.4 Community, Documentation, and Learning Resources

The active communities surrounding parallel programming frameworks and tools greatly enhance the tooling ecosystem. Open-source projects, online forums, and collaborative platforms such as GitHub provide an environment where developers can share code, report issues, and contribute to improvements [82]. This collaborative spirit accelerates innovation and ensures that best practices are disseminated widely across the community [83].

In addition to community support, comprehensive documentation, tutorials, and training courses are critical for empowering developers to use these tools effectively. Many frameworks offer extensive official documentation [84], while academic institutions and industry leaders provide webinars, workshops, and online courses to help new and experienced developers [85]. The continuous evolution of these resources ensures that computer scientists remain up-to-date with the latest advancements and techniques in parallel programming, fostering an environment of lifelong learning and innovation [86].

7.5 Integration and Deployment Tools

Integration with modern build systems and deployment tools is essential to bringing parallel applications from development to production. Tools such as CMake, Make, and various IDE plugins facilitate the compilation and linking of parallel codebases across different platforms [87]. Furthermore, containerization technologies like Docker and orchestration tools like Kubernetes are increasingly used to deploy parallel applications in scalable, cloud-based environments, ensuring consistent performance and manageability [88].

Deployment tools also play a critical role in monitoring and maintaining the health of parallel applications once they are in production. Real-time monitoring systems, log aggregators, and automated testing frameworks help ensure that performance remains optimal and that any issues are promptly identified and addressed [89]. This holistic approach to tooling—from development to deployment—ensures that parallel applications are robust, scalable, and ready to meet the demands of modern computing workloads [49].

8 Case Study of a Road Traffic Simulation

Using a road traffic simulation as a case study, we identify both suitable and unsuitable parallel programming patterns for implementation. We then analyze how different hardware configurations and programming languages affect these patterns' performance. Additionally, we present methods to evaluate the effectiveness of the program implemented with the selected pattern. The detailed specifications of the road traffic simulation are provided in the appendix.

8.1 Chosen Design Patterns

8.1.1 Geometric Decomposition

8.1.1.1 Overview of Geometric Decomposition: Geometric or domain decomposition is a parallel computing strategy that partitions a spatial domain into smaller, manageable subdomains [55]. It could be suited for the simulation model but not without drawbacks; it is typically used in domain problems that require localized handling and communication between neighbors, often in conjunction with the Message Passing Interface [90]. The road traffic simulation represents roads and junctions as a graph structure, where vehicles move from one node to another, engaging with local features such as traffic lights and possibly navigating events such as crashes and collisions. By leveraging geometric decomposition, the simulation can be partitioned based on the geographical layout of the road network, allowing for efficient parallel processing and management of localized interactions. Still, the partitioning is not as straightforward, and the simulation dynamics complicate utilizing the strategy [91].

It is essential to state that the data structure we are dealing with for this simulation is a graph. This makes it challenging to decompose its subdomains across multiple processing elements without problems in synchronization and efficient partitioning. Unlike linear data structures such as arrays or linked lists, which can be easily divided into chunks and distributed based on the amount of UEs available, graph structures involve complex interconnections that complicate decomposition [92]. At the same time, tools like MPI Graph Topology can assist in decomposing a graph for parallel computation; there are bottlenecks in using this approach specifically for this simulation model [93].

However, dividing the number of elements by the number of processes and distributing the remainder across processing elements is not straightforward for a graph data structure. A graph operates on vertices and edges, not on a simple linear sequence of elements, making it difficult to partition evenly [94]. Although the unidirectional graph-like model for the road simulation can simplify decomposition somewhat, the underlying complexities of vertex and edge distribution remain significant. Graphs consist of vertices connected by edges, and these connections often span across different partitions when the graph is divided. This results in communication and synchronization overhead as vertices in one partition may frequently interact with vertices in another. Minimizing such inter-partition edges is crucial for reducing communication costs and enhancing performance, but achieving this balance is inherently challenging due to the graph's topology [95].

Despite these challenges, it is possible to decompose a graph and distribute it across processing elements. Graph partitioning has become a well-established technique in parallel computing, with various strategies developed to address its complexities [96]. Methods such as adjacency lists are commonly used to enable efficient graph processing, particularly for sparse graphs like road networks. Additionally, graph partitioning libraries provide multiple techniques for dividing graphs among processing elements (PEs). One of the most widely used tools for this purpose is the sequential partitioner METIS [97]. Parallel partitioners like ParMETIS, PT-Scotch, and JOSTLE are also designed explicitly for distributed-memory architectures in parallel programs. However, these implementations have specific challenges, which we will explore in detail [98].

8.1.1.2 Advantages in Applying Geometric Decomposition for this Road Traffic Simulation Model: As discussed earlier, the road network's graph can be naturally divided into subgraphs or regions, each containing a subset of junctions and connecting roads. Since vehicles interact primarily with their immediate environments, such as the road they are on or the intersection they are approaching—most of the computational workload is localized within these subdomains and can be effectively managed [99]. Tools like neighborhood collective operations can be used to manage communications in the topology if implemented effectively using Message Passing Interface. For instance, calculating a vehicle's position, speed adjustments due to road congestion, and decisions at junctions are all confined to specific areas of the network; this is possible, but the simulation dynamics could typically complicate this. However, we have to discuss this as the localized processing of sub-domains is one of the highlights of domain decomposition due to its ability to be very efficient as new processes are added. Geometric decomposition could capitalize on this locality by assigning each subdomain to a different processor, enabling concurrent communication and updates without significant interference [100].

Geometric decomposition can enhance computational efficiency and scalability in huge problem sizes but depends on the distribution's granularity [101]. Parallel processing of subdomains means that the simulation can handle more vehicles and more extensive road networks without a linear increase in computation time; this would allow the program to scale proportionally to the number of available PEs, but there would also be limitations as processes grow, due to the serial portion of the code that becomes a bottleneck in parallelization. Each processor could focus on its localized setting, handling interactions in that instance. This division of labor reduces the computational overhead. It allows for more frequent updates and finer-grained simulation details within each subdomain. Still, it is essential to note that allowing this is not as straightforward as discussed due to dynamism and the concurrent activities within the simulation, making it difficult for each execution unit to manage its domain effectively while also being part of the global environment.

Moreover, geometric decomposition facilitates effective management of the simulation's data structures and memory usage. By keeping data localized to specific subdomains, the simulation minimizes the need for global data synchronization and reduces memory contention among processors [102]. This localization is particularly beneficial for recording and reporting simulation statistics when needed as it would allow calculations and derivations of nuances within the simulation, which might be challenging to capture in other parallel pattern environments such as the Actor model due to the granularity as it would require that each actor sends a message as an update and that can impact performance due to message passing overload. Each processor can independently maintain and update these statistics for its subdomain in the Geometric pattern, contributing to a comprehensive final report without excessive inter-processor communication.

8.1.1.3 Challenges in Applying Geometric Decomposition for this Road Traffic Simulation Model: However, applying geometric decomposition requires careful consideration of inter-subdomain interactions and load balancing [103]. Vehicles moving between subdomains introduce the need for communication between processors to ensure seamless transitions and consistent simulation states. There must be established to handle these boundary conditions without introducing significant latency; vehicles transferring locations between processed would have to be communicated, and as the number of cars increases and as the simulation scales with additional complexity, this can eventually become a challenge and a bottleneck in the simulation. The road network may have varying complexity and traffic density regions, leading to unequal computational loads across processors. Dynamic load balancing would required, which is not only complex to implement but also complex to manage [104].

Another significant challenge lies in the limitations of graph partitioning libraries. The graph partitioning libraries are typically optimized for large-scale graphs with millions of vertices and edges [105]. The specifications for the road network provided would not typically be optimized for graph partitioning libraries like METIS, which usually add overhead. Graph partitioning libraries use sophisticated algorithms to segment the graph into efficient, adequate proportions. The time taken to partition a graph increases with the size and complexity of the graph. For massive graphs, graph partitioning libraries are optimized to handle them efficiently. Still, for smaller graphs, the relative partitioning time might be significant compared to the overall simulation runtime, which would be evident in the road traffic simulation, eventually making the program inefficient [106]. If partitioning your graph takes hours and your simulation runs for minutes, it is not the most efficient scenario; a better approach would be a better ratio in proportion to the simulation time.

Furthermore, graph partitioning libraries, such as METIS's algorithms, have specific time complexities [107]. The multilevel approach typically operates in near-linear time relative to the number of edges, but the constants involved can be non-trivial. Additional memory is required to store intermediate graph representations during the coarsening and partitioning phases. This includes data structures for the coarsened graphs, mappings between levels, and other auxiliary information [108].

While for massive graphs, this memory overhead is manageable and often justifiable by the performance gains in partitioning, the relative memory consumption might be less efficient for smaller graphs, potentially limiting resources for other simulation components [109].

It's important to note that the simulation dynamics do not favor the linear pattern of geometric decomposition. For example, the problem's specifications stated that vehicles followed recalculated routes; if this weren't the case, this design strategy would have been more viable as we could apply data parallelism to represent each road as a queue and distribute partitions across processes. The overhead would be more tolerable, but a critical component for this road traffic model is when vehicles arrive at a junction, their destination can be re-routed, which would make readjusting the queue impractical and further exacerbate the unsuitability of this approach for the road traffic simulation [110].

