UNIVERSITY OF CALIFORNIA,
IRVINE

Performance-robust, Non-blocking, Data-driven Barrier Synchronization for Multicore, Multithreaded Parallel Algorithms

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Arturo Garza Rodriguez

Dissertation Committee:
Professor Isaac D. Scherson, Chair
Professor Magda S. El Zarki
Professor Tony D. Givargis

2023

© 2023 Arturo Garza Rodriguez

DEDICATION

To mom and dad

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACKNOWLEDGMENTS

I would like to thank a lot of people, this work would not have been possible without you.

First and foremost, I thank my advisor Isaac D. Scherson. Thank you for your friendship. Thank you for taking me on as your student and for believing in me throughout this process. This dissertation is the result of your constant guidance and I will forever be grateful.

Thank you Prof. El Zarki and Prof. Givargis, for serving as members of my dissertation committee and helping improve this work. Thank you Prof. Shindler, for taking me as your Teaching Assistant. I enjoyed and learned a lot working with you.

Thanks to my family and friends. To my parents, Tirzo and Rosalinda, I cannot overstate how important your support has been to me. You have always been a lifeline during challenging times and there are no words to thank you enough. To my brothers, Alejandro and Alberto. You have always been a pillar and have enriched my life in countless ways. To Melina, thank you for motivating me to keep pushing in this journey. Your help and support has been unparalleled. To all my friends, thank you for the good times. I will always cherish every moment.

Also, I would like to acknowledge the sources of funding that made this dissertation possible. To the department of Computer Science for their support in the form of Teaching Assistantships and thanks to the Consejo Nacional De Ciencia y Tecnología (CONACYT) for their additional financial support during my first four years of doctoral studies.

Finally, the formatting of this dissertation is based on Lars Otten's LaTeX template [74], thanks to everyone that has contributed to it over the years.

VITA

Arturo Garza Rodriguez

EDUCATION

Doctor of Philosophy in Computer Science	**2023**
University of California, Irvine	*Irvine, California*
Master of Science in Computer Science	**2020**
University of California, Irvine	*Irvine, California*
Bachelor of Science in Telematics Engineering	**2015**
Instituto Politécnico Nacional	*Mexico City, Mexico*

RESEARCH EXPERIENCE

Graduate Research Assistant	**2018 – 2023**
University of California, Irvine	*Irvine, California*

TEACHING EXPERIENCE

Teaching Assistant	**2018 – 2023**
University of California, Irvine	*Irvine, California*

ENGINEERING EXPERIENCE

Software Engineering Intern	**Summer 2022**
Google	*Irvine, California*
Software Engineering Intern	**Summer 2021**
Google	*Irvine, California*
Software Engineering Intern	**Summer 2020**
Google	*Irvine, California*
Student Software Developer	**Summer 2019**
CERN	*Irvine, California*
Programmer Analyst	**2015 – 2018**
Oracle	*Guadalajara, Mexico*

REFEREED PUBLICATIONS

Non-blocking Technique for Parallel Algorithms with Global Barrier Synchronization **December 2021**
International Conference on Computational Science and Computational Intelligence

Recursive MaxSquare: Cache-friendly, Parallel, Scalable in situ Rectangular Matrix Transposition **December 2020**
International Conference on Computational Science and Computational Intelligence

SIMD-node Transformations for Non-blocking Data Structures **September 2020**
International Conference on Parallel Processing and Applied Mathematics

ABSTRACT OF THE DISSERTATION

Performance-robust, Non-blocking, Data-driven Barrier Synchronization for Multicore, Multithreaded Parallel Algorithms

By

Arturo Garza Rodriguez

Doctor of Philosophy in Computer Science

University of California, Irvine, 2023

Professor Isaac D. Scherson, Chair

In a general-purpose multicore multithreaded parallel environment, multiple threads work simultaneously to finish a task faster. Usually, threads need to communicate with each other due to the need to share data or synchronize access to shared data. Communications commonly occur when threads need to wait for the computational results of other threads before continuing their own computations; all necessary data dependencies need to be met before subsequent computations take place.

A barrier is a synchronization construct that enforces a collective pause and data sharing at a given execution point between all participating threads of a parallel computation. No thread proceeds beyond a barrier until all other threads have reached it. However, in a general-purpose system where many processes compete for the available computational cores, a scheduler decides which thread gets a core to execute its next line of code. The barrier can easily become a performance bottleneck due to its global blocking nature: one preempted thread blocks the progress of all other threads that are waiting at a barrier.

This dissertation introduces a novel technique that changes the global nature of the barrier into a distributed data-driven synchronization model with non-blocking thread progression guarantees. The idea is to exploit the algorithm-based memory access patterns to extract peer-to-peer interthread communication and remove the explicit use of a barrier synchronization construct. Our proposed technique is experimentally validated. The results are promising and show considerable robustness in performance as opposed to their barrier-based algorithm counterparts.

Chapter 1

Introduction

Prophets have voiced the contention that the organization of a single computer has reached its limits and that truly significant advances can be made only by interconnection of a multiplicity of computers.

Gene Amdahl

Multicore CPU architectures are a pillar in modern computing platforms. However, it is not straightforward to efficiently exploit the underlying parallelism that the hardware provides. This has brought considerable attention to this research field.

One of the most common forms of parallelism for these modern computer architectures is the Single Program Multiple Data (SPMD) model. The SPMD model splits a task into multiple identical execution threads that will run simultaneously on multiple CPU cores and, usually, work on disjoint subsets of data. Generally, these programs are written in sequential languages extended with communication and synchronization primitives, with barrier synchronization being one of the most common constructs for synchronizing parallel programs. Also, these parallel algorithms exhibit data dependencies at different points in time within their implementations; and threads may need to wait for the results produced by other threads. A barrier is a synchronization construct that forces an execution thread to wait until all participating threads reach the same barrier. The necessity for barrier synchronization arises from the algorithmic data dependencies requirements.

Communicating and synchronizing multiple execution threads present significant challenges. Modern CPUs are powerful out of order execution engines, there are features like branch prediction, instruction prefetching, store buffering, and others that accelerate the computations of sequential programs. However, these features may represent major drawbacks when multiple threads try to work collectively.

Additionally, these systems are not designed to execute just specific types of workloads, they are also formidable general purpose engines that execute large volumes of diverse tasks. The oversubscribed nature of multicore computing platforms impose additional difficulties since a CPU core is now a resource that needs to be shared between multiple threads that are just waiting to be executed. The overall performance of a parallel application may suffer significantly when multiple threads are waiting for the result of the computations of another thread that is yet to be assigned for execution on a CPU core.

This work presents a model that addresses the previously mentioned deficiency and provides performance robustness to parallel tasks with data dependencies that require barrier synchronization. By exploiting the memory access patterns, our data-driven model confers better thread progression guarantees against the oversubscribed blocking nature of modern multicore computing platforms. The rest of the thesis is organized as follows.

Chapter 2 presents the preliminary information, concepts, definitions, and terms that are necessary in order to introduce the rest of the material. It explores the notion of parallel computing and its taxonomy, a classification of the different types of thread progression guarantees that have been proposed, and a deeper insight into barrier synchronization.

Chapter 3 illustrates the challenges to overcome and the embodiment for the rest of this work. It presents the target architecture, its capabilities, its limitations, how barrier synchronization can be achieved, and the main complications faced by parallel multithreaded applications.

Chapter 4 gathers and summarizes a thorough study in all the related work that has been performed so far to address the challenges stated in chapter 3. The summary is organized in two major categories. One to address the overhead introduced by barrier synchronization constructs and how to mitigate it. The second one shows how to design parallel algorithms that are more tolerant to unexpected system delays.

Chapter 5 presents the main contribution of this dissertation: the *non-blocking data-driven barrier synchronization* (NBD^{2}BS) model. The general idea is to exploit the fixed memory access patterns that are exhibited in some parallel algorithms. This, in order to generate peer-to-peer communication and synchronization schemes, instead of using blocking barrier synchronization constructs.

Chapter 6 is the experimental verification of the proposed model. We devised novel non-blocking implementations of important algorithms that are core computational tasks in the fields of parallel sorting and signal processing that can be attractive to other research areas.

Chapter 7 motivates future work towards *wait-free* barrier synchronization. A multithreaded task-stealing model is proposed and the experimental results are promising to keep exploring more efficient methods.

Finally, chapter 8 presents the concluding remarks, emphasizing the impact of our contributions and motivating the future opportunities in this research field.

Chapter 2

Preliminaries

If I have seen further, it is by standing on the shoulders of giants.

Isaac Newton

The advent of multicore parallel computing has become a revolution, enabling new technology advances and delivering high performance to the most demanding applications. Over the years, this research field has been widely studied: from the design and implementation of novel parallel computer architectures, all the way to unprecedented approaches to solve massively large problems.

However, only one thing is certain, the field of parallel computing is in constant evolution with different goals in mind; including, but not limited to: improving performance, reducing costs, using multiple computational resources effectively, and ease of programming. This chapter presents the necessary concepts to introduce the rest of the dissertation and serves as a guideline of the main areas in which this thesis contributes to the field of computer science.

2.1 Multicore Parallel Computing

Parallel computing is the art of solving a problem by decomposing it into smaller subproblems and solving them simultaneously using multiple computational resources. It is a necessity in problems and tasks that would otherwise take a long time to complete. With claims in the past few years of the deceleration of Moore's law [50], the need for and popularity of multicore parallel computers has been steadily increasing.

Arguably, one of the most important turning points in parallel computing is the Gustafson's correction [36] to Amdhal's law [7], where the benefits of using multiple computational resources to solve large problems was shown. Michael Flynn introduced a taxonomy of computing systems [30] that has been highly adopted ever since, which includes Single Instruction Multiple Data (SIMD), Multiple Instructions Single Data (MISD), and Multiple Instructions Multiple Data (MIMD) to help classify parallel computing systems.

SIMD-type systems refer to the so-called data-parallelism or vector-parallelism, where the granularity of the computation is at the instruction level. The same instruction is executed simultaneously across multiple pieces of data. Whereas, MISD-type systems refer to multiple different instructions working on a single data stream. A simple example of a MISD computational model is an instruction pipeline where each stage of the pipeline executes, simultaneously, a different instruction over the same data pool. MIMD-type systems extend the MISD model, multiple instructions are being independently executed over different pieces of data and most modern general-purpose processors follow this model.

Throughout the years, MIMD-type computers have been in the vanguard of research efforts. Multiple interconnected processing units (cores) executing independent instruction streams and working cooperatively to solve a given problem. Consequently, recent advances focus on increasing the available parallelism and exploiting such parallelism is at the forefront of modern computing challenges [40, 58].

One fundamental characteristic of multicore systems is the utilization of a shared memory address space. Applications running on such systems exploit their parallelism by using more than one core at the same time. By sharing a common memory address space, multicore processors facilitate data sharing across all cores. It provides low latency and high bandwidth for communication, making them ideal for high performance computing.

Programming parallel algorithms in multicore systems is relatively straightforward. It is similar to writing sequential programs, but multiple identical copies of the same program run simultaneously on different cores and they usually work on disjoint sets of data. This parallel computational model was introduced as Single Program Multiple Data (SPMD).

2.2 Single Program Multiple Data (SPMD)

The Single Program Multiple Data (SPMD) parallel computational model was originally devised by Frederica Darema [20, 21, 22]. This model was first developed as an effort to pursue low-overhead, ease of programming, and high-cooperative parallelism. The premise to maximize cooperativeness is based on that all processing units execute the same stream of instructions; the same computer program. It is often referred to as an execution *thread*. Without loss of generality, a thread can be defined as a lightweight process and as the smallest unit of execution of the SPMD model.

SPMD-type parallelism is sometimes considered a generalized version of the SIMD-type or vector-type parallelism, since SPMD becomes SIMD at its minimum possible program size: one instruction. Another way to think about it is that if all SPMD programs are executed in lockstep, then it becomes SIMD. This is one of the most important characteristics of the SPMD model, there is no restriction about which instruction is executed at any point in time. Multiple threads can be executing different instructions of the same program at the same time and not necessarily moving at the same step.

The described behavior has been also widely studied and it is a research field in its own right. *Concurrency*, then, can be defined as a composition of multiple independent threads. This is, any thread can execute any given instruction at any given point in time.

Furthermore, the SPMD model is also considered more general than the SIMD model in the data spectrum, since each thread is not limited to act on different and disjoint pieces of data; creating contention and consistency problems when some pieces of data are shared across multiple execution threads. Hence, heavily impacting and degrading the performance and scalability of parallel applications.

The SPMD model has been widely adopted and several state-of-the-art parallel programming environments are based on SPMD: OpenMP [91], MPI [90], and multithreading techniques are just a few. SPMD proved that it is straightforward to map parallel applications into parallel machines, but imposes significant challenges as well; including, but not limited to: communication and synchronization among multiple execution threads.

2.3 Concurrent Computations and Synchronization

Synchronization refers to the use of mechanisms or techniques to impose or constraint some order in the operations performed by multiple concurrent threads. In other words, it is a consensus problem, where all involved participant threads need to agree in the order of their actions. In a system with multiple threads, getting all of them to observe the same order of events is a very difficult task [55].

There is an unpredictable nature about when an operation or event can occur or be observed among multiple independent execution threads. It is not hard to imagine a scenario where one action takes place in a particular thread, but it does not communicate it properly to the rest of them. It is possible, that from another thread's point of view, this action never takes place or it just happens after another series of events.

Definition 2.1. *A "happened before" time ordering relation between two events, "a" and "b", of a system is the smallest relation satisfying the following three conditions:*

1. *If "a" and "b" are events in the same thread, and "a" comes before "b", then "a" happens before "b".*
2. *If "a" is the sending of a message by one thread and "b" is the receipt of the same message by another thread, then "a" happens before "b".*
3. *If "a" happens before "b" and "b" happens before "c", then "a" happens before "c".*

Two distinct events "a" and "b" are said to be concurrent if no "happened before" relationship can be established between them.

Leslie Lamport formalized this notion of time in multithreaded systems [55]. The idea that a certain event can happen before another must be specified in terms of events that are observable within the system and not just in terms of physical theories about an event happening at an earlier time than other event. Lamport also devised one of the concepts that refer to the correctness properties of parallel and concurrent algorithms [56]. The *sequentially consistent* concept states that the result of any concurrent execution is the same as if the operations of all threads were executed in some sequential order and the operations of each thread appear in this sequence in the order specified by its program. Sequential consistency has been widely adopted and it is arguably one of the most common ways to corroborate the correctness of multithreaded algorithms.

Definition 2.2. *Sequential consistency states that the result of an execution is the same as a single interleaving of sequential, program-order memory accesses from different threads.*

In the context of shared memory multicore systems, one thread sending a message to another thread is implicit and transparent via memory access instructions (read and write). The sending of a message by one thread is performed via writing into memory and the receipt of the same message by another thread is done by reading from memory.

Definition 2.3. *Concurrent shared memory computing consists of multiple threads, each of which is a sequential program on its own right. These threads communicate by calling methods of objects that reside in a shared memory. Threads are asynchronous, meaning that they run at different speeds, and any thread can halt for an unpredictable duration at any time.*

Multicore systems are naturally concurrent, the operations performed by the many threads running in the different cores can interleave arbitrarily and halt unpredictably. Multiple synchronization mechanisms with different goals in mind have been proposed [44]. The next section presents a model to classify different synchronization mechanisms on how the behavior of one thread can have an effect on the execution of other threads.

2.4 A Thread Progression Model

Maurice Herlihy and Nir Shavit [39, 41, 43, 44] proposed and formalized a widely adopted taxonomy of multithreaded systems based on the progression guarantees that each of the threads exhibit. These thread progression guarantees are classified based on the ability of each thread to take steps towards its completion regardless of the progress of the rest of the threads, see table 2.1.

The concepts illustrated in table 2.1 will be explained by means of example. Without loss of generality, assume we are required to devise a mechanism to solve one of the oldest problems in synchronization: *mutual exclusion*. In order to protect a shared computational resource, often referred as *critical section* (CS), from multiple threads accessing it at the same time.

	Blocking	**Non-blocking**
Some threads make progress	Deadlock-freedom	Lock-freedom
Every thread makes progress	Starvation-freedom	Wait-freedom

Table 2.1: Taxonomy of thread progression guarantees.

Definition 2.4. *Mutual exclusion is when critical sections of different threads do not overlap. For threads A and B, and integers j and k, either CS_A^k happened before CS_B^j or CS_B^j happened before CS_A^k; where CS_A^k is the interval during which A executes the critical section for the k-th time.*

One way to solve the mutual exclusion problem is to let threads compete for a *lock*, which is a construct that a thread can acquire in one indivisible step (*atomic read-modify-write memory operation*). Then, the first thread to acquire the lock is the one who wins access to the shared resource. Otherwise, if a thread fails to acquire the lock, it keeps trying until success. This technique is considered to be *deadlock-free* because at least one thread makes progress towards the shared resource and eventually another thread might gain access to the shared resource when the lock becomes available again.

Definition 2.5. *Deadlock-freedom guarantees that if some thread attempts to acquire the lock, then some thread will succeed in acquiring the lock. If a thread calls acquire_lock() but never acquires the lock, then other threads must be completing an infinite number of critical sections.*

However, there is no guarantee that all involved threads will actually gain access to the shared resource; with just a little bit of bad luck, a thread might never win a contention match to acquire the lock. Hence, it is not considered *starvation-free.*

Definition 2.6. *Starvation-freedom guarantees that every thread that attempts to acquire the lock eventually succeeds. Every call to acquire_lock() eventually returns. This property is sometimes called lockout-freedom.*

A different way to approach mutual exclusion, with starvation-freedom progression guarantees, is to *enqueue* each thread in order to gain access to the shared resource. This fairness mechanism guarantees that all threads eventually will make progress towards the shared resource. Nevertheless, it is important to point out that a stronger progression guarantee does not necessarily imply better performance. The introduction of the queue might slowdown the overall performance, but it ensures that all threads will eventually acquire the lock.

An important aspect of deadlock-freedom and starvation-freedom is that both continue to *block* the computational progress of the rest of the threads while they wait for the lock to become available, this is known as *blocking synchronization*.

Definition 2.7. *Blocking synchronization comprises techniques used to prevent multiple threads from simultaneously accessing shared computational resources by allowing only a single thread to make progress.*

Synchronizing multiple threads using locks is relatively straightforward to reason about. However, threads get blocked under these circumstances and new ways to maximize the available parallelism among multiple threads have become a necessity. Herlihy and Shavit pioneered what is known as *non-blocking synchronization* [39, 41, 43, 44].

Definition 2.8. *Non-blocking synchronization comprises techniques used to prevent multiple threads from simultaneously accessing shared computational resources without having to block other threads progress.*

In non-blocking synchronization there are three different progression guarantees that generally apply to the SPMD parallel model. Each one with better progression guarantees than the previous one. It is said that an operation is *obstruction-free* if a thread makes progress when the rest of the threads do not take any steps in their execution. Then, a *lock-free* mechanism guarantees that at least one thread makes progress. Finally, an operation is *wait-free* if it guarantees that all threads make progress.

Definition 2.9. *A method is obstruction-free if, from any point after which it executes in isolation, it finishes in a finite number of steps.*

Definition 2.10. *A method is lock-free if it guarantees that infinitely often some method call finishes in a finite number of steps.*

Definition 2.11. *A method is wait-free if it guarantees that every call finishes its execution in a finite number of steps.*

These definitions are similar to their blocking counterparts, deadlock-freedom and starvation-freedom. However, non-blocking synchronization algorithms are designed to not block the progress of other threads while executing their own operations. Figure 2.1 presents the complete hierarchy of the different thread progression guarantees.

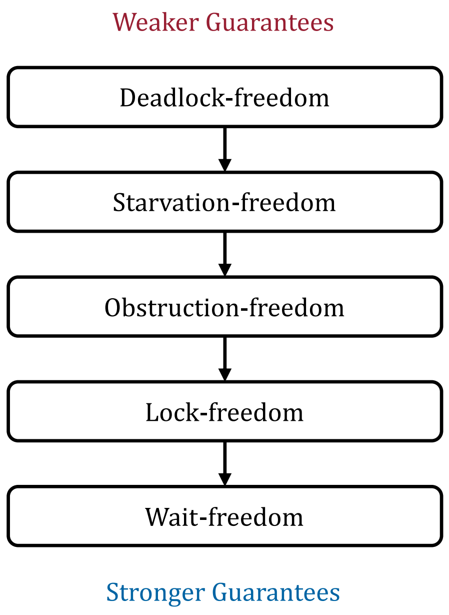

Figure 2.1: Hierarchy of thread progression guarantees.

To illustrate non-blocking synchronization, the mutual exclusion mechanism will be modified and it will allow multiple read-only threads in the shared resource. This is known as a *reader-writer lock*. If a thread only wants to read from the shared resource, it may proceed without waiting. On the other hand, if a thread wants to write into the shared resource, it waits until all read-only threads finish. This technique provides wait-free progression guarantees to all read-only threads. However, a writer thread must compete against all readers in order to access the shared resource and potentially impacting its performance. Ideally, the goal is to maximize progress among multiple threads without side effects; but in practice, this is not easy to accomplish.

This dissertation leverages the same model proposed by Herlihy and Shavit to characterize the different progression guarantees that our proposed technique exhibits. Our target synchronization construct is the so-called *barrier synchronization* and we show how it achieves non-blocking thread progression guarantees.

2.5 Barrier Synchronization

Arguably, the most common construct to coordinate multiple execution threads in the SPMD parallel model is the so-called *barrier synchronization*, ensuring that no thread continues execution beyond a barrier until all other threads reach the same barrier. See figure 2.2.

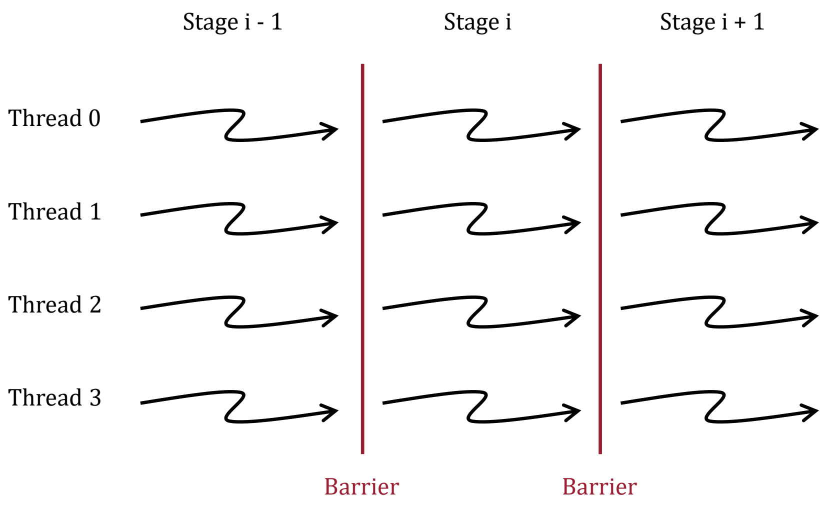

Figure 2.2: Barrier synchronization.

Barrier synchronization is used to protect the concurrent reads and writes that can happen across multiple iterations or stages of a parallel algorithm. This ensures the correctness of the algorithm by safeguarding the possible data dependencies that the algorithm exhibits. Figure 2.3 shows a general pseudocode snippet of the execution flow of a parallel algorithm where each participating thread needs to complete a series of iterations. The explicit barrier construct ensures that all threads complete their current task before proceeding into the next iteration. An important aspect to point out is that the finer the granularity of the computational task, the more barriers may be necessary.

```
// sequential code

// parallel section
for each thread:
  for i in 1 to N:
    Task(i)
    barrier

// sequential code
```

Figure 2.3: Barrier synchronization pseudocode usage example.

The barrier is the synchronization point where data realignment, repartitioning, and all communication between all participating execution threads takes place. Barrier synchronization points can easily become hot-spots and significantly degrade the performance of parallel applications. The overall performance is limited by the slowest of the threads involved in the computations and a single delayed thread blocks the progress of the rest of the threads.

2.6 Common Terms

- **Parallel computing:** Art of solving a problem by decomposing it into smaller subproblems and solving them simultaneously using multiple computational resources.
- **Single Program Multiple Data (SPMD):** Computational model where the same sequences of instructions are executed simultaneously over disjoint sets of data.
- **Thread:** Smallest unit of execution of the SPMD parallel model.

- **Atomic memory operation:** Instruction that allows a thread to read, modify, or write memory in one indivisible step.
- **Concurrency:** Multiple threads can execute multiple operations at any time.
- **Synchronization:** Art of enforcing some order in the operations of concurrent threads.
- **Blocking synchronization:** One thread may prevent others from making progress.
- **Non-blocking synchronization:** One thread may make progress regardless of others.
- **Deadlock-freedom:** At least one thread makes progress in a finite number of steps, but may prevent others from making progress.
- **Starvation-freedom:** Each thread makes progress in a finite number of steps, but may prevent others from making progress.
- **Obstruction-freedom:** Each thread makes progress in a finite number of steps when they are executed in isolation (the rest of threads do not take any steps).
- **Lock-freedom:** At least one thread makes progress in a finite number of steps.
- **Wait-freedom:** Each thread makes progress in a finite number of steps.
- **Barrier synchronization:** One thread waits for all others before resuming execution.

Chapter 3

Problem Statement

If you were plowing a field, which would you rather use:

two strong oxen or 1024 chickens?

Seymour Cray

Barrier synchronization is an algorithmic need in parallel algorithms that are composed of several computational stages. The barrier implementation is critical to deliver high performance. It should be fast, avoid unnecessary overhead, and minimize the time between the last thread reaching the barrier and the last thread leaving the barrier.

In this chapter, the limitations to overcome are presented within the context of the target computer architecture: the Central Processing Unit (CPU) and multicore CPU systems with a shared memory address space. Primarily, there are two major performance troublemakers: the own hardware abilities to guarantee correctness and consistency across all CPUs and the oversubscribed nature of these general-purpose systems where multiple threads compete with each other to execute their next line of code in a CPU core.

3.1 Thread-level Parallelism in Multicore CPUs

SPMD parallel algorithms are relatively straightforward to implement in modern multicore CPU architectures: one thread runs in one CPU core. There are several multithreaded programming techniques and libraries that allow programmers to execute the same program (thread) simultaneously across the different CPU cores. However, efficiently exploiting the full parallel capabilities of multicore systems is a hard task.

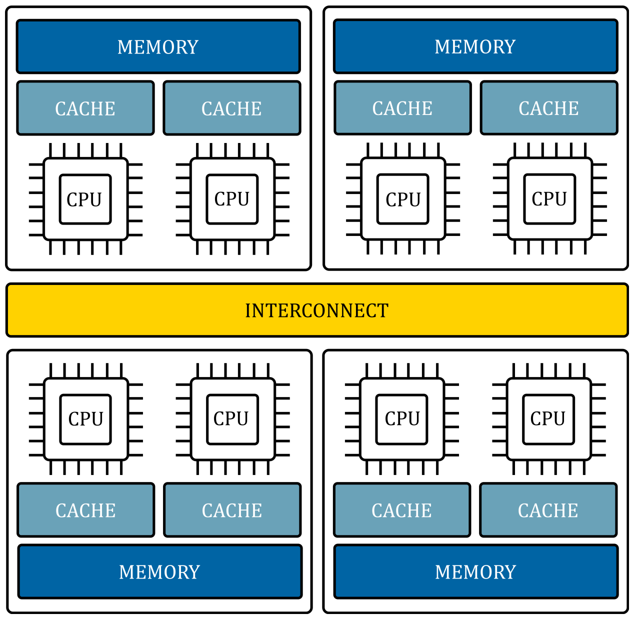

Figure 3.1: Multicore system architecture.

Multicore systems have many CPUs per physical chip, also known as cores, see figure 3.1. CPUs are powerful *out of order* execution engines: branch prediction, instruction prefetching, store buffering, speculation, compiler instruction scheduling, and other compiler optimizations are features to make the CPU as fast as possible and most of the time the program is executed in a different order than it was originally written. These features increase the performance of sequential programs, but they do more harm than good to parallel multithreaded SPMD applications where threads need to communicate: remember Lamport's work on correctness of multithreaded systems [55, 56]. The hardware needs to provide additional conditions to guarantee that a computer correctly executes multithreaded programs.

3.2 CPU Hardware Capabilities

When threads communicate with each other, they do it via *messages*. Intuitively, due to the out of order execution nature of multicore CPU systems, it is not difficult to imagine a scenario where receiving or sending a message from or to another thread actually happens in a different order than expected and, for that reason, compromising the correctness of the algorithm.

The multiple CPU cores communicate through a complex system of interconnected memories, namely, *caches*. In consequence, receiving and sending messages translates into reading and writing memory, respectively. Instructions that interact with memory are usually more expensive than others that perform simple calculations, even for a single thread.

Memory operations become more expensive when shared among multiple cores, since somehow it is necessary to enforce the correct order when reading and writing memory. All CPU cores maintain their own copy of the piece of shared data and it is stored in what is known as a *cache line* and it can store multiple elements depending on their size. Then, *memory consistency* refers to the property where each core must have an up-to-date copy of the shared data. A phenomena known as *race condition* happens when at least two threads perform memory operations to the same location, one of them is a write, and there is no notion that either of them happened before the other.

Definition 3.1. *A memory consistency model defines the ordering of externally observable events, i.e. reads and writes to memory. A read operation returns the value of the most recent write operation.*

Definition 3.2. *A race condition occurs when two or more threads access the same memory location, at least one is a write, and no happened before relation can be established between said operations.*

In order to overcome all these major drawbacks, the hardware provides different mechanisms to help maintain memory consistency for multithreaded parallel SPMD applications. The most important ones are *cache coherence* protocols, *atomic memory operations*, and *memory fences*. Each of these specifically designed to address a particular problem and each has its own associated performance costs.

3.2.1 Cache Coherence

The cache coherence problem arises because all CPU cores maintain their own copy of shared data in their local cache and at least one core modifies its copy. This action *invalidates* the rest of the copies and creates an incoherent situation. To avoid this, a cache coherence protocol defines a set of rules to maintain consistency across all local copies of shared data.

Each cache line is marked with a given status, most coherence protocols in modern multicore CPUs are based in the MESI protocol, which stands for Modified, Exclusive, Shared, and Invalid. One cache line can transition from one state to another depending on a given action. For example, in *bus interconnected* caches, if one core broadcasts on the bus a request of a cache line in exclusive mode, the other cores, which are constantly *snooping* the bus, will invalidate any copies of that particular cache line.

3.2.2 Atomic Memory Operations

An atomic memory operation allows a thread to read, modify, or write memory in one indivisible step. In order to ensure this behavior, the core writing to a piece of shared data must have exclusive access to the specific cache line. This consists in identifying and *locking* the cache lines that contain the shared data. Once the exclusive access is guaranteed, the core can then proceed with its atomic memory operation and relinquish the lock afterwards.

Another important aspect of working with atomic memory operations in shared data is that an update to an individual element of a cache line, *invalidates* the entire cache line. This is particularly harmful when another core wants to read a different piece of data that happens to be stored in the same cache line and needs to wait for the cache line to be valid again, referred to as a *cache miss*. This phenomena is known as *false sharing* and if the cache line gets invalidated frequently, for example in a loop, the performance cost increases.

3.2.3 Memory Fences

Memory fences are used to avoid reordering of memory operations. For example, a store buffer is a First-In-First-Out (FIFO) hardware construct that is used to accelerate the write to memory operations of a single thread. This buffer is smaller than the cache and faster to access. So, whenever a write instruction is issued, the piece of data is written in the store buffer and not directly in the cache. Eventually, the store buffer is flushed and the modified piece of data reaches the cache.

However, in contrast to the cache, the store buffer is also local to the CPU but does not have a coherence protocol with respect to the other store buffers. As a consequence, a write operation that is still in the store buffer has not yet occurred from the perspective of the rest of the threads. A thread that reads the same piece of data will observe a different order of operations than the one that is actually happening in the system. This is known as *bypassing* the store to load operations and it is common in Total Store Order (TSO) memory consistency models such as x86. To address this problem, a memory fence instruction needs to be issued to force the contents of the store buffer to be flushed into memory.

3.3 Barrier Synchronization in Multicore CPUs

Synchronization can be achieved via *busy-wait* techniques. One thread is constantly checking a piece of shared data to see if certain criteria is met. Only then, the thread may proceed with its computations.