Crashes at junctions without traffic lights introduce another layer of complexity in a partitioned environment. Since crashes are localized events involving vehicles from different partitions, detecting and managing these incidents requires robust mechanisms to prevent race conditions [111]. What happens when two cars crash in two distinct processes simultaneously? How do you account for it in the simulation? It would typically require managing a synchronization procedure, which would likely harm performance due to its need to be aware at every point in the simulation. UEs must collaborate to update the state of junctions involved in crashes, ensuring that the removal of vehicles from the simulation is handled accurately and consistently across all relevant partitions [112]. This necessity for coordinated event handling can increase the computational and communication overhead, challenging the overall efficiency of the simulation.

Fuel consumption and vehicle removal processes also experience significant impacts from graph partitioning. Each UE is responsible for tracking the fuel levels of vehicles within its partition, necessitating communication with other UEs when vehicles run out of fuel or are removed due to crashes. Efficiently managing these dynamic state transitions across partitions with the appropriate data structures and memory buffers would require careful programming [113]. The inter-PE communication necessary for these operations can introduce additional latency, particularly when vehicle states frequently change or interact across multiple partitions [114].

It is also possible that domain decomposition could alter vertex ordering within the graph. Since graph partitioning libraries usually focus on optimizing load balancing and reducing communication overhead, they rearrange the vertices to achieve these goals [107]. In the context of road traffic simulation, where junctions represent critical points for vehicle movements and traffic light operations, this

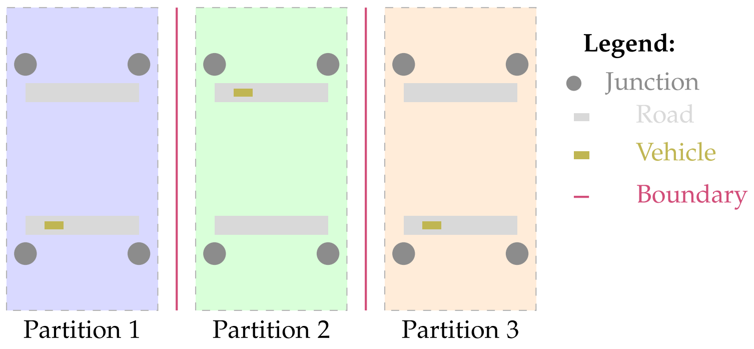

Fig. 7: Domain Decomposition in Road Traffic Simulation

reordering can complicate the management of vehicle routes and traffic flow. Vehicles moving between partitions require seamless communication between UEs to accurately update their locations and statuses [115]. This necessity introduces latency and demands robust synchronization mechanisms to ensure vehicle movements and traffic light states remain consistent across all partitions.

The geometric decomposition pattern complicates the dynamic addition of vehicles, primarily due to challenges like identifying the appropriate unit of execution (UE) for placing new vehicles, tracking newly added cars, and managing the communication overhead involved in dynamically handling the addition and removal of vehicles [34]. As a result, this strategy is not well-suited for supporting this specific dynamic aspect of the simulation.

These points all emphasize one theme within the simulation that geometric decomposition cannot possibly handle: synchronization to manage the simulation's dynamism and nonlinearity [111]. In the next section, we will discuss the synchronization required.

8.1.1.4 Synchronization and Memory Considerations:

The synchronization mechanisms needed for this parallel design strategy in the road traffic simulation are complex and robust, making their development, testing, and debugging time-consuming [112], [113]. A significant bottleneck is ensuring that each partition remains independent while the overall model operates seamlessly.

Memory requirements are generally lower per processing element in distributed memory architectures than in shared memory architectures [114]. However, overall memory usage remains a consideration because the data structures used to store graph representations still consume significant memory resources. The most considerable bottleneck associated with this design pattern is latency, arising from the communication overhead between distributed processing elements. Multiple messages are being sent between PEs simultaneously to ensure synchronization, which can lead to a build-up in the queues, further exacerbating latency and affecting the overall time of the simulation [116].

Due to the reordering of vertices, messaging patterns across processing elements (PEs) can become unpredictable. This unpredictability hampers traceability, root cause analysis, and debugging processes, limiting the ability to successfully and efficiently derive an optimal and correct simulation within an adequate amount of time [111]. When vertices are distributed in a non-sequential manner, tracking the flow of messages and identifying where and why specific issues occur becomes challenging. This lack of clarity complicates the debugging process, making it difficult to pinpoint errors or inefficiencies within the simulation, eventually leading to significantly larger development time.

Beyond the challenges in messaging and debugging, vertex disordering adversely affects data locality and cache performance [115]. In a well-ordered graph, related vertices are often stored closely in memory, facilitating faster data access and improved cache utilization. However, when vertices are reordered and dispersed across different partitions, the simulation may experience increased cache misses and slower data retrieval times. This degradation in cache performance can significantly reduce the overall efficiency of the simulation, as more time is spent accessing scattered data rather than processing it; this can also be very time-consuming as trying to find the next edge in the graph non-linearly might not be very efficient. Additionally, the complexity of data access patterns increases, making it harder to optimize memory usage and further impacting the simulation's performance [75].

8.1.1.5 Impact on Hardware: Multi-core or many-core parallelism is typically integrated into general-purpose CPUs. In contrast, Graphical Processing Units (GPUs) offer a high degree of parallelism through numerous simple cores. However, effectively using GPUs requires structuring compute problems to fit GPU hardware's regular, parallel nature [117]. Additionally, the simulation in question would involve irregular operations that are "sparse"—entailing numerous random memory accesses due to the nonlinearity of the simulation—which can negatively impact data locality and cache performance [118].

While shared-memory systems are adequate for the scale of this simulation, multi-node systems are more suitable for handling the I/O required at the simulation's end. Multi-node systems, especially those with NUMA (Non-Uniform Memory Access) architecture, benefit from data decomposition, which allows more efficient data placement across memory banks [72]. If the simulation is properly load-balanced and inter-unit communication is minimized, these systems can better use the available memory bandwidth.

8.1.1.6 Impact on Programming Language: For implementing this program, I recommend using a low-level language such as C, C++, or Fortran, with a particular preference for C. C offers fine-grained control over memory management and is well-supported by libraries for geometric decomposition, making it ideal for this task [119]. While high-level languages like Python are an option, they are less practical for efficiency due to the overhead involved with recursive calls and garbage collection. Python acts as an abstraction over C, meaning that using it introduces extra overhead, lacks compilation advantages, and makes it more challenging to optimize the program, ultimately exacerbating execution speed and inefficiency [143].

8.1.1.7 Summary and Transition to Alternative Strategies: In summary, geometric decomposition is a well-tested and foundational design pattern in parallel computing; it allows flexibility and imposes no limitation on how granular the problem can be decomposed, nor on how many PEs the load can be distributed across [34]. While it is ideal for linear data structures like arrays, linked lists, and structures that follow a sequential order, it could also be suitable for a graph data structure. However, the specifications and dynamics of this problem limit its potential. Consequently, this leads us to explore another parallel design strategy that would be ideal for this problem—the Actor Model—which will be discussed in the next section [120].

8.1.2 Actor Model

8.1.2.1 Overview of the Actor Model: Given the unpredictability and nonlinearity of computations in road traffic simulations, we propose using the actor model rather than employing a decomposition strategy to solve the problem. This approach is a mathematical theory that defines 'Actors' as the fundamental building blocks of concurrent computation [131]. True to its name, the model involves independent entities (actors) that leverage the nonlinearity of the simulation. While it doesn't follow the traditional parallel computing design patterns, the Actor model has gained popularity for its ability to handle high levels of concurrency. In this framework, an actor is a computational entity that responds to messages it receives [131].

8.1.2.2 Rationale for Choosing the Actor Model: This model introduces flexibility and dynamism that wasn't achievable with geometric decomposition. The unpredictability of partition placements across execution units (UEs) is no longer an issue. The Actor model allows actors to be mapped to PEs in the programmer's chosen order. Still, significant considerations should be factored in when mapping to optimize for efficiency. However, it's important to note that the graph structure becomes irrelevant and redundant within this framework. All components within the simulation can act independently as actors. This includes the roads, junctions, and vehicles; these actors would be concurrent entities communicating through asynchronous message-passing, which imposes no limitation on the scalability of this program and allows the company to run even heavier simulations due to the amount of concurrency that can be extracted from modeling the simulation using the Actor model [121].

8.1.2.3 Advantages of the Actor Model: The actor model has significant advantages in terms of concurrency and scalability. This parallelism is crucial for large-scale simulations involving thousands of junctions and vehicles. The model inherently supports high concurrency, allowing numerous actors to execute simultaneously without interfering with each other's states, thus improving overall simulation performance and scalability [122].

Another key benefit is the simplified synchronization and state management. In the Actor model, each actor maintains its state and interacts with others solely through message passing. This isolation minimizes the risks of race

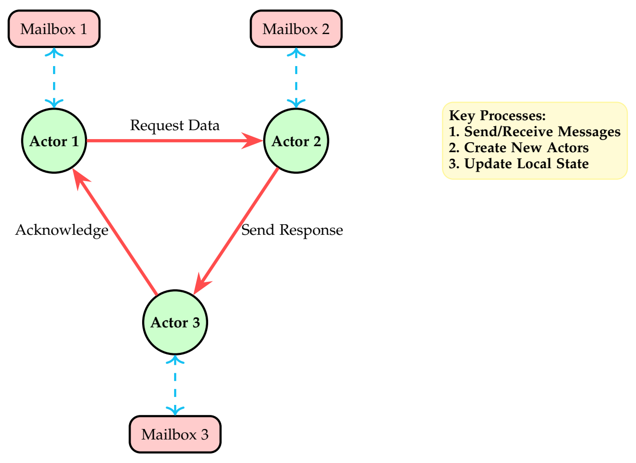

Fig. 8: The Actor Model of Computation

conditions and deadlocks. For instance, when a vehicle interacts with a junction, it sends a message requesting permission to proceed, eliminating the need for complex locking mechanisms. This leads to cleaner, more maintainable code and reduces the complexity of managing shared resources [121].

8.1.2.4 Drawbacks of the Actor Model: However, there are notable drawbacks to using the Actor model for this simulation. One primary concern is the overhead associated with message passing. The simulation involves frequent interactions between many actors, such as vehicles requesting access to junctions or updating their routes. This high volume of messages can introduce significant latency and processing overhead, potentially slowing down the simulation, especially when dealing with real-time constraints like traffic light synchronization and vehicle movements [123].

The Actor model can also lead to resource contention and bottlenecks, particularly at heavily trafficked junctions. Junction actors may become overwhelmed with incoming messages from numerous vehicles, limiting their ability to process requests efficiently. This can hinder scalability and degrade performance, as the throughput of these critical actors directly impacts the overall simulation. Furthermore, managing many actors can result in high memory consumption, posing challenges for simulations that require hundreds of thousands of actors to model extensive road networks [124].

The Actor pattern can also make traceability difficult, as we might struggle to understand the overall nature of the program. Debugging tools might help, but a measure of concurrency within the program would make it challenging to perform root cause analysis [125].

The actor pattern is the preferred pattern over the design strategies due to its ability to handle dynamism; the actor pattern should also be the preferred pattern for a variety of simulation problems; geometric decomposition is a clever and concise parallel design but should be more suited for computations which are predetermined beforehand, although we discussed the limitations of graph partitioning, this is not the most crucial problem of using this strategy the main issues that arise is synchronization and how to write code that cannot be solved in a linear sequence [126].

8.1.2.5 Impact on Hardware: The Actor model profoundly impacts the selection of hardware platforms due to its inherently concurrent and decentralized nature. Hardware that excels in parallel processing and efficient inter-process communication is essential for optimal performance. Multi-core and many-core processors and distributed computing systems are well-suited because they can assign actors to separate processing units, enabling precise concurrent execution [124]. Additionally, efficient and high-performance networks would be needed, especially as the scale grows; for a large simulation, writing the program on a consumer-grade system would be very inefficient due to the limited number of cores and also the synchronization of those, as throughput can be a significant determinant of the performance of actor-based systems.

Implementing the Actor model requires a message-passing library, which makes hardware choice a critical factor. Distributed computing architectures like supercomputers are ideal for achieving maximum scalability. While there is no definitive processor for implementing this model—since most modern multi-core processors meet the basic requirements—careful consideration should be given to selecting the most suitable compiler and compiler flags for the architecture to maximize efficiency and also maximize the utility of resources [127].

8.1.2.6 Implementation Considerations: As all actors act independently, we can identify them by their ID; we can map an actor to a process or share the actors across processes and have a hashing procedure to help us understand which process to direct a message to. A hashing procedure maps an actor's unique ID to a specific shard or process by applying a hash function, ensuring that all messages intended for a particular actor are consistently directed to the same process. This method enables efficient and scalable message routing, as the hash function distributes actors evenly across available shards, preventing load imbalances and reducing the likelihood of bottlenecks [128].

By utilizing a hashing-based approach for actor-to-process mapping, the simulation can achieve high performance and scalability, effectively managing communication and resource allocation even as the number of actors increases substantially. This strategy optimizes computational resources, ensuring the system remains responsive and resilient under varying loads, which is crucial for accurately modeling complex and large-scale road traffic scenarios [129].

In the geometric decomposition section, we discussed libraries that assist with graph partitioning for parallel programming. Some libraries help implement the actor model, but they have limitations. Libactor is one of them, catering to C programming and using threads for concurrency. However, it is not designed for distributed memory architectures. If we were to apply the Actor Model to the road simulation, we would need to use message-passing libraries [121].

The most suitable library in this context would be MPI (Message Passing Interface). While MPI is not ideal for the Actor Model, as actors are supposed to create other actors dynamically for this simulation, which is challenging in MPI due to its fixed number of processes, it is still feasible. Additionally, while the Actor Model relies heavily on asynchronous, fire-and-forget messaging, and MPI does support non-blocking communication, it is not as intuitive or straightforward as in dedicated actor systems.

For the development of this program, C++ is recommended due to its support for object-oriented programming through classes, which effectively abstract the definitions of the Actor Model and provide a high-level representation of properties. This abstraction facilitates more accessible and efficient program development. In contrast, C relies on structs, which offer greater granular precision in defining components. Nevertheless, C++ affords sufficient granular control over the program while maintaining a higher level of abstraction than C. The advantages of C++'s abstraction mechanisms render the increased abstraction non-problematic, except in scenarios where maximum speed and efficiency are critical [130]. Still, the rationale for the C++ recommendation is that we would need a base of abstraction in which we can build foundational classes rather than structs that interact more cohesively, as building these foundations with C structs would require significant nuance and take a lot of development time as we would have to create our ideologies from scratch. It would require more effort to build these programs; however, Rust may also be considered if expertise in the language is available, as it is a newer language with complex features that require careful programming. Other languages like Python and Go are impractical for programs that handle extensive simulations. Although it is possible to write simulations in these languages, the high levels of abstraction and the need for precise control would exacerbate efficiency deficiencies. Later, in this paper, we would discuss why high-level programming languages are not suitable for writing this road traffic simulation using a cellular automaton program as the subject of the experiment [131].

High-Level Implementation Framework

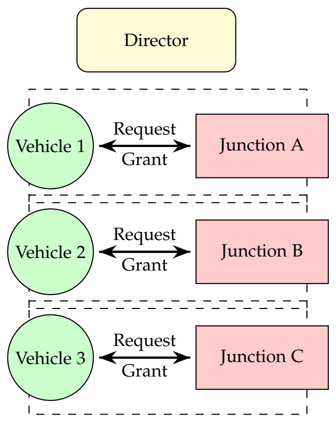

Fig. 9: Mapping of Road Traffic Simulation Using the Actor Model

8.1.2.7 Conclusion: It is important to note that the actor model is a powerful and precise model of computation, which makes it very important for the programmer to know what they are doing, as it is easy to produce subpar or incorrect code due to the complexity of implementing the Actor Model, which can make debugging and testing very time-consuming [126]. Still, due to the problem's specifications, the Actor Model would be the most appropriate parallel strategy. The simulation is very dynamic, and numerous operations are happening at once. The future actions of the program are not predictable; it is not like a parallel heat equation solver or a cellular automaton problem, which would require writing the domain code and handling synchronization between processes; this problem is not as linear as those problems and would require significant complexity in synchronization. The reason why the Actor model is appropriate for this problem is that each actor manages itself independently and only acts on the messages it receives; we can program a message struct with numerous enums for each actor and decide the course of action based on the message enum and the variables within the struct [132].

The Actor Model presents a robust and scalable parallel design strategy suitable for dynamic and complex simulations like road traffic modeling. Its ability to handle high concurrency, simplify synchronization, and maintain scalability makes it an ideal choice despite drawbacks such as message passing overhead and potential resource contention [133]..

8.2 Inappropriate Design Patterns

8.2.1 Pipeline

8.2.1.1 Overview of the Pipeline Design Pattern: Pipelining is a technique that enhances processing performance by dividing tasks into sequential stages, allowing multiple operations to overlap and execute concurrently. This procedure is used in various fields to improve efficiency and is prominently evident in computer science, particularly in parallel computing.

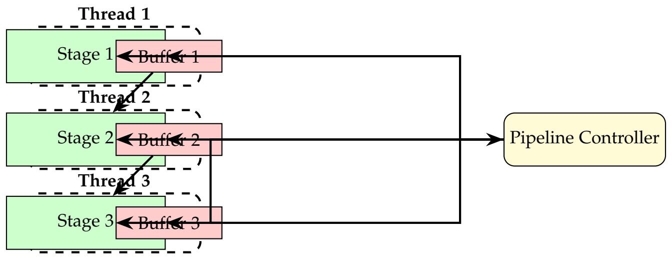

Fig. 10: Pipeline Design Pattern in Parallel Computing Using Multiple Threads

8.2.1.2 Unsuitability of Pipeline for Road Traffic Simulation: A useful heuristic for determining whether the pipeline design strategy is appropriate is to assess if the overall computation involves performing calculations on multiple datasets that flow through a sequence of stages. This approach leverages concurrency in linear processes, similar to parallel computing [134].