To implement a barrier synchronization construct, it is necessary to consider how to properly signal all the threads that are involved in the parallel execution. First, a thread should announce when it reaches the barrier and, second, a thread should know when to leave the barrier. Algorithm 1 presents a pseudocode implementation of a *central counter* barrier.

Algorithm 1 Central counter barrier implementation.

```
procedure Wait(atomic &thread_count, atomic &global_flag)
    this_thread_flag = load(&global_flag)
    this_thread_position = fetch_and_sub(&thread_count, 1)

    if (this_thread_position == 1)                      ▷ Last thread reaches the barrier
        store(&thread_count, total_number_of_threads)
        store(&global_flag, ~this_thread_flag)
    else
        while (load(&global_flag) == this_thread_flag)          ▷ Wait for last thread
            yield_cpu()
```

The central counter barrier has two pieces of data that are shared between all participating threads. These two variables are marked as atomic: the thread count is used to signal that a thread has reached the barrier and the global flag is employed to notify threads to leave the barrier.

Whenever a thread reaches the barrier, it atomically decreases the thread count and, if it is not the last thread that reaches the barrier, it *waits* for the last thread to flip the global flag by constantly and atomically reading the flag value. Otherwise, if the thread that reaches the barrier is the last one, it then atomically resets the thread count and flips the global flag so the other threads can leave the barrier.

It is important to mention that when a thread waits for synchronization, it can basically engage in only one of two different behaviors (line 10 of algorithm 1): *active* waiting (consuming CPU cycles) and *passive* waiting (yielding the CPU core to another thread).

Definition 3.3. *A thread is actively waiting for synchronization if it is constantly consuming CPU cycles without doing any meaningful computation towards completion.*

Definition 3.4. *A thread is passively waiting for synchronization if it yields the CPU and then it is scheduled back at a later time. This, however, requires support from the kernel scheduler and it has no difference from active waiting if there are no threads waiting to be executed.*

Nevertheless, none of the two waiting mechanism help to unblock a thread faster while waiting since this totally depends on other threads delays. Yet, algorithms with passive waiting might outperform those with active waiting in scenarios where CPU cores are heavily contended by multiple threads, since there will be threads that have not yet reached the barrier, but they are waiting to be assigned to a CPU core by the kernel *scheduler*. This problem will be discussed in more detail in section 3.4.2.

Algorithm 1, even though it is easy to reason about, may present some inconveniences. One, already mentioned, there is another agent (the scheduler) that may impact parallel multithreaded applications and, second, the inherent overhead of the barrier itself. For example, the complexity of decreasing the thread count grows exponentially with the number of threads trying to access it because the threads need to compete against each other in order to gain exclusive access to the cache line where the variable is stored. This problem will be discussed in more detail in section 3.4.1.

3.4 Challenges in Barrier-synchronized Multithreaded Parallel Algorithms

Barriers are widely used and, in parallel algorithms that require multiple of them, it is of utmost importance to minimize their inherent overhead in order to maximize the application performance. The challenges to overcome are the *scalability* of the atomic operations within the barrier implementation and the *unnecessary waiting* that a thread has to endure due to their asynchronous nature, i.e. the execution threads will not reach the barrier at the same time. Being the latter the main problem addressed in this dissertation.

3.4.1 Scalability of Atomic Memory Operations

Atomic operations are expensive because the may involve flushing the store buffer, obtaining exclusive access to a cache line, and invalidating other copies of a shared piece of data.

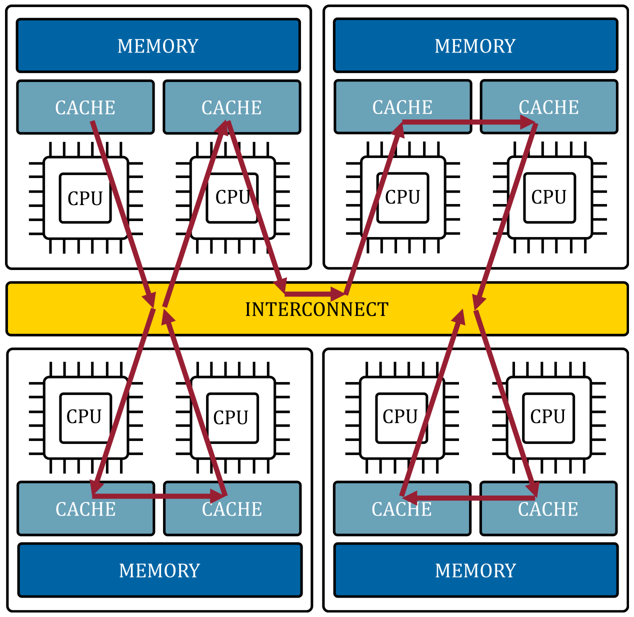

Figure 3.2: Cache line bouncing.

Imagine a scenario where we want to implement a concurrent counter. This counter is shared among multiple execution threads and each thread increments the count by one. This is not trivial to do so. In order for a thread to correctly increase the count, it needs to consider all the hardware limitations to make sure that each increment is *consistent*.

Each thread running in a CPU core will update its local copy of the counter and will invalidate the rest of the copies. The most up-to-date copy of the counter will be moving from cache to cache so that each thread correctly increments its own copy of the value, figure 3.2. This phenomena is known as *cache line bouncing* and results in poor performance, figure 3.3.

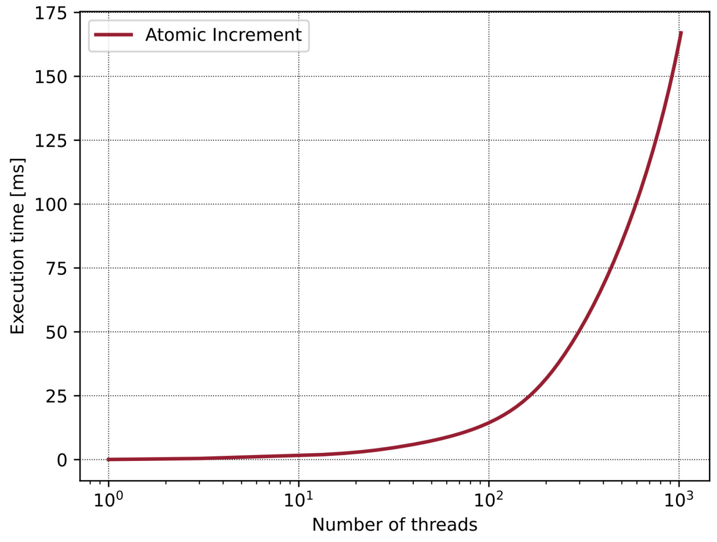

Figure 3.3: Atomic increment scalabilty in x86.

When a CPU reads an invalid cache line, it waits for the most up-to-date value from another CPU cache and only then proceed with its computations. This is known as a *cache miss* and they are one of the major obstacles in CPU performance.

3.4.2 Blocking Due to Unexpected System Delays

Unexpected delays are common due to the asynchronous nature of multicore systems, hence breaking the ideal computational model where all threads reach the barrier at the same time, figure 3.4. In modern multicore systems, some of the common culprits are, but not limited to: cache misses, page faults, clock rates, network, I/O, and interruptions.

However, a CPU is also a shared resource. An execution thread is assigned to a CPU core by the operating system kernel *scheduler*, but this thread can also be *preempted* and give up the CPU to another thread. This phenomena is known as *context-switching* and it can severely degrade the performance of barrier-synchronized multithreaded parallel SPMD programs. It is not hard to imagine a scenario where a preempted thread will block the progress of all the threads that are already waiting at the barrier, see figure 3.5. Hence, heavily impacting the overall execution time of a parallel application.

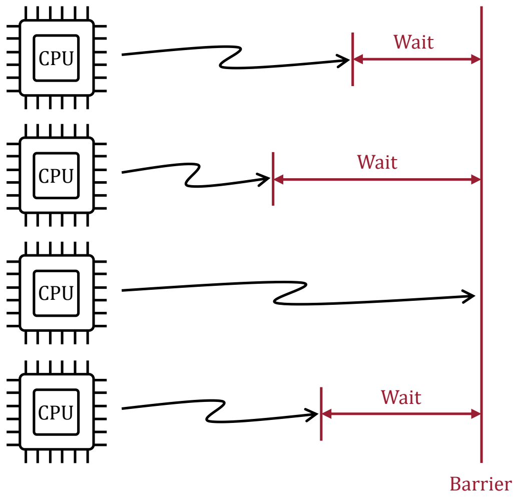

Figure 3.4: Waiting for slowest thread.

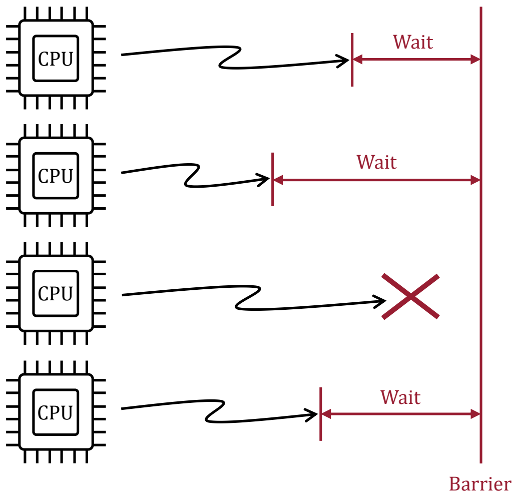

Figure 3.5: Blocking the overall task.

This is the main problem that this dissertation addresses. Thread preemption is very likely to happen due to the oversubscribed nature of modern multicore systems. The number of threads from different programs that are being executed is much greater than the number of physical CPU cores.

When developing parallel solutions, it is natural to try to use all available physical CPU cores in order to take full advantage of the parallelism offered by the hardware. These programs run in what is called *user space* and all hardware resources are managed by the operating system *kernel*, see figure 3.6. These resources include, but are not limited to: scheduling threads to CPU cores, memory management, I/O, interruption handling, etc. It is noteworthy to clarify that the software stack levels are independent of each other and one user space program can only interact with whatever kernel space provides. For example, it is not possible to avoid thread preemption from user space.

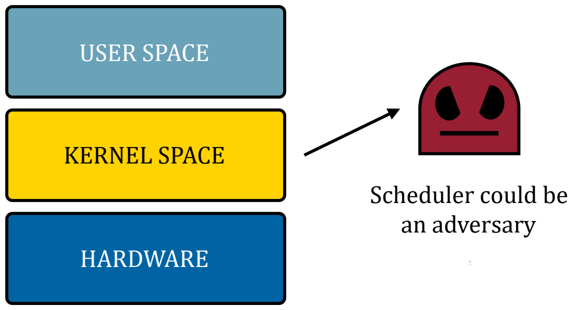

Figure 3.6: Kernel scheduler could be an adversary.

As a consequence, the kernel scheduler could be consider as a non-deterministic adversary that can block the execution of all threads of a parallel application and, overall, degrade its performance. Figures 3.7 and 3.8 respectively show what an ideal computational model looks like and what actually occurs in reality where a thread is preempted and tasks partitioned at any time, so the CPU can be shared with other programs.

The only assumption we make about the underlying scheduler is that eventually all threads will be scheduled for execution. This dissertation contributes with a novel *scheduler-oblivious* and *preempt-robust* technique to provide performance-robustness and non-blocking thread progression guarantees to barrier-synchronized multithreaded parallel SPMD applications.

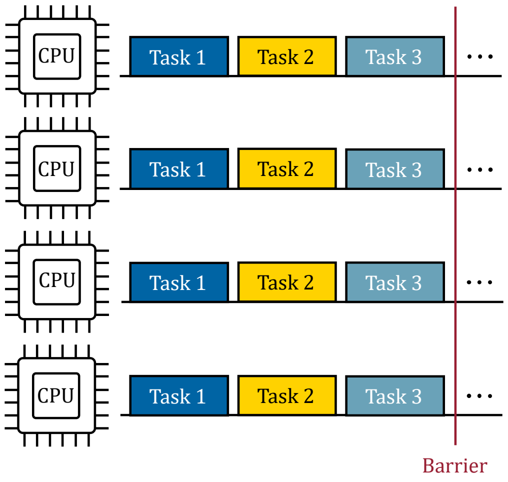

Figure 3.7: Ideal barrier-synchronized parallel computational model.

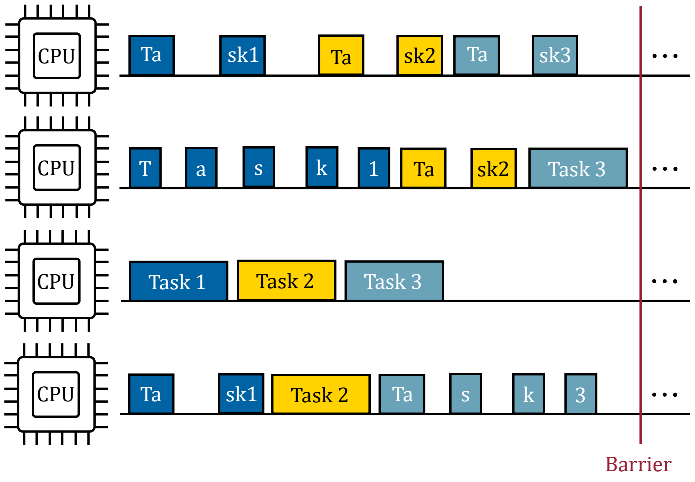

Figure 3.8: Real barrier-synchronized parallel computational model.

Chapter 4

Related Work

I know of no way of judging the future but by the past.

Patrick Henry

Past literature has given great attention to barrier synchronization primitives and their implementations. It has been shown how barriers can easily become a hot-spot in parallel applications due to the challenges presented in section 3.4. Research in barrier synchronization has targeted different ways to avoid the performance degradation caused by atomic operations and unexpected system delays.

This chapter includes a summary of the related work in the field. In section 4.1, a variety of different barrier synchronization implementations are presented: counter barriers, tree-based barriers, dissemination barriers, and tournament barriers. In section 4.2, a variety of different mechanisms to avoid the unnecessary waiting problem are presented: use of multiple disjoint barriers, fuzzy barriers, task-stealing barriers, and speculative barrier synchronization. Other proposals are based on these techniques or they solve the barrier synchronization problem by leveraging additional, but sometimes unpractical, hardware support [1, 2, 25, 31, 45, 78].

4.1 Scaling Barrier Synchronization

Barriers are commonly used across stages of data parallel algorithms in order to protect data dependencies between those stages. One research goal is to reduce the associated cost of the atomic memory operations that take place to ensure algorithmic correctness and to notify other threads when they reach the barrier.

The barrier implementations that rely on busy-waiting (spinning) in a global flag produce high amounts of memory and interconnection network contention that significantly impact the performance of parallel applications. Thus, research on the field aims to improve the locality of reference of atomic operations in pursuance of decreasing the inherent overhead of the barrier synchronization construct.

4.1.1 Counter Barriers

Counter-based barriers have been widely studied [38, 44, 61, 66, 67, 73, 76, 87, 99]. They are conceptually simple and straightforward to think about. However, they can suffer from high contention problems in central shared counters.

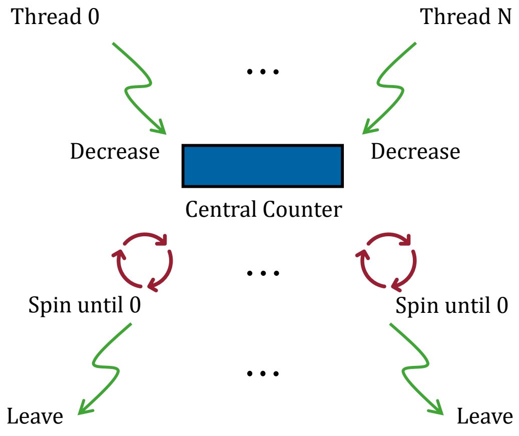

Figure 4.1: Central counter barrier.

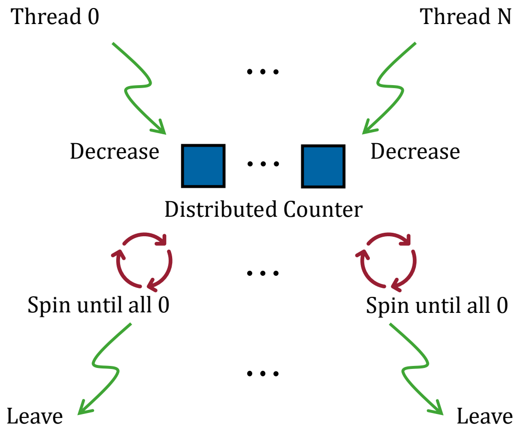

Figure 4.2: Distributed counter barrier.

In a central counter barrier, figure 4.1, the value of the counter is initialized to the number of execution threads that are participating in the parallel computations. When a thread reaches the barrier, it decreases the value of this counter and then waits (spins) until it becomes zero. This means that the last thread from the execution group has reached the barrier.

On the other hand, distributing the shared counter across local counters in each participating thread, figure 4.2, helps to avoid contention in a central counter. The downside of this implementation is that threads waiting at the barrier need to verify that all local counters of all participating threads have reached zero so they can proceed.

4.1.2 Tree-based Barriers

Tree-based barriers have been widely studied [8, 38, 44, 47, 66, 67, 69, 87, 98, 99]. The general idea behind a tree-based barrier is to distribute the signaling between threads in a tree-like fashion, where threads are assigned to nodes in a tree with notifications going up and down whenever a thread reaches the barrier. This, in order to alleviate the contention that is commonly presented in their central barrier counterparts. Some important contributions are the static-tree barrier [44], the combining-tree barrier [98], and the MCS (Mellor-Crummey and Scott) barrier [66].

In the combining-tree barrier, the execution threads are organized in groups of k and each group is assigned to a leaf of a *k-ary* tree. When a thread reaches the barrier, it performs an increment operation in a local counter of its own group. The last thread of the group continues up the tree by updating the count of the node's parent. This eventually propagates the updates to the root of the tree when the last thread of all groups reaches the barrier. Finally, a series of notifications are propagated down the tree so the waiting threads wake up and continue execution.

In the MCS barrier, to synchronize k threads, the barrier employs a pair of *k-node* trees, an arrival, and a wake-up tree. The general idea is that when a thread reaches the barrier, it sets a flag in its parent node in the arrival tree and gets notified by its parent in the wake-up tree when all threads have reached the barrier. This way, any given thread (at a node) does not modify nor test any other nodes rather than its parent or its children, leveraging the importance of locality of reference.

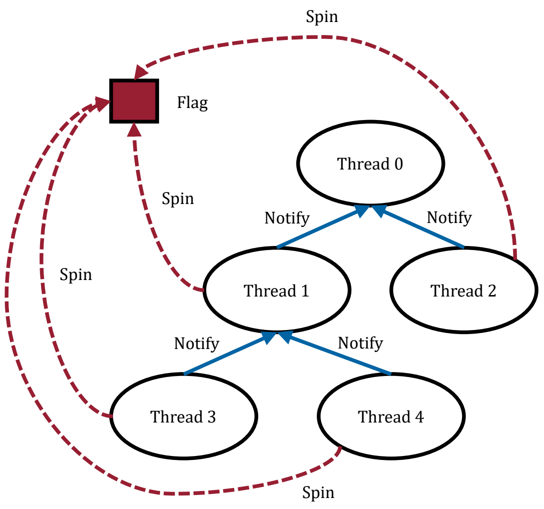

Figure 4.3: Wait at the static-tree barrier.

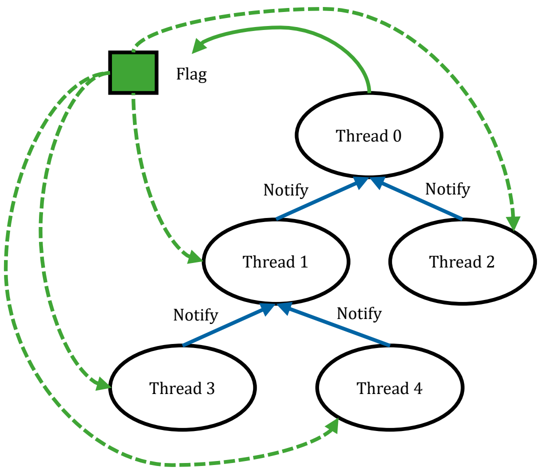

Figure 4.4: Leave the static-tree barrier.

Finally, the static-tree barrier is a great pedagogical example, and it will be used to show the importance of tree-like barriers in a more detailed way. Figures 4.3 and 4.4 help exemplify. Same as the MCS barrier, each thread is statically assigned to a node in the tree. This is, k nodes to synchronize k execution threads and each node has an associated count to it, which is the number of children that have not yet reached the barrier. When a thread reaches the barrier: if its local counter is 0, it means that all its children have reached the barrier, so it notifies its parent node by decreasing the local counter of the parent and it waits by spinning on a global flag (figure 4.3). Otherwise, the thread waits locally until its local counter becomes 0. Eventually, the root counter will become 0 and this thread is in charge to flip the global flag so the rest of the threads may proceed (figure 4.4).

4.1.3 Dissemination Barriers

Dissemination barriers are possibly the most widely used barriers in the field thanks to their performance and scalability [13, 31, 38, 66, 67, 70, 78]. First devised by Brooks [13] with a *butterfly barrier* proposal and then Hensgen et al. introduced the *dissemination barrier* [38], which enhances the original algorithm with a more efficient synchronization signaling pattern. An example can be found in figure 4.5.

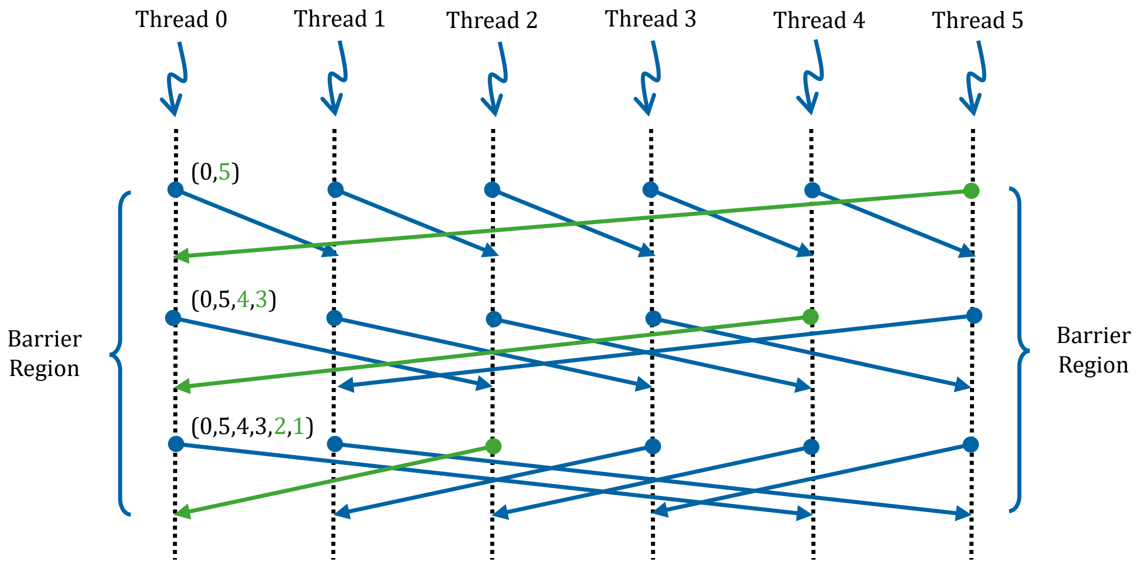

Figure 4.5: Dissemination barrier.

The main idea is that one thread will signal only another thread in a specific pattern. In the illustrated example, in round j, thread i signals thread $((i+2^j) \mod k)$ and waits for thread $((i-2^j) \mod k)$ in order to synchronize k threads. As a result of this, a given thread only needs to wait for another specific thread to move forward within the barrier region requiring only $\lceil \log_2 k \rceil$ synchronization rounds. Figure 4.5 shows how *thread 0* is signaled by a specific thread in each round.

The dissemination barriers present characteristics that make them good candidates for shared memory computer architectures where network interconnect accesses can be performed in parallel, due to the nature of its series of symmetric and pairwise synchronization operations.

4.1.4 Tournament Barriers

Tournament barriers are a form of dissemination barriers where *winner* threads continue to the next round and *loser* threads wait at a global flag. The idea behind these barriers is that only winner threads wait for the signaling of other winner threads.

These type of barriers might be more suitable for different scenarios than their dissemination barrier counterparts, for example, in architectures with a central memory bus where contention in the interconnection network can happen [67]. Figure 4.6 shows an example of a tournament barrier.

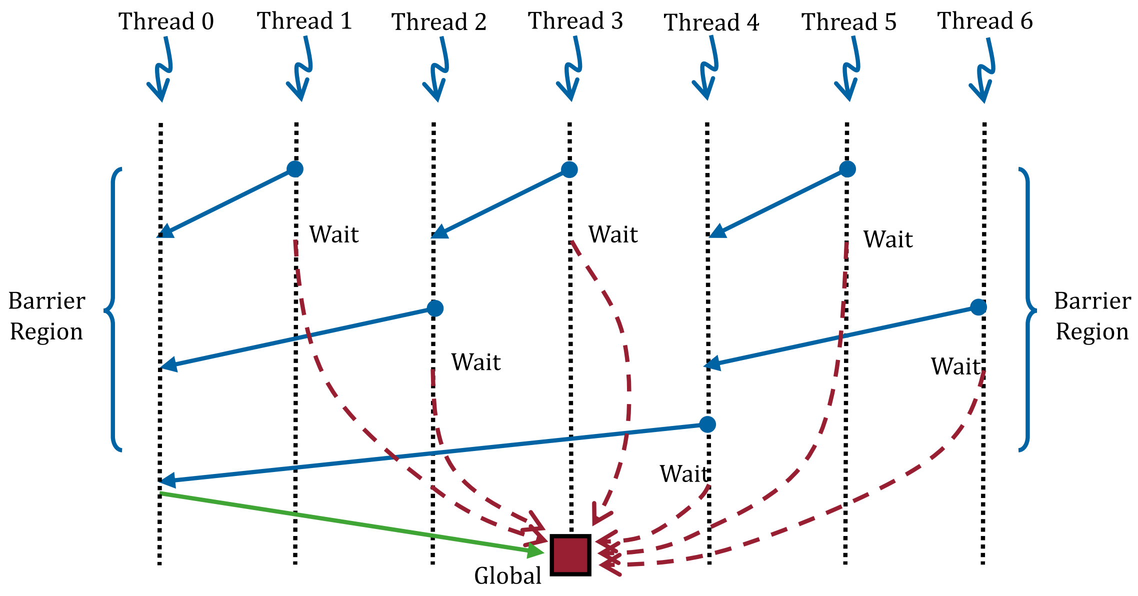

Figure 4.6: Tournament barrier.

Lubachevsky [61] and Hensgen et al. [38] first devised the tournament barriers. At each round within the barrier region, a winner thread is statically determined and continues to the next round. Whereas the loser threads spins at the global flag, waiting for *thread 0* to flip it at the end of the synchronization rounds. A tournament barrier, as in the dissemination barriers, requires a total of $\lceil \log_2 k \rceil$ rounds to synchronize k threads.

4.2 Unblocking Barrier Synchronization

Barrier-synchronized parallel algorithms suffer from the global blocking nature of the barrier itself. Section 4.1 shows different ways to implement different barrier synchronization constructs, aiming to reduce their intrinsic overhead.

However, research in non-blocking barrier synchronization and this dissertation have a different goal: allow the necessary system-wide progress of all execution threads without compromising the algorithmic correctness that is enforced by data dependencies.

4.2.1 Multiple Disjoint Barriers

If different execution threads do not exhibit data dependencies that will force them to synchronize every time a barrier is used. Disjoint threads can synchronize only among themselves by placing independent barriers for only those threads involved, see figure 4.7.

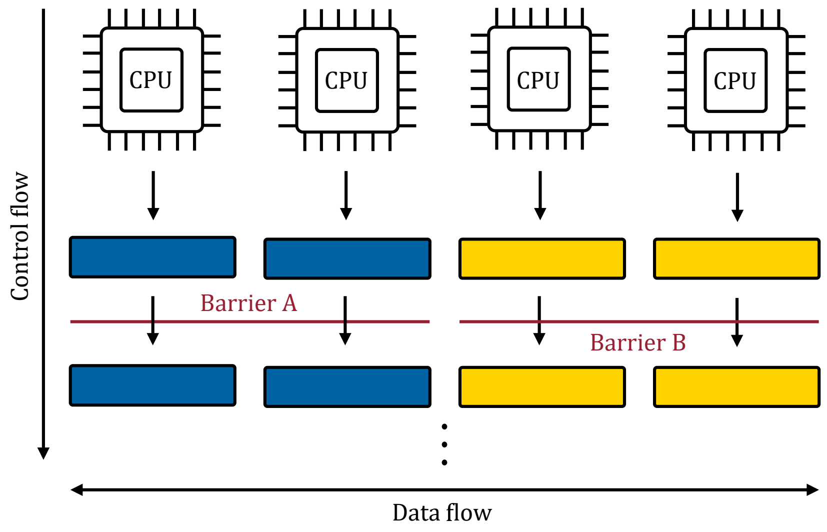

Figure 4.7: Progress with multiple disjoint barriers.

There are numerous ways to achieve this. The key idea is that each thread must anticipate which other threads are involved at each barrier synchronization.

Logically, separate barriers are assigned to distinct subsets of the input data. As a consequence, a thread needs to know which barrier to reach at each synchronization point. This requires overcomplicated control flows.

Furthermore, it has been shown that a system with *N* threads, with at most *N* input data streams, may need at most *N - 1* barriers [35]. Hence, although the problem is solved and unnecessary barrier synchronization constructs are eliminated, the added complexity in the program's control flow makes this technique exceptionally difficult to maintain and scale to a general model.

4.2.2 Fuzzy Barriers

The Fuzzy Barrier was originally devised by Rajiv Gupta in 1989 [35]. The main idea is to extend the barrier concept to include a sequence of statements that a thread can execute while waiting for the others to reach the barrier and necessary data communication, alignment, or partitioning can happen at any time inside the *barrier region.* Ideally, the larger the barrier region, the more likely it is that none of the threads will have to stall.

```
repeat N times:
  S0
  if:
    S1
  else:
    S2
  barrier
```

Figure 4.8: Branches outside barrier.

```
repeat N times:
  S0
  barrier:
   if:
     S1
   else:
     S2
```

Figure 4.9: Branches inside barrier.

For example, figures 4.8 and 4.10 show a simple case where a thread executes multiple statements with branches and waits for the rest of the threads at the end of each iteration. Without loss of generality, assume that statement *S1* takes longer to execute than statement *S2*. Then, all threads taking the *S2* branch will reach the barrier first and stall.

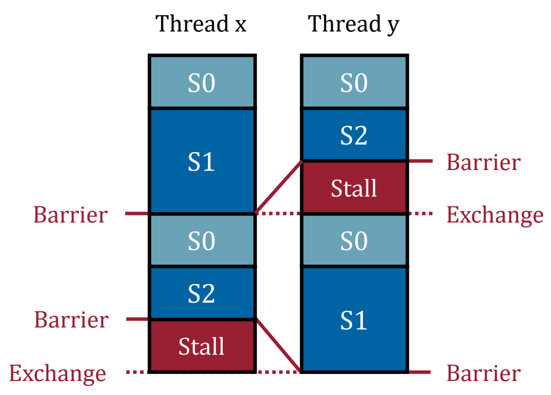

Figure 4.10: Progress with normal barrier.

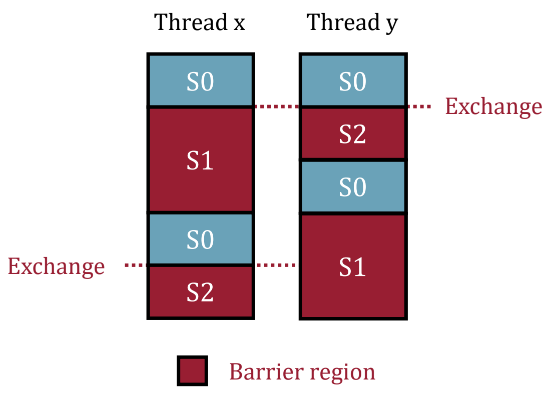

Figure 4.11: Progress with fuzzy barrier.

If the branches are included as part of the barrier region, even if multiple threads take different execution paths, they may not have to stall; see figures 4.9 and 4.11. It is important to point out that both *S1* and *S2* can be independent of each other, which means that a thread can start its next iteration without waiting for other threads. Note that the data exchange/communication is triggered when the last thread reaches the beginning of its barrier region and all other threads are inside their respective barrier region.