Applying this heuristic reveals that the current problem is not well-suited for the pipeline pattern. Although the issue is somewhat linear due to unidirectional graphs, dividing the computation into stages is impractical due to the dynamic nature of the simulation. While each junction could be represented as a pipeline stage, the computations at each intersection vary and depend on unpredictable sequences of events. The pipeline strategy is effective when there is a predetermined sequential order of operations at each step, which is not the case here [135]. Consequently, it is unlikely to be an appropriate strategy for this or similar simulation problems. We discussed in the earlier sections that geometric decomposition is not suitable for not being able to predetermine computations effectively as simulations typically deal with randomness; we cannot account for randomness in the pipeline strategy as it is meant to follow a determined route [136].

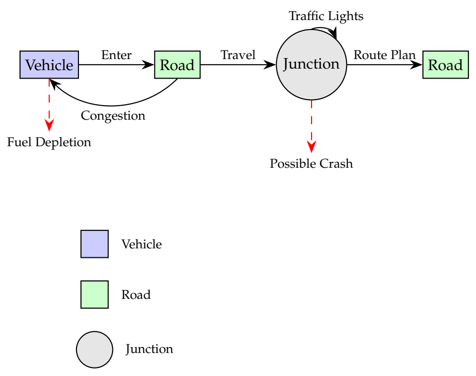

Fig. 11: Illustration of Complex Dependencies in the Road Simulation Model That Cannot Be Modeled Using a Pipeline

As shown in Figure 11, the road traffic simulation's complex dependencies demonstrate why a pipeline strategy would not be ideal for this problem.

Even if we somehow managed to implement the pipeline strategy, a significant question arises: how to map the pipeline stages effectively. Suppose we utilize OpenMP tasks and tasking features, even though pipelining isn't inherently supported. In that case, we could attempt to use OpenMP to manage the pipeline stages [137]. However, this raises the question: Should we treat each junction as a pipeline stage by assigning a separate thread to each stage? For a medium-sized simulation with 20,000 junctions, this approach would require 20,000 threads, which is impractical. The same issue arises when using processes, as the mapping would render this design strategy unsuitable for the problem [138].

Synchronization between stages in the pipeline could eventually become a bottleneck. Another point is that pipelining is typically ideal for programs when broken into stages; each stage is equally computationally expensive or has an equal computational load. If one stage in the pipeline generally varies widely from the median, the slowest stage would become a bottleneck. In the case of the road traffic simulation, we are not aware beforehand of the computation distribution of the problem, hindering the ability to load balance effectively [139].

8.2.2 *Recursive Data*

8.2.2.1 Overview of the Recursive Data Design Pattern: The recursive data design strategy is a programming approach where data structures are defined in terms of themselves. This self-referential definition allows for the creation of complex, hierarchical data models like linked lists, trees, and nested objects. Using recursion, developers can simplify handling data that naturally fits into nested or hierarchical patterns, making code more intuitive and easier to manage for specific problems [140].

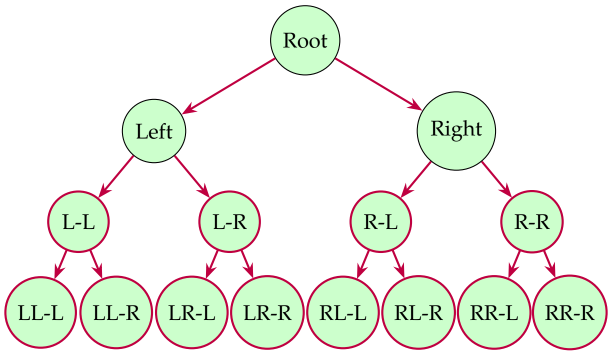

Fig. 12: Example of Recursive Data on an Extended Binary Tree

A Recursive Data pattern operates on a sequential data structure. As Figure 12 highlights, it would be appropriate for programs requiring sequential processing. It can also be used for graph processing, but the main reason it isn't suitable for the current simulation problem is the aspect of the simulation dynamics.

8.2.2.2 Unsuitability of Recursive Data for Road Traffic Simulation: Understanding that the recursive data problem involves a distributed recursive function is essential. Problems that can be solved recursively on a single processing element (PE) could, in theory, be adapted to a distributed recursive data pattern. However, when considering the specific problem it cannot be solved recursively. Additionally, writing a recursive program to solve a simulation problem is currently not feasible.

Implementing the recursive strategy in a serial context could involve a depth-first search algorithm on a tree, where the program reaches its base case and then pops off elements from a stack to combine the results.

In a parallel context, it could imply a parallel depth-first or breadth-first search in a graph where parallelism occurs at each node and its neighbors, allowing multiple edges to be processed simultaneously. This strategy would be excellent for standardized graph problems with deterministic computations at each node, as it will enable the time complexity to be broken down from an $O(N)$ algorithm to an $O(\log N)$ algorithm. However, this is not the case for the current problem because, similar to why the pipeline strategy is unsuitable, we cannot predict what computations will be done at each stage. We do not know the calculations that would be done at each edge of the graph, as different simulation parameters can lead to computations that cannot be accounted for beforehand [141]–[143].

Another rationale is why recursive data is not a suitable approach. The simulation represents a network of roads and junctions modeled as a graph, where roads are unidirectional edges and junctions are nodes. Vehicles move through this network, and their interactions depend on dynamic factors like traffic lights, congestion, and fuel levels. The road network's simulation dynamics do not naturally lend themselves to recursive data modeling because it involves complex, non-hierarchical relationships and cycles [144]–[146].

Using recursive data structures to model the road network could lead to complications such as circular references, which are challenging to manage and cause issues like infinite loops or stack overflows [147], [148]. Additionally, the simulation requires efficient algorithms for route planning, congestion management, and vehicle movement, all of which benefit from iterative processing and data structures optimized for graph traversal, such as adjacency lists or matrices [149], [150].

Moreover, the simulation involves real-time updates to the state of vehicles, roads, and junctions. Vehicles need to recalculate routes based on current road speeds and traffic light statuses, and the system must handle events like crashes and fuel depletion. Implementing these dynamic behaviors using recursive data design would introduce unnecessary complexity and potential performance bottlenecks [151], [152]. Iterative algorithms are better suited for handling the mutable state and frequent updates required in the simulation

The rationale for choosing the Actor model stems from the significant synchronization required in this program, which would be unsuitable for linear computation models. Recursive data patterns would struggle to handle this level of synchronization, and even if we attempted to use such a pattern, structuring the graph to integrate it effectively would be highly impractical [153], [154]. Integrating this pattern within this framework is likely more impractical than using the pipeline design pattern.

8.3 Approaches to Evaluate Parallel Code

To evaluate the proposed parallel program implementing the actor model for this road simulation using MPI, we would need to employ a combination of performance metrics, analytical methods, and specialized tools to thoroughly assess its efficiency and scalability [155]. Key metrics such as running time, speedup, parallel efficiency, and convergence properties provide valuable insights into how well the parallelization performs relative to the original serial code [156]; good experimental practice also has to take place to ensure the reliability and consistency of findings [157]. Additionally, understanding the optimal mapping of actors to processors, knowing how to segment the actors efficiently that allows clean code and scalability, and determining the optimal processor count is also crucial for maximizing performance [158].

8.3.0.1 Measuring Running Time and Speedup: Firstly, measure the **running time** T_p of the parallel program on p processors. By comparing this to the running time of the serial program T_1, compute the **speedup** S_p, defined as [159]:

$$S_p = \frac{T_1}{T_p}$$

This metric would help indicate how much faster the parallel program runs than the serial version. Ideally, we should aim for linear speedup, where $S_p = p$, but in practice, the speedup is often sub-linear due to overheads such as communication and synchronization and other factors that are not accounted for [160].

8.3.0.2 Calculating Parallel Efficiency: We should also calculate the **parallel efficiency** E_p using [161]:

$$E_p = \frac{S_p}{p} = \frac{T_1}{p \cdot T_p}$$

Parallel efficiency measures how effectively the processors are utilized in the parallel program. A high efficiency close to 1 signifies that the processors are being used optimally, whereas a low efficiency indicates overheads diminishing the benefits of parallelization [162]. Analyzing E_p assesses the program's scalability and identifies diminishing returns as the processor count increases; this would help identify the optimal processor count and also unveil insights into our program that may not be obvious.

8.3.0.3 Utilizing Performance Analysis Tools: We also need to thoroughly evaluate these metrics and utilize various **performance analysis tools** designed explicitly for parallel programs [163]. Numerous profiling tools can provide insights about the bottlenecks in our parallel program; there is an MPI profiling tool called Scalasca [164], which can pinpoint hotspots and areas for optimization within our code. We can use this to understand tradeoffs in our implementation details, such as using persistent communication rather than point-to-point communication. These tools can profile MPI applications to identify bottlenecks in communication and computation [165]. They collect data on MPI function calls, message sizes, and frequencies, helping understand the overhead introduced by inter-process communication.

8.3.0.4 Optimizing Actor-to-Processor Mapping: We can also examine the **optimum mapping of actors to processors** [166]. In the actor model implemented with MPI, actors represent entities such as vehicles, roads, or junctions, and their mapping to processors affects communication patterns and load balancing [167]. Experiment with different mapping strategies, such as:

- **Geographic Partitioning**: Assigning actors based on their spatial locality in the simulation to minimize communication between processors [168].
- **Load Balancing Algorithms**: Using dynamic load balancing techniques to distribute actors evenly across processors to prevent some processors from becoming bottlenecks [169].