This example shows how the fuzzy barrier provides better protection against the performance degradation that is caused by the unexpected delays of other threads. Data communication can take place at any point in a wider range rather than forcing all threads to do so at a specific point.

The concept of the fuzzy barrier pioneered the idea of exploiting the communication and computation overlap that some parallel algorithms exhibit. Motivating the pursuit of more efficient implementations of parallel algorithms [48, 79, 83].

4.2.3 Task-stealing Barriers

Task-stealing barriers form part of a broader spectrum of techniques known as *load balancing* [24, 29, 44, 51, 54, 57, 65, 97]. As seen so far, barrier synchronization is used when the computations of a program are divided into several stages and a given thread waits for the necessary data dependencies in order to proceed to its next stage.

From the perspective of a task-stealing barrier, a thread waiting at a barrier could mean unbalanced computations between the participating threads. A task-stealing barrier aims to mitigate this, i.e. balance the computational workload.

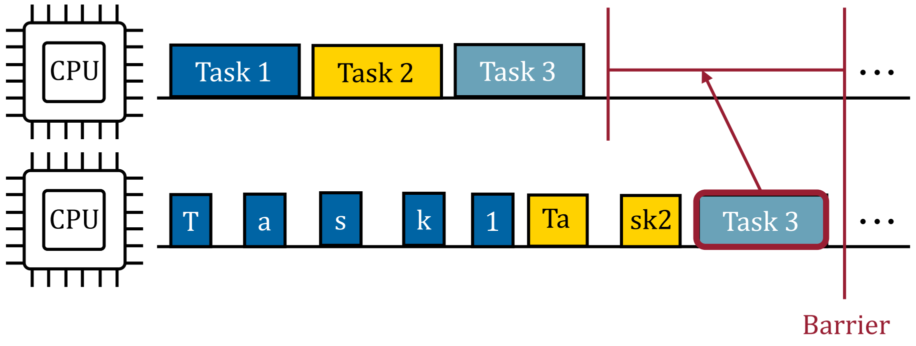

Figure 4.12: Task-stealing barrier.

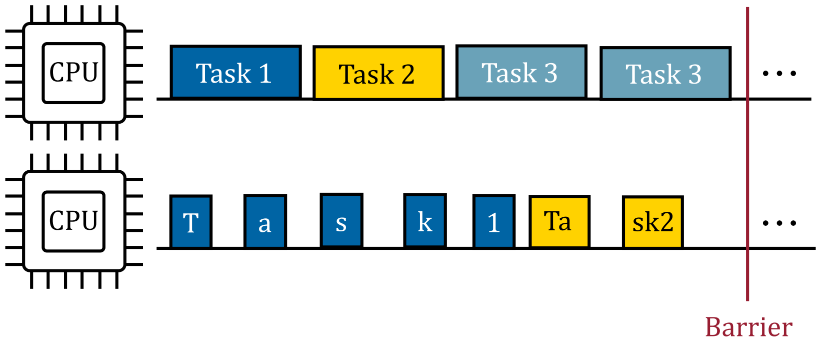

Figure 4.13: Task-stealing barrier steals task.

Figures 4.12 and 4.13 illustrate the task-stealing behavior: a stalled thread waiting at a barrier, can steal a task from another thread yet to reach the barrier. Inherently, this technique introduces overhead. Each thread maintains a queue of tasks, where multiple threads produce multiple tasks and multiple threads might consume multiple tasks. Thus, making the communication among threads more expensive than other barrier synchronization mechanisms and usually involves the use of complex underlying runtime environments.

4.2.4 Speculative Barriers

Speculative synchronization has been broadly studied [18, 27, 34, 37, 44, 53, 63, 64, 75, 77, 84]. In general, speculative synchronization tries to exploit the idea that some synchronization constructs are placed suboptimally and that a thread can make progress without conflicts.

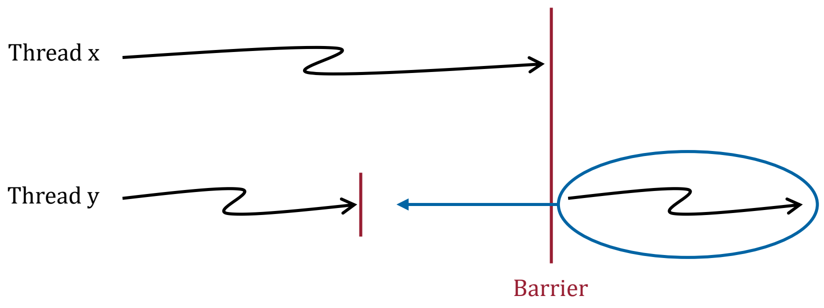

Figure 4.14: Speculative thread is rolled back.

In the barrier synchronization context, a thread can execute after the barrier and proceed if no conflict is detected. However, if a conflict is detected, the thread is rolled back. Figure 4.14 illustrates this behavior, where a conflict is detected in one of the threads and it is rolled back to recompute before the barrier.

One of the most common examples of thread speculation is transactional memory [42, 81]. A transaction is always performed in a *speculative* manner. As a transaction executes, it makes *tentative* changes to pieces of data. If the transaction completes without synchronization conflicts, then it commits. Otherwise, it aborts.

Modern hardware can nearly emulate transactional memory. Each CPU has its local cache and synchronization conflicts are handled via a cache coherence protocol. Each CPU can take advantage of their respective store buffers for tentative changes before they reach memory and force a flush to emulate a transaction commit operation. Therefore, small changes to hardware have been proposed in order to achieve hardware transactional memory [44]. Without hardware transactional support, software transactional memory techniques (using atomic memory instructions) are employed to detect synchronization conflicts.

4.3 Related Work Summary

This section presents a summary of the research performed in the barrier synchronization spectrum. There has been multiple barrier synchronization techniques that have been proposed and that address either the *scalability* problem or the *unnecessary waiting* problem. Table 4.1 shows the advantages and disadvantages of each proposal.

	Pros	**Cons**
Counter barriers	Practical	Blocking
Tree-based barriers	Low contention	Blocking
Dissemination barriers	Low contention	Blocking
Tournament barriers	Low contention	Blocking
Multiple disjoint barriers	Non-blocking	Complex control flow
Fuzzy barriers	Non-blocking	Application specific
Task-stealing barriers	Non-blocking	Complex runtime support
Speculative barriers	Non-blocking	Tradeoff: accuracy vs speed

Table 4.1: Summary of related work in barrier synchronization.

There are two main groups: the blocking barriers and the non-blocking barriers. The novel contribution of this dissertation is in the latter group in order to address the unnecessary waiting problem that threads encounter while waiting at the barrier.

In this work, we propose a non-blocking barrier mechanism driven by data that does not require creating complex control flows nor expensive runtime support. Parallel algorithms that exhibit certain criteria can benefit from the model presented in the next chapter.

Chapter 5

A Non-blocking Data-driven Barrier Synchronization Model (NBD^{2}BS)

The barrier is a rather strong means for synchronizing, and it may be more severe than is actually necessary.

Harold Stone

Previously, multiple ways were shown to reduce the associated overhead of a barrier synchronization construct and how to achieve non-blocking thread synchronization. However, most of the proposals address the problem from the perspective of the program's execution, i.e. the *control flow*. Thus, completely excluding another important dimension: the *data flow*. See figure 5.1.

The general rule of thumb to parallelize an algorithm is to *partition* the input data into multiple disjoint subsets. By means of multithreaded programming, each partition is assigned to a thread to be executed on a CPU core. Finally, perform all necessary synchronization and proceed to the next algorithmic stage.

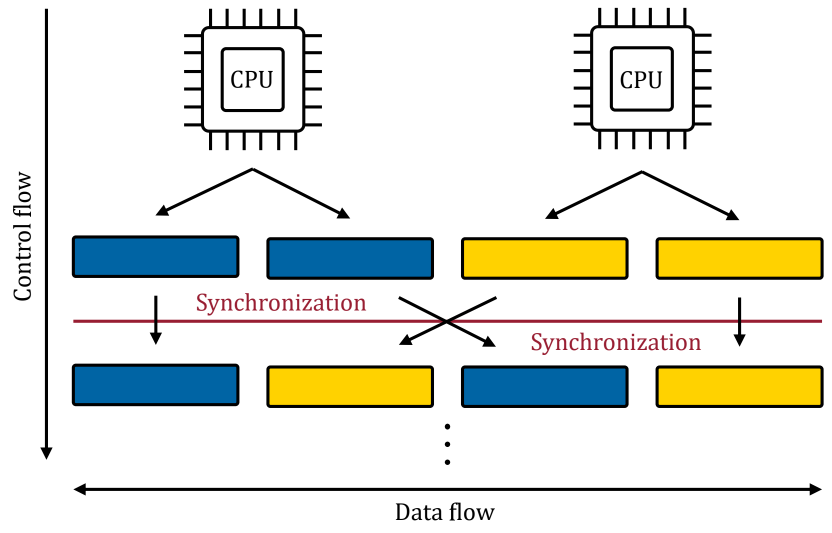

Figure 5.1: Control flow vs data flow in parallel applications with barrier synchronization.

Since there is no guarantee that every thread is executed at the same pace, there is no way to avoid the barrier synchronization point. However, it is possible to take advantage of the data flow dimension if given by a regular access pattern. This is, having knowledge of all *data-code* convergence points. Additionally, we are adding an extra dimension into our model: the parallelism. We call it the *thread-data-code three-dimensional convergence*. In other words, the access pattern is regular enough that it is possible to know which thread at which algorithmic stage needs which data partition.

Definition 5.1. *A parallel algorithm is three-convergent if the data access pattern is regular enough to know which thread at which algorithmic stage needs which data partition in order to proceed with its computations.*

Barrier synchronization enforces strong synchronization semantics to ensure that a thread reads the correct value written by another thread in subsequent stages, figure 5.2. Our model employs a more relaxed semantics model with finer granularity that achieves the same behavior: post-wait semantics.

5.1 Post-wait Synchronization Semantics

Post-wait synchronization semantics is generally used as a technique for efficient placement of synchronization constructs. It has been widely studied [4, 14, 23, 52, 71, 86], but most of these studies are focused on correctness analysis of parallel programs, algorithm-specific optimizations, and compiler techniques that leverage only the information available at compile-time. The idea behind post-wait synchronization is to be able to identify the data dependencies that are exhibited across the different stages of the algorithm. Each piece of data has associated a *flag* construct that is used to represent whenever that particular data has been *produced* by one thread and it is ready to be *consumed* by another thread, see figure 5.3.

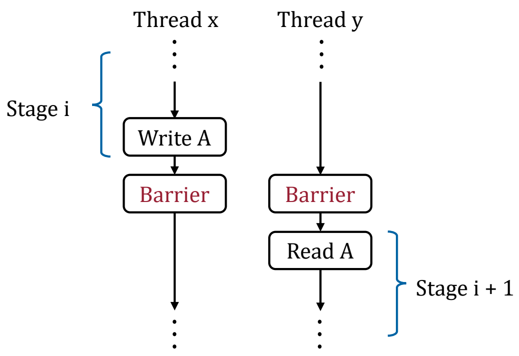

Figure 5.2: Barrier semantics.

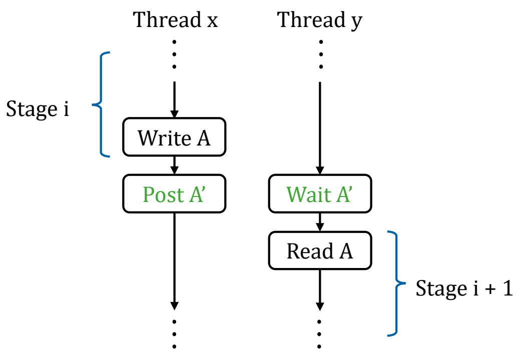

Figure 5.3: Post-wait semantics.

Figures 5.4 and 5.5 show a more concrete example of how *three-convergent algorithms* can benefit from explicit *post-wait* synchronization. In figure 5.4, *thread w* and *thread y* write in disjoint pieces of data. Whereas, *thread x* and *thread z* read their respective pieces of data in the following stage. The barrier synchronization construct enforces consistency in the values that are going to be read in the subsequent stage. However, this solves the problem from the control-flow perspective, all threads must finish their computations before proceeding to the next stage. This is a perfectly correct solution, but *thread w* and *thread x* do not present any data dependencies with *thread y* and *thread z* across these two stages; and there is no reason to stop the execution of either pair of threads if they are ready to begin their next stage.

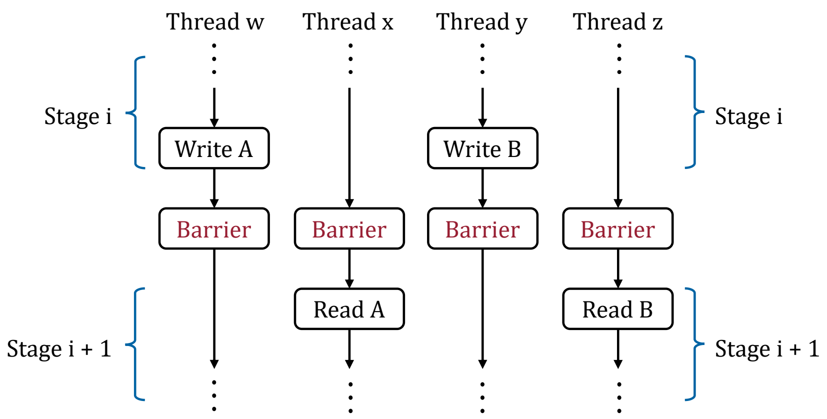

Figure 5.4: Progress using barrier synchronization semantics.

Different ways to address this problem have been discussed, see related work in section 4.2, but none of them fully exploit the data flow dimension. Our contribution is based in identifying and leveraging the fixed data patterns of parallel algorithms and map them into post-wait synchronization semantics.

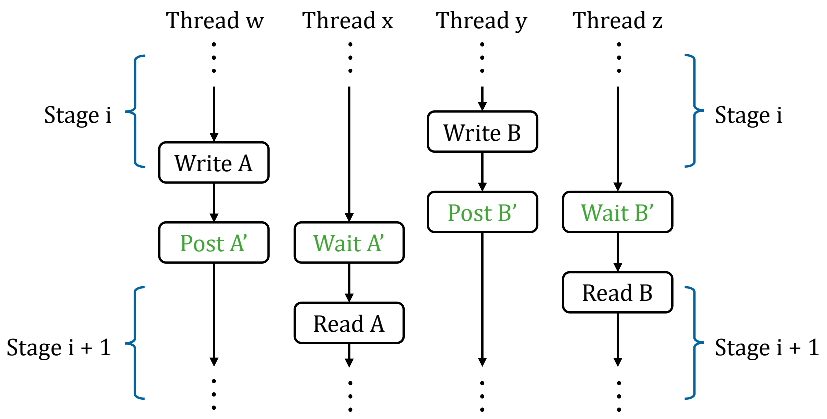

Figure 5.5: Progress using post-wait synchronization semantics.

Figure 5.5 presents the same scenario. Due to the two disjoint pieces of data, two different post-wait semantic constructs are necessary in order to know which piece is ready for the next stage. Hence, implying $\Theta(n)$ additional space for all post-wait semantic constructs, where n is the number of disjoint partitions of data. With this in mind, *thread x* and *thread z* can proceed to the subsequent stage independently of each other. *Thread x* only depends on *thread w* to post the necessary synchronization construct in order to proceed.

5.2 Non-blocking Data-driven Barrier Synchronization

Barrier synchronization is now driven by data, see figure 5.6. By exploiting the algorithm's memory access pattern, it is possible to identify data dependencies across the parallel computational stages of the algorithm.

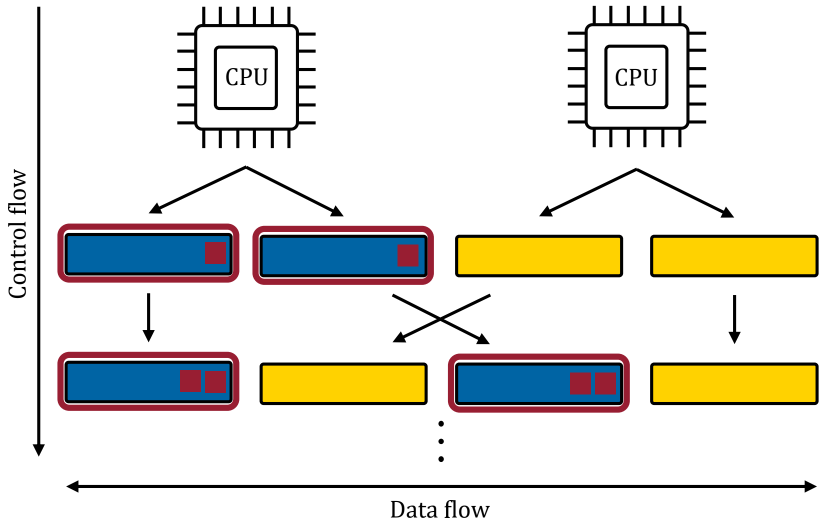

Figure 5.6: Control flow vs data flow in parallel applications with post-wait synchronization.

This enables peer-to-peer synchronization and removes the global blocking nature of the barrier synchronization construct. One thread can block only another thread at a given computational stage. Hence, providing non-blocking thread progression guarantees and allowing system-wide progress.

```
for each thread:
  for each stage:
    Task()
    barrier
```

Figure 5.7: Control flow barrier.

```
for each thread:
  for each stage:
    wait for my data
    Task()
```

Figure 5.8: Data flow barrier.

Figures 5.7 and 5.8 show the differences in pseudocode of parallel regions of a barrier-synchronized algorithm and an algorithm using our proposed model. Individual partitions of data have now their associated *flags* for post-wait synchronization and they indicate the particular algorithmic stage in which said partitions of data are currently at.

These flags can be implemented as atomic counters that represent the current algorithmic stage of the corresponding segment. Each *post* synchronization operation increases the count of its associated data segment. Each *wait* synchronization operation is translated to a *busy-wait* (spin) read from the associated atomic counter. This way, a thread only waits for the partitions that it needs at its current computational stage.

We first introduced our model in [32] and we are naming it: A *non-blocking data-driven barrier synchronization* model or *NBD^{2}BS*, for short.

Definition 5.2. *An algorithm is considered NBD^{2}BS if:*

1. *can be parallelized as a linear composition of smaller subproblems of the same type,*
2. *can represent its control flow as a number of barrier-synchronized stages,*
3. *can represent its data flow across those stages as a directed acyclic graph (DAG),*
4. *and the data access pattern, within a single stage of this graph, is fixed.*

NBD^{2}BS algorithms are expected to guarantee better thread progression conditions than their barrier-based counterparts. Barrier-synchronized parallel algorithms naturally block the progress of those threads waiting at a barrier. Yet, it is important to point out that barrier-based algorithms may exhibit a weak non-blocking property, like *obstruction-freedom*, based on its definition.

Lemma 5.1. *A barrier-synchronized parallel algorithm is at most obstruction-free.*

Proof. By contradiction. Assume, without loss of generality, one thread gets stuck before reaching the barrier at a given computational stage. By the obstruction-freedom definition, eventually this thread will run in isolation for a sufficient duration. Hence, reaching the barrier and allowing all threads to make progress. □

It is not hard to imagine a scenario where if each thread runs in isolation for a sufficient duration; then, all threads will eventually reach the barrier. However, in the context of fault tolerance or robustness against context-switching, the obstruction-freedom property does not hold the same progress conditions in the presence of arbitrary halting of other threads.

On the contrary, NBD^{2}BS parallel algorithms manifest a stronger non-blocking progression condition.

Lemma 5.2. *An NBD^{2}BS parallel algorithm is at most lock-free.*

Proof. By contradiction. Assume, without loss of generality, one thread gets stuck before reaching the barrier at a given computational stage. Following from the NBD^{2}BS definition, this thread will block at most one different thread. Hence, allowing the rest of the threads to make progress. □

The premise of the NBD^{2}BS model is to provide barrier-synchronized parallel algorithms with non-blocking progression guarantees. Chapter 6 shows the experimental validation of our model and the advantages of an NBD^{2}BS algorithm with respect to its barrier-synchronized counterpart, being *preempt-robust* and, hence, *performance-robust* the foremost of them.

5.3 Summary of Conditions for the Applicability and Properties of NBD^{2}BS

To determine whether an algorithm possesses the required properties so that the NBD^{2}BS model can be applied, a thorough assessment is necessary. The properties can be viewed as essential characteristics that ensure that the algorithm aligns with the requirements of our proposed technique.

Generally, to parallelize an algorithm means to partition the input data and assign it evenly to multiple execution threads to perform simultaneous computations over these subsets of data. Whenever a thread needs data from the computations of another thread, it waits until this data is *available*.

Instead of using a barrier synchronization construct, the idea behind NBD^{2}BS is to synchronize only those threads that are data-dependent at defined stages of an algorithm. This is achieved entirely by analyzing the data flow graph of the algorithm and exploiting the usage of post-wait synchronization semantics without adding any additional control flow construct.

Different algorithms may benefit from the NBD^{2}BS model. In chapter 6, we experimentally validate our model by showing its applicability to different algorithms. Ideally, what we are looking for in an algorithm is:

- The algorithm and its input data can be partitioned into smaller subproblems of the same type.
- The algorithm can be written in the form of multiple iterations and it exhibits fixed data dependencies across these iterations.

It is possible to parallelize these type of algorithms and insert a barrier synchronization construct to protect the data dependencies between each iteration. However, following these characteristics, it is also possible to associate a post-wait synchronization construct to each of the input data partitions and parallelize the algorithm with our NBD^{2}BS model. The synchronization is now driven by the data flow and not by the control flow anymore.

Table 5.1 shows a summary of the main differences of a traditional barrier-based parallel algorithm and an NBD^{2}BS-based parallel algorithm.

	Barrier-based	**NBD^{2}BS-based**
Synchronization semantics	Central barrier	Distributed post-wait
Synchronization degree	Global	Peer-to-peer
Synchronization category	Blocking	Non-blocking
Synchronization driven by	Control flow	Data flow
Best progression condition	Obstruction-freedom	Lock-freedom

Table 5.1: Differences between barrier-based and NBD^{2}BS-based parallel algorithms.

Chapter 6

Experimental Verification

The true method of knowledge is experiment.

William Blake

This chapter presents the experimental validation of our proposed NBD^{2}BS model for barrier-synchronized parallel SPMD applications. Multiple algorithms may benefit from our contribution, specially in the domain of high performance computing (HPC). We selected a handful of algorithms that are essential and core computational tasks in diverse fields such as, but not limited to: sorting, query processing, databases, signal processing, and machine learning.

The methodology is broadly applicable to other families of algorithms and it is general enough to cover the main obstacles that barrier-synchronized multithreaded parallel algorithms face-off in modern multicore CPU systems. The benefits of NBD^{2}BS algorithms with respect to their barrier-synchronized counterparts are promising.

The performance of each of the algorithms presented in this dissertation is characterized on a system with 16 x 1700 MHz (up to 4.1 GHz) x86 AMD Ryzen 7 CPUs with the following memory hierarchy: L1 Instruction 32 KiB (x8), L1 Data 32 KiB (x8), L2 Unified 512 KiB (x8), and L3 Unified 4096 KiB (x2).

We are interested in the scalability and performance robustness of the NBD^{2}BS algorithms and how they compare against the traditional barrier-synchronized implementation of said algorithms. In order to do this, the system is intentionally stressed to force context-switching among the threads that are participating in the execution and, hence, increasing the wait time of individual threads at barrier synchronization points. The system is stressed using the Linux *stress* tool [89] as a workload generator without any particular affinity to bound CPUs by spinning on *sqrt()*. The system stress rate is then defined as the ratio of CPUs that are already busy.

The experiments are carried out in a machine with a Linux 5.15 kernel. This version contains a Complete Fair Scheduler (CFS) to allocate CPU time to each execution thread [88]. The CFS algorithm is based on the idea of *virtual runtime*, where each thread is assigned a virtual runtime value that is proportional to the amount of CPU time it has consumed so far. Then, the thread with the smallest virtual runtime value is considered to be the next one to be allocated in a CPU.

The fair scheduler maintains a time-ordered red-black tree structure to track the virtual runtime of each thread. The leftmost thread in the tree is the one with the smallest virtual runtime and, as the system progresses forward, the executed threads are inserted more and more to the right part of the tree. This mechanism gives a chance for every thread to reach the leftmost part of the tree and, therefore, being assigned to a CPU core.

It is important to mention that this scheduler provides the only assumption we make in our thread progression model, which is that eventually all threads are scheduled for execution in a given CPU core.

6.1 Parallel Sorting

Sorting is one of the most researched problems in the field of computer science and one of the most important basic computational tasks that serves other major research areas and applications. Sorting is an important task because it enables efficient search and retrieval of data from large datasets. Sorted data is easier to analyze and visualize. Multiple scientific research areas may benefit from sorting algorithms that are more robust against the context-switching nature of oversubscribed modern general-purpose multicore CPU systems.

This section contributes two new parallel implementations of different sorting algorithms that were devised using our NBD^{2}BS model, bitonicsort and odd-even transposition sort. Our contributions convey non-blocking thread progression guarantees and robustness against context-switching in high-load scenarios. Our algorithms achieve performance-robustness since our novel implementations exhibit graceful degradation in their performance in the presence of unexpected system delays.

6.1.1 High Performance Sorting: Bitonicsort

Arguably the most ground breaking contribution to the family of parallel sorting, *bitonicsort* was originally developed by Kenneth E. Batcher in 1968 [11] and extensively studied ever since. The popularity of bitonicsort increased over the years due to its characteristics, being *massively parallelizable* the foremost of them.

Bitonicsort can sort any power of 2 sequence with a worst-case time complexity of $O(n \log^2 n)$ by repeatedly comparing pairs of elements that are at a fixed distance from each other and recursively sorting smaller subsequences. The correctness of the algorithm is based on the *bitonic theorem*.

Definition 6.1. *A bitonic sequence is a sequence of keys consisting of a first non-decreasing section followed by a non-increasing section, or a sequence on which a finite number of cyclic shifts will bring to the non-decreasing, non-increasing form.*

Theorem 6.1. *An odd-even compare-exchange on a shuffled bitonic sequence results in two mutually sorted sequences obtained by grouping, in order, the low (and high) outputs of the compare-exchange.*

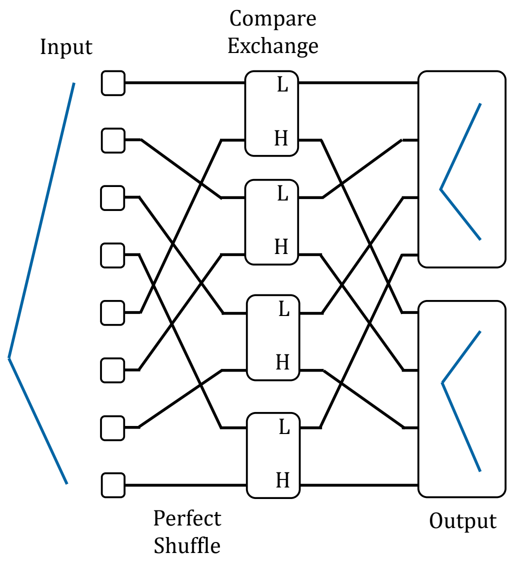

Figure 6.1: Bitonic theorem for a bitonic sequence of size 8.

Figure 6.1 refers to the bitonic theorem and shows how the outputs of the compare-exchange operations are grouped, generating two bitonic sequences sorted with respect to each other.

By definition, the smallest bitonic sequence that can be formed is of size 2 and, after the bitonic theorem, the sequence can be grouped in sequences of size 1 that are sorted with respect to each other in just one compare-exchange stage. Following, a bitonic sequence of size 4 requires two compare-exchange stages, the first stage is to generate two bitonic sequences of size 2 (out of the bitonic sequence of size 4) and the second stage sorts each bitonic sequence of size 2 (refer to the 4-bitonic sorter of figure 6.2). Then, for example, to sort a bitonic sequence of size 8 (refer to the 8-bitonic sorter of figure 6.2); the first stage generates two bitonic sequences of size 4, the second stage generates 4 bitonic sequences of size 2, and the last stage sorts each sequence of size 2. This means that each bitonic sorter can sort any bitonic sequence of size n in just $O(\log n)$ stages.

If multiple bitonic sorters are grouped together, the resulting network can sort any arbitrary sequence and not just any bitonic sequence. Suppose a scenario with an arbitrary input sequence of 8 elements and the desired output is the same sequence in non-descending order, see figure 6.2.

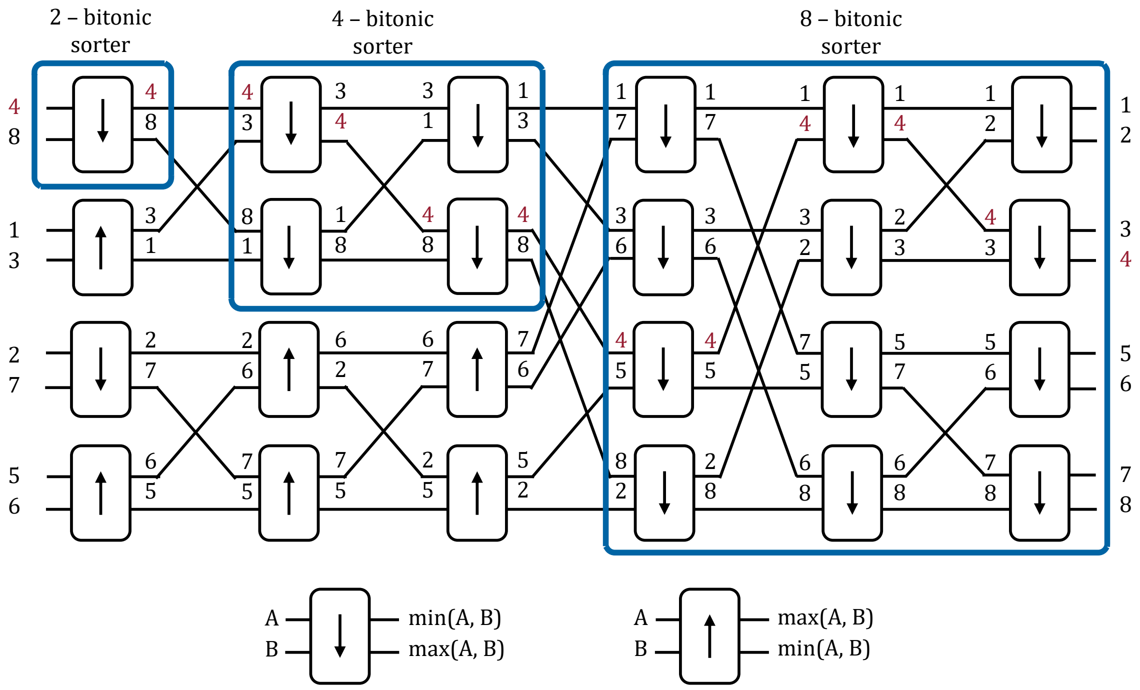

Figure 6.2: Bitonicsort network for an arbitrary sequence of size 8.

The idea is to generate a series of bitonic sequences which serve as input to the next stage of the sorting network. For instance, to sort a bitonic sequence of size 2, only one 2-bitonic sorter is needed. To generate a bitonic sequence of size 4, two 2-bitonic sorters, in opposite directions, are needed. Then, to sort the bitonic sequence of size 4, a 4-bitonic sorter is needed. Consequently, using two 4-bitonic sorters, in opposite directions, it is possible to generate a bitonic sequence of size 8 which, finally, can be sorted with an 8-bitonic sorter. This requires $O(\log n)$ bitonic sorters to sort any arbitrary sequence of size n, or $O(\log n)$

bitonic sorters of $O(\log n)$ stages each; leading to a total execution time complexity of $O(n \log^2 n)$ to sort any arbitrary sequence of size n.

On the other hand, it might not be straightforward to picture the bitonic sorting network in a shared memory computer architecture. Refer to figure 6.3 for a different way to represent the sorting network, where each compare-exchange operation takes place in a respective algorithmic stage.