8.3.0.5 Assessing Convergence Properties: The most essential thing in every computational process is the result. If the result is not accurate, then the whole program is not useful, which is why we would need **convergence properties** to ensure that parallelization does not adversely affect the accuracy or stability of the simulation results [170]. Verify that the parallel program produces results consistent with the serial version by comparing key outputs [171].

8.3.0.6 Conducting Scalability Testing: Another crucial component is to conduct **scalability testing**. Perform both **strong scaling** and **weak scaling** tests [172]. This would help evaluate the parallel program's performance under different scaling scenarios and provide us with insights that other evaluations couldn't.

All these tools should be used strategically, as they complement each other and don't work in isolation; they could be used interchangeably. It should be known when to use a tool to evaluate and conclude the result of another tool, as these would provide the most insights and intuitions about the program.

9 Conclusion

Parallel computing is a central aspect of modern computer science applied to anything from mundane smartphone to supercomputer usage. Following its path of evolution back to the very origins—theoretical foundations of PRAM and BSP to modern multi-core and heterogeneous systems—it is possible to appreciate both the principles that are the foundations of parallelism and the practical considerations needed to parallelize successfully [80].

Central to this is the understanding that parallel performance is a function of several interrelated factors. The intrinsic character of a problem, notably the ease with which a problem can be decomposed into parts that can then be distributed among tasks, usually determines the degree to which parallelization can provide meaningful speedups [160]. Additionally, hardware elements like CPUs, caches, memory, and special-purpose accelerators add to the complications since load balancing, cache coherence, and memory hierarchy directly impact the efficiency of execution [49]. No less significant are the software models of programming—processes, threads, message passing, actor-oriented designs, and a broad range of parallel patterns (such as geometric decomposition, pipeline, and recursive data)—each contributing models of coordinating work among distributed concurrent environments [35].

Beyond foundational principles, this effort also strongly emphasizes quality tooling and a supportive environment. Profiling and debugging software like Intel VTune, Allinea DDT, or NVIDIA Nsight, combined with well-engineered programming frameworks (MPI, OpenMP and CUDA), supply the capabilities to develop, refine, and support scalable parallel applications [173]. Concurrent with this are shared knowledge bases and information that supply the means to learn constantly and improve together. As the domain pushes ahead—confronting challenges of exascale computing, parallelism in the cloud, and AI workloads—these parallelization skills will increasingly become paramount to master. Understanding problem decompositions, hardware-software interactions, and contemporary debugging and optimizing techniques is not merely advantageous to the domain of HPC but also echoes all areas of the computing domain. Computer science can tackle the concurrent future with strong, scalable parallel programs by incorporating historical knowledge, theoretic models, and practical applications into their designs [174].

References

[1] K. M. Chandy, "Programming parallel computers," in *IEEE International Conference on Computer Languages*, 1988.

[2] T. Mattson, B. Sanders, and B. L. Massingill, *Patterns for Parallel Programming*, 2004.

[3] R. Sen, "Developing parallel programs," *Advances in Computer Science: an International Journal*, vol. 1, pp. 18–27, 2012.

[4] E. Gallopoulos, B. Philippe, and A. Sameh, *Parallel Programming Paradigms*, 2016.

[5] C. Fields, *Introduction to Parallel Programming*, 2022.

[6] S. Rastogi and H. Zaheer, "Significance of parallel computation over serial computation," in *2016 International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT)*, 2016, pp. 2307–2310.

[7] M. M, V. Hajjarge, V. V. Divate, K. Mohan, and A. Kanavalli, "Efficiency comparison of parallel and serial computation techniques for multi-regional weather data aggregation," in *2024 International Conference on Intelligent and Innovative Technologies in Computing, Electrical and Electronics (IITCEE)*, 2024, pp. 1–6.

[8] W. Yue, "Mathematical models and algorithm description of parallelizing serial algorithm," *Computer Technology and Development*, 2012.

[9] Y. Jian, "Parallel computing for complex flows," *Hydro-Science and Engineering*, 2002.

[10] Z. Hua, "Numerical simulation and parallel computation of flow over object using fluent," *Computer Engineering and Design*, 2005.

[11] A. B. Rathod, R. Khadse, and M. F. Bagwan, "Serial computing vs. parallel computing: A comparative study using matlab," 2014.

[12] G. Nikolic, B. Dimitrijevic, T. Nikolic, and M. Stojcev, "Fifty years of microprocessor evolution: from single cpu to multicore and manycore systems," *Facta Universitatis - Series: Electronics and Energetics*, 2022.

[13] P. Gepner and M. Kowalik, "Multi-core processors: New way to achieve high system performance," in *International Symposium on Parallel Computing in Electrical Engineering (PARELEC'06)*, 2006, pp. 9–13.

[14] G. D'Angelo and M. Marzolla, "New trends in parallel and distributed simulation: From many-cores to cloud computing," *ArXiv*, vol. abs/1407.6470, 2014.

[15] G. Chennupati, R. M. A. Azad, and C. Ryan, "Multi-core ge: automatic evolution of cpu based multi-core parallel programs," in *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation*, 2014.

[16] S. Ghose, "General-purpose multicore architectures," *ArXiv*, vol. abs/2408.12999, 2024.

[17] P. Hall, "Foundations of parallel algorithms pram model time, work, cost self-simulation and brent's theorem speedup," in *Multicore Computing Lecture Series*, 2011.

[18] R. Karp, "A survey of parallel algorithms for shared-memory machines," 1988.

[19] D. Lecomber, C. J. Siniolakis, and K. R. Sujithan, "Pram programming: in theory and in practice," *Concurr. Pract. Exp.*, vol. 12, pp. 211–226, 2000.

[20] K. Mulmuley, "Lower bounds in a parallel model without bit operations," *SIAM J. Comput.*, vol. 28, pp. 1460–1509, 1999.

[21] J. D. Ullman and A. V. Gelder, "Parallel complexity of logical query programs," pp. 438–454, 1986.

[22] W. Mccoll, "Bsp programming 1. the bsp model," 1994.

[23] D. Culler, R. Karp, D. Patterson, A. Sahay, E. Santos, K. Schauser, R. Subramonian, and T. V. Eicken, "Logp: a practical model of parallel computation," *Commun. ACM*, vol. 39, pp. 78–85, 1996.

[24] D. T. Hasta and A. Mutiara, "Performance evaluation of parallel message passing and thread programming model on multicore architectures," *ArXiv*, vol. abs/1012.2273, 2010.

[25] S. Hua and Z. Yang, "Comparison and analysis of parallel computing performance using openmp and mpi," *The Open Automation and Control Systems Journal*, vol. 5, pp. 38–44, 2013.

[26] T. Hilbrich, M. S. Müller, and B. Krammer, "Mpi correctness checking for openmp/mpi applications," *International Journal of Parallel Programming*, vol. 37, pp. 277–291, 2009.

[27] H. Jin and G. Jost, "Support of multidimensional parallelism in the openmp programming model," in *International Conference on High-Performance Computing and Networking*, 2003, pp. 511–522.

[28] R. Bocian, D. Pawlowska, K. Stencel, and P. Wisniewski, "Openmp as an efficient method to parallelize code with dense synchronization," pp. 120–127, 2018.

[29] G. Zhang, "Extending the openmp standard for thread mapping and grouping," in *International Conference on OpenMP*, 2005, pp. 435–446.

[30] J. Shirako, K. Sharma, and V. Sarkar, "Unifying barrier and point-to-point synchronization in openmp with phasers," in *International Conference on High-Performance Computing*, 2011, pp. 122–137.

[31] A. Mahéo, S. Koliai, P. Carribault, M. Pérache, and W. Jalby, "Adaptive openmp for large numa nodes," in *International Conference on High-Performance Computing and Simulation*, 2012, pp. 254–257.

[32] V. Rajput and A. Katiyar, "Proactive bottleneck performance analysis in parallel computing using openmp," in *ArXiv*, vol. abs/1311.1907, 2013.

[33] I. T. Foster, *Designing and Building Parallel Programs*. Addison-Wesley, 1995.

[34] M. J. Quinn, *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, 2004.

[35] M. McCool, A. D. Robison, and J. Reinders, *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann, 2012.

[36] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*, 2nd ed. Morgan Kaufmann, 2020.

[37] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. Purcell, "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, 2007.

[38] M. Kulkarni, M. J. Bridges, R. L. Bocchino, S. A. Edwards, S. V. Adve, V. S. Adve, and D. A. Padua, "Optimistic parallelism requires abstractions," in *Conference on Programming Language Design and Implementation (PLDI)*, 2007, pp. 211–222.

[39] A. P. Liavas and B. D. Rao, "Parallel algorithms for sparse matrix factorization and their applications," *IEEE Transactions on Signal Processing*, vol. 46, no. 9, pp. 2429–2442, 1998.

[40] R. Greenlaw, H. J. Hoover, and W. L. Ruzzo, *Limits to Parallel Computation: P-Completeness Theory*. Oxford University Press, 1995.

[41] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[42] J. Blythe, E. Amir, E. Kamar, and Y. Gil, "Task planning for collaborative execution: A distributed, parallel approach," *Journal of Artificial Intelligence Research*, vol. 18, pp. 365–408, 2003.

[43] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. MIT Press, 2022.

[44] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[45] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *IEEE Computer*, vol. 41, no. 7, pp. 33–38, 2013.

[46] B. Barney, "Introduction to parallel computing," *Lawrence Livermore National Laboratory*, 2010.