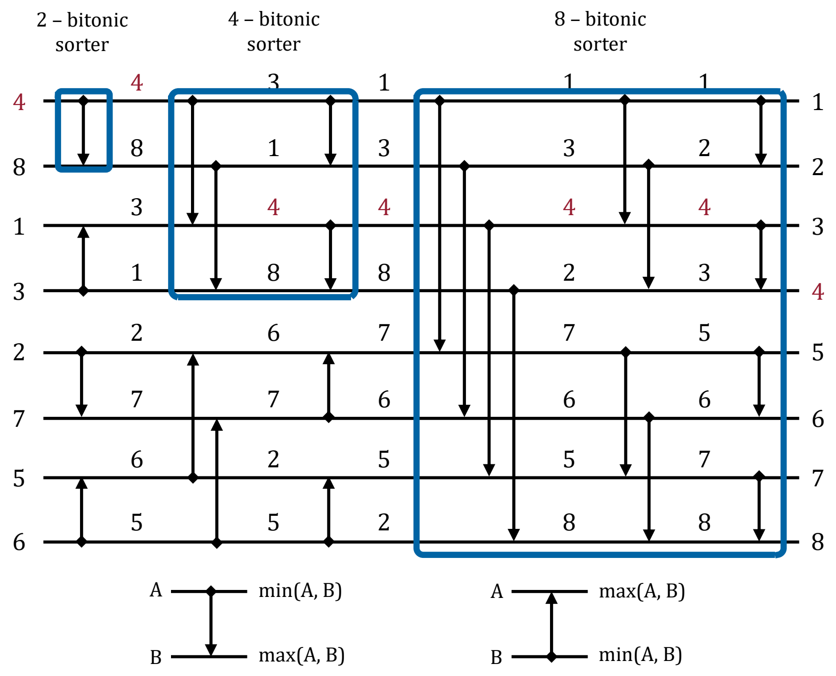

Figure 6.3: Bitonicsort algorithm pattern for an arbitrary sequence of size 8.

By transitivity, if doing compare-exchange operations sorts the input sequence. Then, doing *merge* operations, in previously sorted segments of data, sorts the input sequence as well. By definition, the low half (and high) of the resulting groups of the merge operation are mutually sorted, see figure 6.4.

The next sections present two different ways to parallelize this segmented bitonicsort. Section 6.1.1.1 presents the traditional way to parallelize it using a barrier synchronization. Then, section 6.1.1.2 presents a novel solution to the problem, by mapping it to our contributed model from chapter 5 and creating the first non-blocking parallel version of bitonicsort.

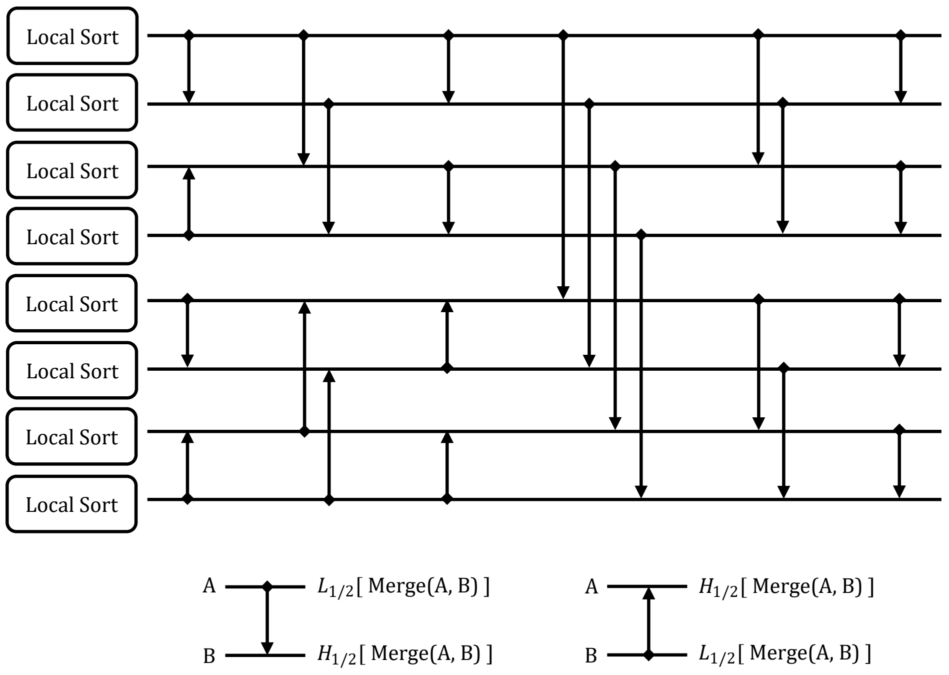

Figure 6.4: Bitonicsort algorithm pattern for an arbitrary sequence divided into segments.

6.1.1.1 Barrier-synchronized Parallel Bitonicsort

This section presents our barrier-synchronized multithreaded implementation of bitonicsort, see figure 6.5. The input is divided into segments of arbitrary size and evenly partitioned among all the execution threads.

Initially, individual segments are locally sorted and, then, merged in $O(\log^2 n)$ stages. Each stage is protected with a barrier synchronization construct ensuring that all merge operations in a given stage finish before a thread can proceed to its next computational stage. From figure 6.5, the blue thread and the green thread are forced to advance to each stage together and algorithm 2 is the pseudocode implementation that shows how to achieve this.

Lines 4 and 9 of algorithm 2 are the barrier synchronization constructs that impose the progression conditions for each thread. In other words, all segments need to be locally sorted before proceeding to the merging network and all merge operations need to finish before advancing to the next stage.

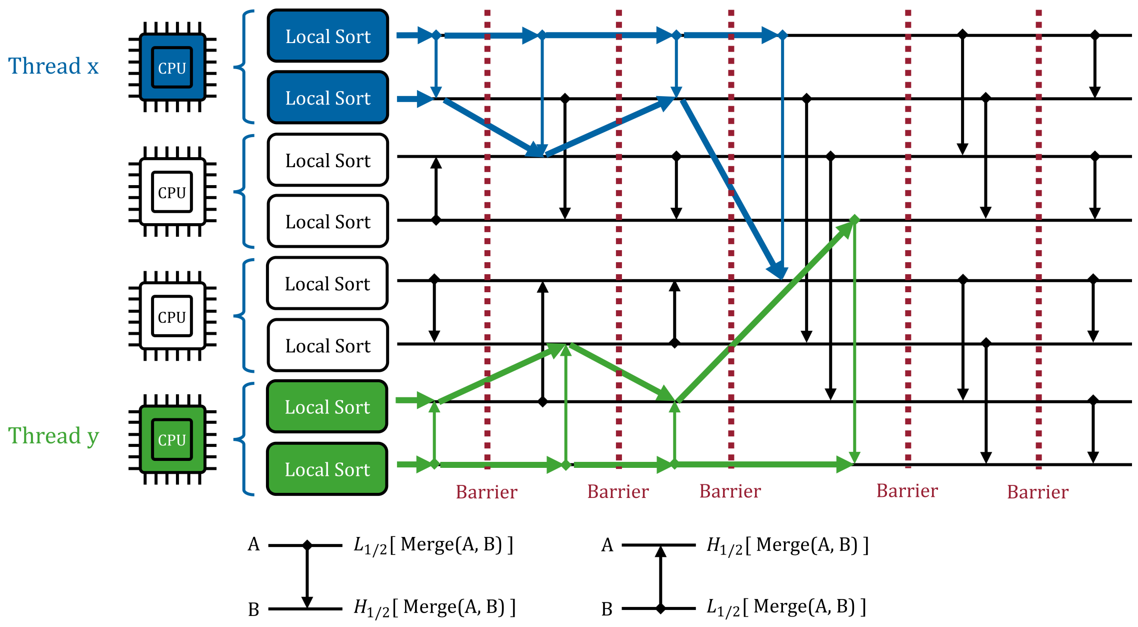

Figure 6.5: Barrier-synchronized parallel bitonicsort example.

Algorithm 2 Blocking barrier bitonicsort implementation.

```
1: procedure BlockingBarrierThread(&barrier)
2:     for segment : assigned_segments
3:         sort(&segment)
4:     wait(&barrier)                                  ▷ Wait for all threads
5:
6:     for each stage
7:         for segment_1, segment_2 : assigned_segments
8:             merge(&segment_1, &segment_2)
9:         wait(&barrier)                              ▷ Wait for all threads
```

6.1.1.2 Non-blocking Parallel Bitonicsort

This section presents our NBD^{2}BS multithreaded bitonicsort. To the best of our knowledge, this is the first time a multithreaded non-blocking version of bitonicsort is devised.

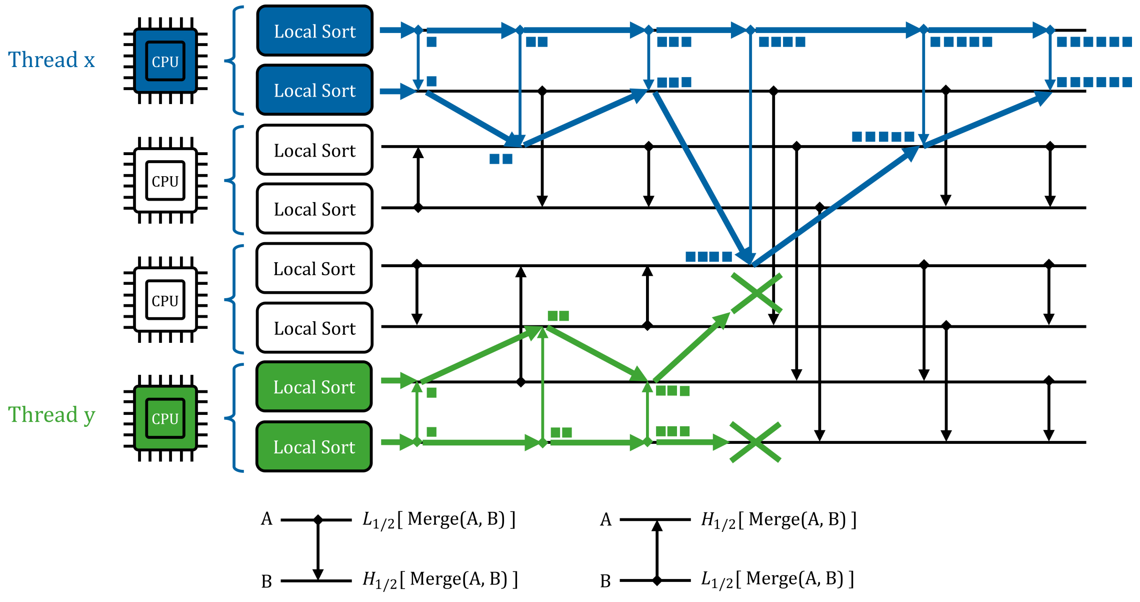

Figure 6.6: Non-blocking parallel bitonicsort example.

Figure 6.6 illustrates the thread progression under our new model. The input is divided into segments of arbitrary size and evenly partitioned among all the execution threads, but the synchronization is now driven by data. For example, once a segment is locally sorted, it is therefore ready to be merged in the next stage and this operation does not need to wait for the rest of the threads to finish their respective local sort operations.

Algorithm 3 shows the high-level implementation. Each thread keeps track of its current execution stage (lines 2, 7, and 20) and, once it finishes any given operation on any given segment, it will notify that the segment is ready for the next stage (lines 6, 18, and 19) by incrementing the associated atomic counter. Similarly, in order to execute the expected operations of a given stage, a thread only needs to wait the notification that the expected segment is ready for that particular stage (lines 11 and 13), i.e. spinning until the atomic counter of the expected segment is the same as the local count of the execution thread.

Algorithm 3 Non-blocking bitonicsort implementation.

```
procedure NonBlockingThread(atomic &segment_stage[number_of_segments])
    my_stage = 0

    for segment : assigned_segments
        sort(&segment)
        fetch_and_add(&segment_stage[segment], 1)        ▷ Post segment for next stage
    my_stage++

    for each stage
        for segment_1, segment_2 : assigned_segments
            while (my_stage != load(&segment_stage[segment_1]))    ▷ Wait for segment
                yield_cpu()
            while (my_stage != load(&segment_stage[segment_2]))    ▷ Wait for segment
                yield_cpu()

            merge(&segment_1, &segment_2)

            fetch_and_add(&segment_stage[segment_1], 1)   ▷ Post segment for next stage
            fetch_and_add(&segment_stage[segment_2], 1)   ▷ Post segment for next stage
        my_stage++
```

From figure 6.6, it is important to point out that even though the green thread gets blocked, the blue thread is able to make progress because there are no data dependencies between the two threads in the following stages.

6.1.1.3 Performance Analysis

In all the experiments that were carried out, the implementation of the barrier synchronization that was used is a central counter barrier like the one presented in algorithm 1 in section 3.3. Other implementations were not used as our goal is not the barrier itself, but the system-wide progress of the NBD^{2}BS algorithm against the barrier-synchronized implementation. Throughout this section, every time we refer to the non-blocking implementation of bitonicsort, it refers to the implementation based on our NBD^{2}BS model. The sequential implementation behavior is shown for reference purposes.

The first experiment shows how both algorithms scale when the input size increases, see figure 6.7. This figure show a comparison between the sequential (single-threaded) implementation of bitonicsort and the parallel implementations (NBD^{2}BS and barrier-based) using all available physical cores (16 threads). After removing the barrier synchronization, the performance of the NBD^{2}BS bitonicsort is slighlty better since it is no longer affected by all potential minor delays that could restrict the progress of each thread, i.e. it is no longer necessary to wait for the slowest thread.

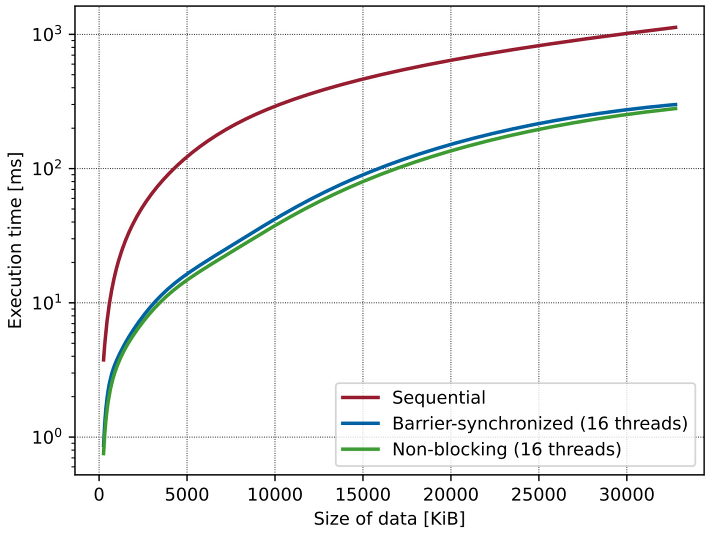

Figure 6.7: Bitonicsort performance behavior in terms of input size.

It is important to point out that the input data sizes force the system to interact with the memory hierarchy, i.e. the data does not fit only in L1 nor L2 caches. For smaller input data sizes, due to the distributed synchronization nature of our NBD^{2}BS model, the associated cost of the atomic operations to signal the stage of each segment can be more significant and impact its performance more than in the barrier-synchronized implementation. However, once the input data size is big enough and the system delays start to become more significant, our novel non-blocking bitonicsort performs better than its barrier-based counterpart.

The setup for the next experiments is sorting a total of 1024 KiB of data, where each element is a 32 bit integer. The data is partitioned into 256 segments of 4 KiB each, resulting in a bitonic sorting network of 36 stages.

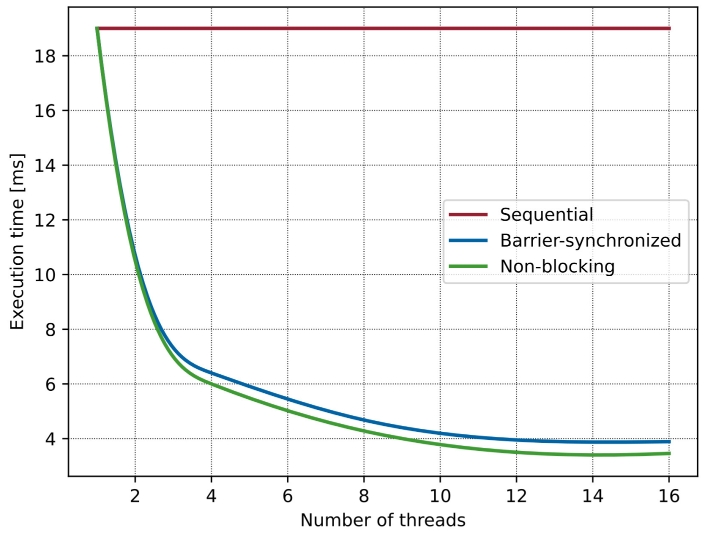

Figure 6.8: Bitonicsort performance behavior in terms of number of threads.