[47] C. Fu and T. Yang, "Space and time efficient execution of parallel irregular computations," *ACM Symposium on Parallel Algorithms and Architectures*, vol. 32, pp. 57–68, 1997.

[48] X. Song and Y. Chen, "Parallel memory management for large-scale scientific computation," in *International Conference on Parallel and Distributed Systems*, 2009, pp. 537–544.

[49] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed. Morgan Kaufmann, 2011.

[50] C. Fagundes and R. Fernandes, "Evaluation of cache efficiency in parallel processing," *Journal of Parallel and Distributed Computing*, 2024.

[51] L. Arge, "Fundamental parallel algorithms for cache-efficient computing," *ACM Transactions on Algorithms*, vol. 4, no. 3, pp. 1–30, 2008.

[52] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, 2nd ed. MIT Press, 1999.

[53] R. Thakur, W. Gropp, and E. Lusk, "Optimization of collective communication operations in mpich," *International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005.

[54] Z. Huda, "Identification of suitable parallelization patterns for sequential programs," in *Thesis, TU Darmstadt*, 2021.

[55] J. Michalakes, J. Purser, and R. Swinbank, "Data structure and parallel decomposition considerations on a fibonacci grid," 1999.

[56] L. Rinaldi, M. Torquati, G. Mencagli, M. Danelutto, and T. Menga, "Accelerating actor-based applications with parallel patterns," *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pp. 140–147, 2019.

[57] M. Aguilar, R. Leupers, G. Ascheid, and J. F. E. Giraldo, "Extraction of recursion level parallelism for embedded multicore systems," in *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, 2017, pp. 154–162.

[58] A. Navarro, R. Asenjo, S. Tabik, and C. Cascaval, "Analytical modeling of pipeline parallelism," in *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, 2009, pp. 281–290.

[59] R. Collins, "Data-driven programming abstractions and optimization for multi-core platforms," 2011.

[60] F. Javier, M. R. A. Moreno Arboleda, and J. A. Hernández Riveros, "Performance of parallelism in python and c++," 2023.

[61] H.-C. Park, J. DeNio, J. Choi, and H. Lee, "mpipython: A robust python mpi binding," in *2020 3rd International Conference on Information and Computer Technologies (ICICT)*, 2020, pp. 96–101.

[62] H. Elshazly, F. Lordan, J. Ejarque, and R. M. Badia, "Performance meets programmabilty: Enabling native python mpi tasks in pycompss," in *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 2020, pp. 63–66.

[63] L. Dalcin, R. Paz, P. A. Kler, and A. Cosimo, "Parallel distributed computing using python," *Advances in Water Resources*, vol. 34, pp. 1124–1139, 2011.

[64] V. Zosimov and O. Bulgakova, "Optimizing computational performance with openmp parallel programming techniques," *Control Systems and Computers*, 2023.

[65] F. G. Tinetti and M. Ruiz, "Efficient cache utilization in parallel systems," *Journal of Parallel and Distributed Computing*, vol. 73, pp. 1023–1036, 2013.

[66] G. C. Fox and M. Johnson, *Parallel Computing: Theory and Practice*. Morgan Kaufmann, 1991.

[67] A. Cheptsov and M. Fiedler, "Efficient parallel programming for high-performance applications," *Journal of Supercomputing*, vol. 65, pp. 674–690, 2013.

[68] P. Bientinesi and S. Kromer, "Accelerator-based parallel computing: Gpus, fpgas, and beyond," *ACM Computing Surveys*, vol. 47, pp. 1–36, 2015.

[69] Y. Khan and I. Ali, "Multi-core cpu architectures and their impact on parallel computing," *IEEE Transactions on Computers*, vol. 61, pp. 1451–1462, 2012.

[70] K. Dezhgosha, "Cpu architectures and performance in parallel systems," *Parallel Computing Journal*, vol. 22, pp. 1213–1230, 1996.

[71] M. Amoretti, "Homogeneous and heterogeneous architectures in high-performance computing," *Concurrency and Computation: Practice and Experience*, vol. 32, p. e5520, 2020.

[72] D. Kimpe and R. Ross, "Memory management strategies for distributed parallel systems," *Journal of Parallel and Distributed Computing*, vol. 66, pp. 1508–1523, 2006.

[73] S. Jong and H. Lee, "Memory optimization techniques in heterogeneous systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, pp. 1053–1064, 2010.

[74] Z. Hao, "Ai accelerators and their role in modern parallel computing," *IEEE Transactions on Computers*, vol. 72, pp. 1534–1547, 2023.

[75] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 3, pp. 260–274, 2002.

[76] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[77] I. Corporation, *Intel VTune Profiler User Guide*, 2022, https://www.intel.com/content/www/us/en/develop/documentation/vtune-cookbook/top.html.

[78] N. Corporation, *NVIDIA Nsight Systems User Guide*, 2022, https://developer.nvidia.com/nsight-systems.

[79] J. Rogers, *TotalView User Guide*, 2003, https://www.perforce.com/products/totalview.

[80] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[81] K. S. Perumalla, "Parallel and distributed simulation: Traditional techniques and recent advances," *Proceedings of the 38th conference on Winter simulation*, pp. 84–95, 2006.

[82] E. S. Raymond, *The cathedral and the bazaar: Musings on Linux and open source by an accidental revolutionary*. O'Reilly Media, 1999.

[83] E. A. Von Hippel, "Democratizing innovation," *MIT Press*, 2003.

[84] O. A. R. Board, *OpenMP Application Programming Interface Version 5.1*, 2021.

[85] R. Smith and M. Williams, *OpenMP in Practice: Portable Multi-Level Parallelism on Modern Systems*. MIT Press, 2011.

[86] G. E. Blelloch, "Programming parallel algorithms," *Communications of the ACM*, vol. 39, no. 3, pp. 85–97, 1996.

[87] K. M. Hoffman *et al.*, *Mastering CMake*. Kitware, Inc., 2016.

[88] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.

[89] L. Liu and X. Chen, "Automated performance monitoring and diagnosis in cloud environments," *IEEE Transactions on Cloud Computing*, vol. 5, no. 4, pp. 810–821, 2017.

[90] A. Smyk and M. Tudruj, "Optimization of parallel fdtd computations based on program macro data flow graph transformations," in *International Conference on Parallel Processing and Applied Mathematics (PPAM)*, 2011.

[91] R. Garmann, "Maintaining dynamic geometric objects on parallel processors," in *Proceedings IEEE Symposium on Parallel Rendering (PRS'97)*, 1997, pp. 31–38.

[92] K. Bourgeois, S. Robert, S. Limet, and V. Essayan, "Efficient implicit parallel patterns for geographic information system," *Procedia Computer Science*, vol. 108, pp. 545–554, 2017.

[93] D. Paulius and M. Boumédine, "On the scalability of parallel quicksort: A case study on distributed vs. shared-memory models," in *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, 2015.

[94] I. Moulitsas, Y. Saad, and G. Karypis, "Graph partitioning for scientific computing applications," 2005.

[95] F. Pellegrini, "Current challenges for parallel graph partitioning and static mapping," 2011.

[96] R. Engelen, "Graph partitioning for high performance scientific simulations," 2000.

[97] G. Karypis, "Graph partitioning for high performance scientific simulations," 2000.

[98] M. Predari and A. Esnard, "Graph partitioning techniques for load balancing of coupled simulations," 2016.

[99] J. Miller, L. Trümper, C. Terboven, and M. S. Müller, "A theoretical model for global optimization of parallel algorithms," 2021.

[100] A. Svitenkov, P. Zun, O. Rekin, and A. Hoekstra, "Partitioning of arterial tree for parallel decomposition of hemodynamic calculations," 2016.

[101] S. Kirmani, H. Sun, and P. Raghavan, "A scalability and sensitivity study of parallel geometric algorithms for graph partitioning," 2018.

[102] L. Axner, J. Bernsdorf, T. Zeiser, P. Lammers, J. Linxweiler, and A. Hoekstra, "Performance evaluation of a parallel sparse lattice boltzmann solver," 2008.

[103] X. Guo, Y. Wang, and J. Killough, "The application of static load balancers in parallel compositional reservoir simulation on distributed memory system," 2016.

[104] M. Goodrich, "46 parallel algorithms in geometry," 2016.

[105] B. Chamberlain, "Graph partitioning algorithms for distributing workloads of parallel computations," 2001.

[106] J. Zheng, Z. Wang, Z. Xie, X. Peng, C. Zhao, and W. Wu, "Parallel communication optimization based on graph partition for hexagonal neutron transport simulation using moc method," 2023.

[107] G. Karypis and V. Kumar, "A fast and high-quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[108] ——, "Multilevel k-way hypergraph partitioning," *VLSI Design*, vol. 11, no. 3, pp. 285–300, 1999.

[109] J. Chen, P. Raghavan, and R. Ramanathan, "Partitioning large graphs for distributed memory parallel computing: Recent advances and challenges," *Parallel Computing*, vol. 33, no. 9, pp. 550–566, 2007.

[110] D. A. Bader, *Parallel Algorithms for Regular Architectures: Meshes and Hypercubes*. SIAM, 2006.

[111] T. Hoefler and A. Lumsdaine, "Performance modeling for distributed-memory applications," *Journal of Parallel and Distributed Computing*, vol. 70, no. 12, pp. 1140–1150, 2010.

[112] T. Rauber and G. Rünger, *Parallel Programming: for Multicore and Cluster Systems*. Springer Science & Business Media, 2013.

[113] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*. Addison-Wesley, 2003.

[114] A. S. Tanenbaum and M. Van Steen, *Distributed Systems: Principles and Paradigms*. Pearson Education, 2007.

[115] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2017.

[116] V. Kumaran, Y. Wang, and I. Stojmenovic, "Performance evaluation of parallel road traffic simulation," *Journal of Parallel and Distributed Computing*, vol. 63, no. 3, pp. 300–310, 2003.

[117] D. B. Kirk and W.-M. W. Hwu, *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016.

[118] S. Cook, *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Elsevier, 2019.

[119] G. E. Karniadakis and R. M. Kirby II, *Parallel Scientific Computing in C++ and MPI: A Seamless Approach to Parallel Algorithms and Their Implementation*. Cambridge University Press, 2003.

[120] C. Hewitt, "A universal modular actor formalism for artificial intelligence," *IJCAI*, vol. 73, pp. 235–245, 1973.

[121] P. Haller and F. Sommers, *Actors in Scala*, 2012.

[122] Y. Hayduk, A. Sobe, D. Harmanci, P. Marlier, and P. Felber, "Speculative concurrent processing with transactional memory in the actor model," 2013.

[123] C. Scholliers, E. Tanter, and W. Meuter, "Parallel actor monitors: Disentangling task-level parallelism from data partitioning in the actor model," 2014.

[124] W. Kim, "Thal: An actor system for efficient and scalable concurrent computing," 1997.

[125] A. Rosà, L. Chen, and W. Binder, "Efficient profiling of actor-based applications in parallel and distributed systems," 2016.

[126] G. Agha and W. Kim, "Parallel programming and complexity analysis using actors," 1997.

[127] R. Hiesgen, D. Charousset, and T. Schmidt, "Opencl actors - adding data parallelism to actor-based programming with caf," 2017.

[128] D. Charousset, T. Schmidt, R. Hiesgen, and M. Wählisch, "Native actors: A scalable software platform for distributed, heterogeneous environments," 2013.

[129] J. D. Koster, S. Marr, T. V. Cutsem, and T. D'Hondt, "Domains: Sharing state in the communicating event-loop actor model," 2016.

[130] A. Masud, B. Lisper, and F. Ciccozzi, "Automatic inference of task parallelism in task-graph-based actor models," 2018.

[131] C. Hewitt, "Actor model of computation," 2015.

[132] Y. Latrous and G. Mazaré, "Distributing code in a parallel fine grain machine using the actor model," in *Proceedings Euromicro Workshop on Parallel and Distributed Processing*, 1995, pp. 122–129.

[133] K. Warwas and K. Augustynek, "Actor model approach to control stability of an articulated vehicle," pp. 41–51, 2016.

[134] A. Navarro, R. Asenjo, S. Tabik, and C. Cascaval, "Analytical modeling of pipeline parallelism," in *2009 18th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2009, pp. 281–290.

[135] S. A. Mirsoleimani, H. J. V. Herik, A. Plaat, and J. Vermaseren, "Pipeline pattern for parallel mcts," in *Proceedings of the 2018 International Conference on High Performance Computing & Simulation (HPCS)*, 2018, pp. 614–621.

[136] C.-H. Chiu and T.-W. Huang, "Composing pipeline parallelism using control taskflow graph," in *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2022.

[137] X. Liu, R. cai Zhao, and L. Han, "A compile-time cost model for automatic openmp decoupled software pipelining parallelization," in *2013 14th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, 2013, pp. 253–260.

[138] T. Preud'homme, J. Sopena, G. Thomas, and B. Folliot, "An improvement of openmp pipeline parallelism with the batchqueue algorithm," in *2012 IEEE 18th International Conference on Parallel and Distributed Systems*, 2012, pp. 348–355.

[139] D. Bell, J. Shao, and M. Hull, "A pipelined strategy for processing recursive queries in parallel," *Data Knowl. Eng.*, vol. 6, pp. 367–391, 1991.

[140] Z. Nawaz, "Recursive variable expansion: A transformation for reconfigurable computing," 2011.

[141] S. Nishimura and A. Ohori, "Parallel functional programming on recursively defined data structures," *Journal of Functional Programming*, vol. 9, no. 4, pp. 529–546, 1999.

[142] J. W. Ahn and Y. Han, "Analysis of parallelism in recursive functions on computer architectures," *Parallel Processing Letters*, vol. 17, no. 3, pp. 315–330, 2007.

[143] J. Misra, "Powerlist: A structure for parallel recursion," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 6, pp. 1737–1767, 1994.

[144] Y.-S. Hwang and J. Saltz, "Interprocedural definition-use chains of dynamic recursive data structures," 1998.

[145] A. Benczúr and B. Kósa, "Static analysis of structural recursion in semistructured databases and its consequences," 2004.

[146] J. Westbrook, "Algorithms and data structures for dynamic graph problems," 1989.

[147] F. Corbera, R. Asenjo, and E. Zapata, "Accurate shape analysis for recursive data structures," 2000.

[148] W. Arnold and R. Haydock, "A parallel, object-oriented implementation of the dynamic recursion method," 1998.

[149] L. Hendren and A. Nicolau, "Parallelizing programs with recursive data structures," 1989.

[150] A. Navarro, F. Corbera, R. Asenjo, R. Castillo, and E. Zapata, "A data dependence test based on the projection of paths over shape graphs," 2012.

[151] M. Bianchini, M. Gori, and F. Scarselli, "Recursive processing of cyclic graphs," 2002.

[152] N. Klarlund and M. Schwartzbach, "Graph types," 1993.

[153] Y.-S. Hwang and J. Saltz, "Identifying def/use information of statements that construct and traverse dynamic recursive data structures," 1997.

[154] D. Chu and J. Jaffar, "Local reasoning on recursive data structures," 2017.

[155] J. Dongarra, J. L. Martin, and S. Moore, "Performance evaluation and benchmarking of parallel and distributed computing tools," *Parallel Computing*, vol. 35, no. 3, pp. 141–143, 2009.

[156] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to parallel computing*. Benjamin/Cummings Publishing Company, 1994.

[157] R. Jain, *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling*. John Wiley & Sons, 1991.

[158] G. Agha and W. Kim, "Performance evaluation of actor oriented parallel systems," *Parallel Computing*, vol. 19, no. 8, pp. 859–872, 1993.

[159] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," *Proceedings of the April 18-20, 1967, spring joint computer conference*, pp. 483–485, 1967.

[160] J. L. Gustafson, "Reevaluating amdahl's law," *Communications of the ACM*, vol. 31, no. 5, pp. 532–533, 1988.

[161] D. L. Eager, J. Zahorjan, and E. D. Lazowska, "Speedup versus efficiency in parallel systems," *IEEE Transactions on computers*, vol. 38, no. 3, pp. 408–423, 1989.

[162] R. M. Karp and V. Ramachandran, "Optimal parallel algorithms for sparse matrix operations," *SIAM Journal on Computing*, vol. 19, no. 1, pp. 1–22, 1990.

[163] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach, "Vampir: Visualization and analysis of mpi resources," in *Supercomputer*, vol. 12, 1996, pp. 69–80.

[164] M. Geimer, F. Wolf, B. J. Wylie, E. Ábrahám, D. Becker, and B. Mohr, "The scalasca performance toolset architecture," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 702–719, 2010.

[165] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel, "The vampir performance analysis tool-set," in *Tools for High Performance Computing*. Springer, 2008, pp. 139–155.

[166] M. K. Vernon, J. Zahorjan, and E. D. Lazowska, "Performance evaluation of parallel processor computer systems," *IEEE Transactions on Software Engineering*, vol. 14, no. 4, pp. 501–513, 1988.

[167] R. Buyya, *High performance cluster computing: Architectures and systems*. Prentice Hall PTR, 1999, vol. 1.

[168] F. Pellegrini and J. Roman, "Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs," *High-Performance Computing and Networking*, pp. 493–498, 1996.

[169] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, and C. Vaughan, "Zoltan data management services for parallel dynamic applications," *Computing in Science & Engineering*, vol. 4, no. 2, pp. 90–96, 2002.
[170] J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon, and A. White, *Numerical libraries and tools for scalable parallel cluster computing*. Morgan Kaufmann, 2003.
[171] D. H. Bailey, "The parallel performance of robustness and convergence tests," *The Journal of Supercomputing*, vol. 5, no. 4, pp. 315–331, 1991.
[172] G. Hager and G. Wellein, *Introduction to high performance computing for scientists and engineers*. CRC Press, 2010.
[173] A. Shafi *et al.*, "Multi-core parallel programming using intel threading building blocks," *Concurrency and Computation: Practice and Experience*, vol. 21, no. 8, pp. 965–977, 2009.
[174] J. Dongarra *et al.*, "The international exascale software project roadmap," *International Journal of High Performance Computing Applications*, vol. 25, no. 1, pp. 3–60, 2011.

10 Appendix

10.1 Details of the Cirrus supercomputer

The Cirrus HPC system is located at the Uni- versity of Edinburgh. The system comprises 283 compute nodes, each with 2.1 GHz, 18-core Intel Xeon E5-2695 processors and 256 GB of memory. The Cirrus supercomputer features 36 GPU compute nodes. Each node is equipped with two Intel Xeon Gold 6148 (Cascade Lake) processors running at 2.5 GHz, with each processor containing 20 cores. These cores support 2 hardware threads (Hyperthreads) per core, which are enabled by default. Addi- tionally, each node contains four NVIDIA Tesla V100-SXM2-16GB (Volta) GPU accelerators, which are interconnected and connected to the host processors via PCIe. In total, the GPU compute nodes provide 144 GPU accelerators (4 GPUs × 36 nodes) and 1,440 CPU cores (40 cores × 36 nodes) across the system. The cache hierarchies for the compute nodes are detailed in Table 3

TABLE 1: Cache Hierarchies for CPU and GPU Nodes

(a) CPU Node Cache Hierarchy

Cache Level	**Size**
L1 Cache	32 KiB per core
L2 Cache	256 KiB per core
L3 Cache	45 MiB shared

(b) GPU Node Cache Hierarchy

Cache Level	**Size**
L1 Cache	32 KiB per core
L2 Cache	1 MiB per core
L3 Cache	27.5 MiB shared

10.2 Pseudocode for MPI-Based Cellular Automaton Simulation

Algorithm 1 Main Simulation Procedure

```
1: procedure MAIN
2:     $args \leftarrow$ PARSEARGUMENTS
3:     if $args.mode =$ "serial" $\vee$ MPI_Size() $= 1$ then
4:         RUNSIMULATIONSERIAL($args$)
5:     else
6:         RUNSIMULATIONPARALLEL($args$)
7:     end if
8: end procedure
```

Procedure: RunSimulationSerial($args$)

```
1: Set parameters: $L, \rho, seed, maxstep, printfreq \leftarrow args$
2: Initialize random generator with $seed$
3: $grid \leftarrow$ ZeroMatrix($L+2, L+2$)    ▷ Create grid with ghost boundaries
4: for each cell $(i, j)$ in $grid[1:L, 1:L]$ do
5:     Set $grid[i, j] \leftarrow 1$ with probability $\rho$, else 0
6: end for
7: $initial\_live \leftarrow$ Sum($grid[1:L, 1:L]$)
8: Print simulation parameters
9: Start timer
10: for $step = 1$ to $maxstep$ do
```

```
11:     UpdateGhostBoundaries($grid$)    ▷ Enforce periodic conditions
12:     ApplyBoundaryConditions($grid, L$)
13:     $neighbors \leftarrow$ ComputeNeighbors($grid$)    ▷ Sum of 4-adjacent cells
14:     $live \leftarrow$ UpdateCells($grid, neighbors$)    ▷ Cells become alive if neighbor count is in {2,4,5}
15:     if $step \mod printfreq = 0$ then
16:         Print current step and live cell count
17:     end if
18:     if $live < 0.75 \times initial\_live \vee live > 1.33 \times initial\_live$ then
19:         Print termination message and break
20:     end if
21: end for
22: Stop timer and compute average time per iteration
23: Write $grid[1:L, 1:L]$ to output file
```

Procedure: RunSimulationParallel($args$)

```
1: Set parameters: $L, \rho, seed, maxstep, printfreq \leftarrow args$
2: Initialize MPI and create Cartesian topology: obtain $comm, rank, size, dims, coords, cart\_comm$
3: Compute local dimensions: $(local\_rows, local\_cols) \leftarrow L/dims$
4: if $rank = 0$ then
5:     $global\_grid \leftarrow$ CreateRandomMatrix($L, L$) with probability $\rho$
6: end if
7: $global\_grid \leftarrow$ Broadcast($global\_grid$, root $= 0$)
8: $local\_grid \leftarrow$ ExtractSubgrid($global\_grid, coords, local\_rows, local\_cols$)
9: $grid \leftarrow$ CreateExtendedGrid($local\_grid, local\_rows, local\_cols$)
10: $local\_initial\_live \leftarrow$ Sum($local\_grid$)
11: $initial\_live \leftarrow$ ReduceSum($local\_initial\_live$) and Broadcast to all processes
12: if $rank = 0$ then
13:     Print simulation parameters and number of processes
14: end if
15: Barrier synchronization on $cart\_comm$
16: Start timer (MPI_Wtime)
17: for $step = 1$ to $maxstep$ do
18:     ExchangeHalos($grid, cart\_comm, local\_rows, local\_cols$)
19:     AdjustBoundaries($grid, coords, dims, local\_rows, local\_cols, L$)
20:     $neighbors \leftarrow$ ComputeNeighbors($grid$)
21:     $local\_live \leftarrow$ UpdateCells($grid, neighbors$)
22:     $total\_live \leftarrow$ ReduceSum($local\_live$) over all processes
23:     if $rank = 0 \wedge (step \mod printfreq = 0)$ then
24:         Print current step and $total\_live$
25:     end if
26:     if $total\_live < 0.75 \times initial\_live \vee total\_live > 1.33 \times initial\_live$ then
27:         if $rank = 0$ then
28:             Print termination message and break
29:         end if
30:     end if
31: end for
32: Stop timer and compute average time per iteration (on $rank = 0$)
33: $global\_result \leftarrow$ GatherSubgrids($grid[1 : local\_rows, 1 : local\_cols]$)
34: if $rank = 0$ then
35:     Write $global\_result$ to output file
36: end if
37: Finalize MPI
```

10.3 Details of the Road Simulation Model

- The road map is represented as a graph:
  - Junctions are nodes in the graph, and roads are edges between these nodes.
  - Individual roads (graph edges) are always unidirectional (one way), so always connect one junction (graph node) to another and not the other way round.
  - Whilst there will be routes connecting a specific junction to many others in the graph, there is no guarantee that there is a route between every pair of junctions.
  - The graph is provided to the model via an input file, which is read during initialisation.
- Roads are all different lengths, expressed in metres:
  - Roads also have a maximum vehicle speed and current speed. The current speed depends upon how congested the road is (i.e., the number of vehicles currently on the road).
  - The road's length and maximum vehicle speed are also specified in the initialisation file.
- Vehicles travel from one junction to another:
  - Vehicles are periodically added to the simulation, and at this point, their source (starting junction) and destination junction are set.
  - While the source and destination junctions are random, only those with valid routes between them will be selected.
- Junctions may have multiple roads connecting them to other junctions:
  - A junction may or may not have traffic lights.
  - If traffic lights are present:
    * Only one road connected to the junction can be used at any time.
    * Vehicles requiring other roads must wait until the traffic lights enable their desired road.
    * Traffic lights change each simulated minute, working in a round-robin manner to enable one road after another.
  - If no traffic lights are present, all roads connected to the junction can be used at all times.
  - Traffic light information is included in the initialisation file.
- Vehicles travel on roads and through junctions:
  - There are six types of vehicles: car, bus, minibus, coach, motorbike, and bike.

        * Each has different capabilities, including maximum speed and maximum passenger capacity.
        * These parameters are defined in the code.
        * When a vehicle is created, the number of passengers is randomly generated up to the maximum capacity.
    - While on a road, vehicles continually update their location based on their current speed.
        * The speed at which a vehicle travels depends on the road's speed when the vehicle left the junction and the vehicle's maximum speed (whichever is smaller).
        * Vehicles do not continuously check the road's current speed after entering it.
    - When a vehicle arrives at a junction:
        * If the junction is not its destination, the vehicle re-plans its route.
        * Route planning considers the road lengths and, for most roads, their maximum speed.
        * For roads connected to the current junction, the vehicle uses the current speed instead of the maximum speed.
        * If the junction has traffic lights, the vehicle waits until the selected road is enabled.
        * If the junction is the vehicle's destination, it is removed from the simulation.

- Vehicle crashes can occur at junctions without traffic lights:
    - All crashes involve a single vehicle (i.e., no multi-vehicle collisions).
    - Crashes only occur at junctions, never on roads.
    - The likelihood of a crash increases with the number of vehicles at the junction.
    - A crashed vehicle is removed from the simulation.
- Vehicles have a finite amount of fuel, which can run out:
    - When a vehicle is created, its fuel amount (expressed in seconds of running time) is randomly generated.
    - Fuel consumption is continuous, whether the vehicle is moving or waiting at a junction.
    - Vehicle speed does not affect fuel consumption.
    - When fuel runs out, the vehicle is removed from the simulation.
- The simulation periodically prints progress summaries to `stdio`, including:
    - Current simulation time (in minutes).
    - Total number of vehicles added to the simulation.
    - Total number of passengers delivered to their destination.
    - Total number of passengers stranded (due to crashes or fuel depletion).
    - Number of vehicle crashes.
    - Number of vehicles that have run out of fuel.
- The simulation terminates after a predetermined number of simulation minutes:
    - A final summary is printed to `stdio`, including:
        * Current simulation time (in minutes).
        * Total number of vehicles added to the simulation.
        * Total number of passengers delivered to their destination.
        * Total number of passengers stranded.
        * Number of vehicle crashes.
        * Number of vehicles that ran out of fuel.
    - A file is written with more detailed statistics:
        * For each junction, the number of crashes and the number of vehicles that have passed through.
        * For each road, the number of vehicles that traveled on it and the highest concurrent vehicle count (i.e., congestion levels).