Figure 6.8 shows the same expected behavior but varying the number of execution threads. Our novel non-blocking bitonicsort performs slightly better than the traditional parallel barrier-based implementation.

However, these results are based on an ideal scenario: all CPUs are available to execute a given thread and no other tasks are contending for a CPU. The results become more interesting when the system is stressed (under high loads). As defined before, the system stress rate is the number of CPUs that are already executing another task. Thus, forcing context-switching and intentionally causing delays in the progress of the involved execution threads.

Figure 6.9 shows the performance robustness of the different implementations of bitonicsort. Notice the big performance discontinuity in the barrier-synchronized bitonicsort. When the system is busy enough, the performance of the parallel barrier-based implementation

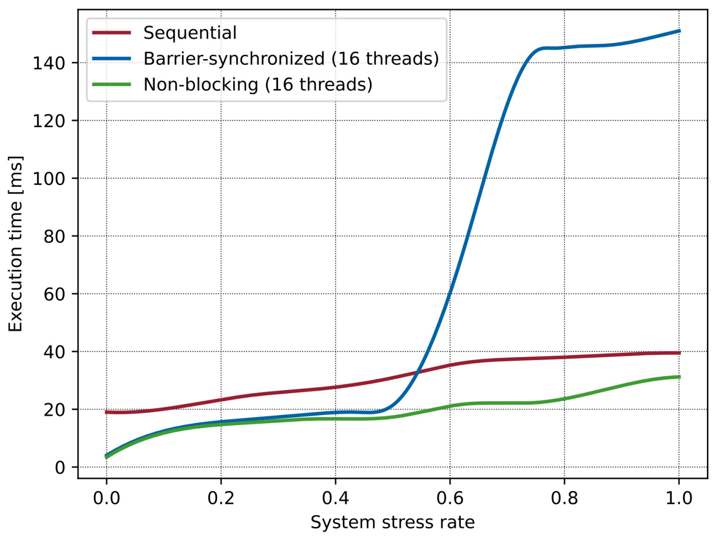

Figure 6.9: Bitonicsort performance behavior in terms of system stress rate.

is severely impacted, it could even be 3.8 times slower than its sequential counterpart and 4.8 times slower than our novel non-blocking implementation. Mainly, because one single preempted thread slows down the rest of the execution threads by blocking the progress of those waiting at a barrier. Non-blocking bitonicsort excels under stressed circumstances, achieving graceful performance degradation when the CPUs are highly contended. Providing robustness against thread preemption by removing the global blocking nature of each barrier.

Figures 6.10 and 6.11 show the performance robustness in terms of different number of execution threads and with different system stress rates. Figure 6.10 illustrates the performance behavior at 50% system stress rate: 8 out of 16 CPUs are already busy. Both bitonicsort implementations scale until they start using more than 8 threads. They start to decrease in their performance but the barrier-based decreases faster. At higher stress rates, figure 6.11 shows the performance characterization at 75% system stress rate: 12 out of 16 CPUs are already busy. Similarly, both implementations scale while there are CPUs available to run their threads. After that, there is a big cliff in the performance of the barrier-synchronized bitonicsort implementation, whereas non-blocking bitonicsort achieves graceful degradation.

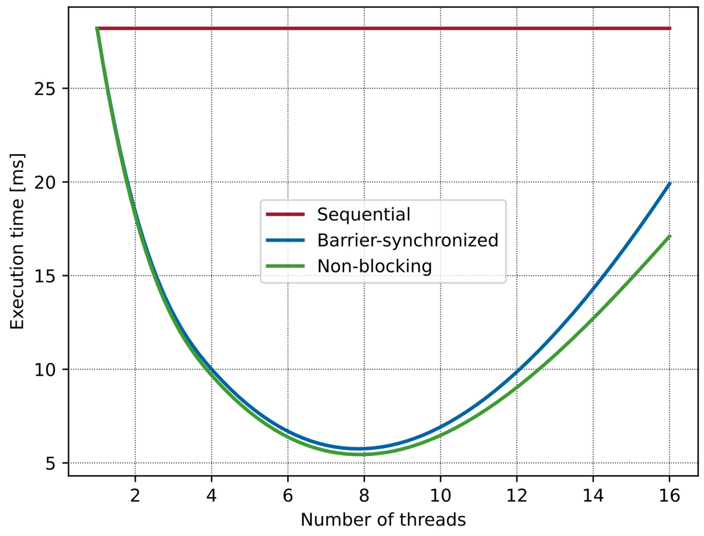

Figure 6.10: Bitonicsort performance robustness at 50% system stress rate.

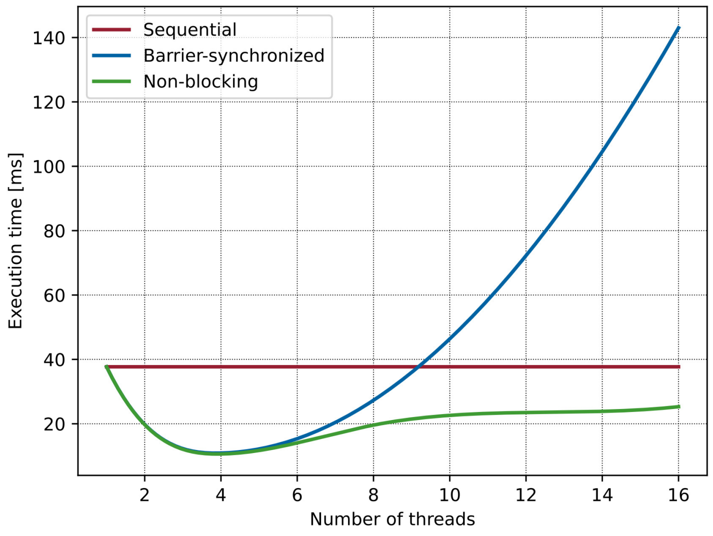

Figure 6.11: Bitonicsort performance robustness at 75% system stress rate.

6.1.1.4 Bitonicsort and Multiway Mergesort Comparative Analysis

The previous section covered the performance characterization of our novel non-blocking bitonicsort implementation based in the NBD^{2}BS model. In this section, a comparative analysis against the state-of-the-art parallel sorting implementation is presented.

The GNU parallel extension to the C++ standard library uses OpenMP [91] to implement *Multiway Mergesort* or MWMS, for short. This is an algorithm that can sort any sequence of size n in $O(n \log n)$ time complexity. The parallel implementation is based on Peter Sanders algorithm [80, 82]. Initially, the input data is partitioned into N segments of arbitrary size, where N is the number of execution threads. Each thread creates a local temporal copy of its assigned partition and sorts it. Then, each thread performs a multisequence partitioning that will result in the splitting points where each thread will merge its final sequence. Finally, each thread executes a *N-way* merge into their respective places in the original array.

It is important to point out the barrier synchronization points that this algorithm exhibits. The first one is that all local sorts must finish before proceeding into the partitioning stage. Then, each thread needs to conclude its partitioning stage before the merging stage begins. Finally, the merging must end before each thread can start deleting its temporary local storage. The MWMS algorithm consists of 4 algorithmic stages despite of the size of the input. Observe that this is the first big difference with respect to bitonicsort, the number of stages in the bitonicsort network depends on the number of input segments. The granularity of the operations performed is also different, the partitioning time complexity of N segments of size k is $O(k \log N)$. Whereas, one merge operation takes only $O(k)$; implying that bitonicsort may reach a barrier synchronization point faster than multiway mergesort. Notice that smaller operations are more likely to finish before the next context-switch happens.

Figure 6.12 shows the performance behavior under ideal circumstances, with the same experimental setup as section 6.1.1.3, and using the sequential *std::sort* (hybrid between quicksort and heapsort) as performance baseline. MWMS algorithm exhibits better performance, at least a factor of 2 faster than our non-blocking bitonicsort.

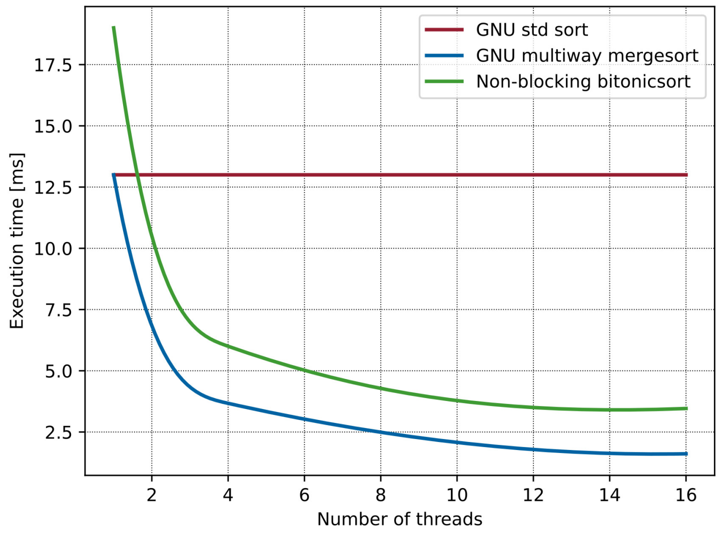

Figure 6.12: Bitonicsort and Multiway Mergesort varying the number of threads.

Figure 6.13 illustrates the behavior under stressed circumstances. NBD^{2}BS bitonicsort is more robust, its performance degrades similarly to the sequential algorithm that runs in just one CPU core.

Another scenario to explore in terms of performance robustness and scalability is where multiple instances of the same program run concurrently and compete against each other for the CPU cores. These situations commonly occur in server or workstation environments where numerous requests from different clients are being processed. Figure 6.14 illustrates this structure, where the number of concurrent instances of the same program varies. In this experiment, 4096 KiB of 32 bit integers are being sorted and it is shown that our non-blocking bitonicsort is more performance-robust than the GNU multiway mergesort implementation. For example, in the case where 64 concurrent instances (of 16 threads each) compete for the available 16 CPUs, the non-blocking bitonicsort is at least a factor of 3 faster than the GNU parallel multiway mergesort.

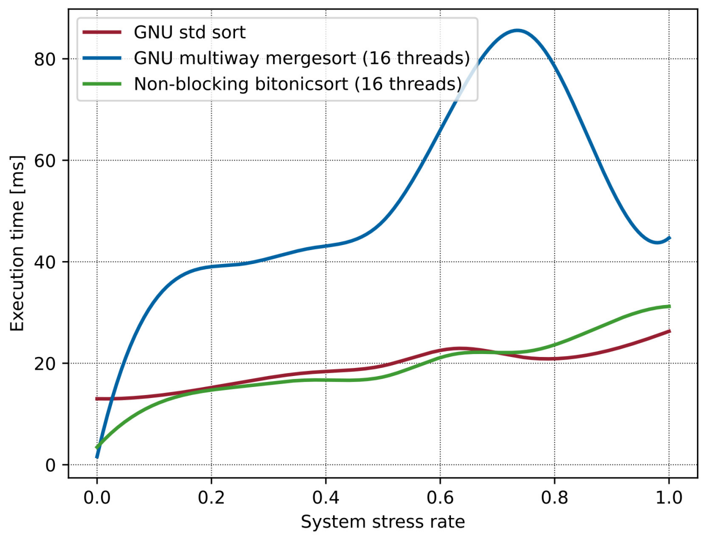

Figure 6.13: Bitonicsort and Multiway Mergesort varying the system stress rate.

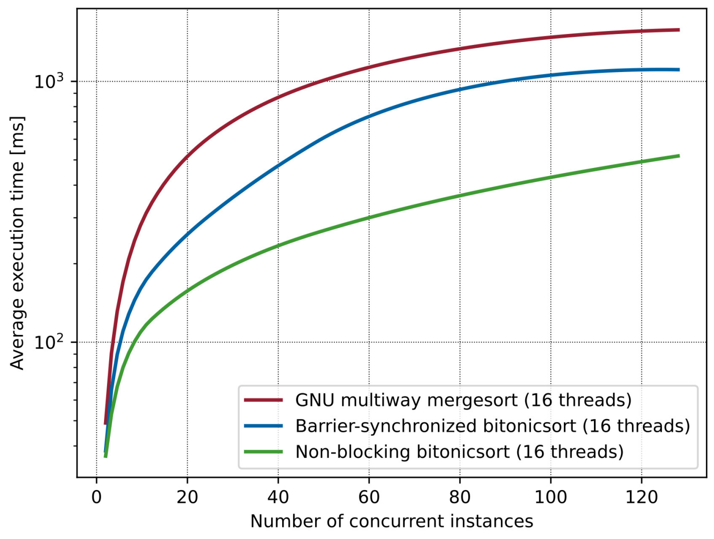

Figure 6.14: Bitonicsort and Multiway Mergesort varying the number of concurrent instances.

6.1.2 A Pedagogical Example: Odd Even Transposition Sort

Other algorithms with different, but regular, access patterns are good candidates for our NBD^{2}BS model as well, like *Odd-Even Transposition Sort*, figure 6.15. We learned about this algorithm from Selim G. Akl's book in design and analysis of parallel algorithms [5].

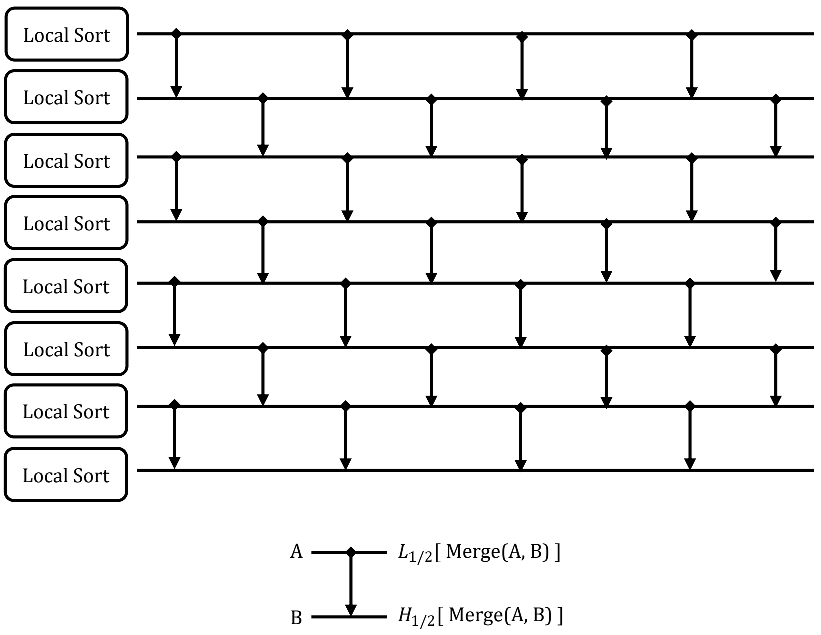

Figure 6.15: Odd-even transpose pattern for an arbitrary sequence divided into segments.

The algorithm starts by dividing the input data into segments of arbitrary size which will be assigned to all participating execution threads. First, each thread locally sorts their assigned segments of data. Then, two steps are performed repeatedly. In the first step, each odd-numbered thread T_i merges two segments, S_i and S_{i+1} into a sorted sequence. The second step is identical except that it is performed by all even-numbered execution threads. Finally, these two steps are repeated alternately and after $\lceil N/2 \rceil$ iterations (being N the number of input segments) the input data will be globally sorted.

Let n be the total number of data elements to be sorted and, without loss of generality, assume the data is divided into N segments. Then, the first step to locally sort individual segments takes $O((n/N)\log(n/N))$ steps and each merge requires $O(n/N)$. Therefore, the total cost in parallel is $O((n/N)\log(n/N)) + \lceil N/2 \rceil \times O(n/N)$ and, then, the cost can be reduced to $O(n \log n) + O(nN)$, which can be optimal when $N \leq \log n$. The characteristics and extreme simplicity of this algorithm makes it an ideally candidate for the classroom.

6.1.2.1 Barrier-synchronized Parallel Odd Even Transposition Sort

This section presents our barrier-synchronized multithreaded implementation of odd-even transposition sort, figure 6.16. First, the input is divided into segments of arbitrary size and evenly partitioned among all the execution threads.

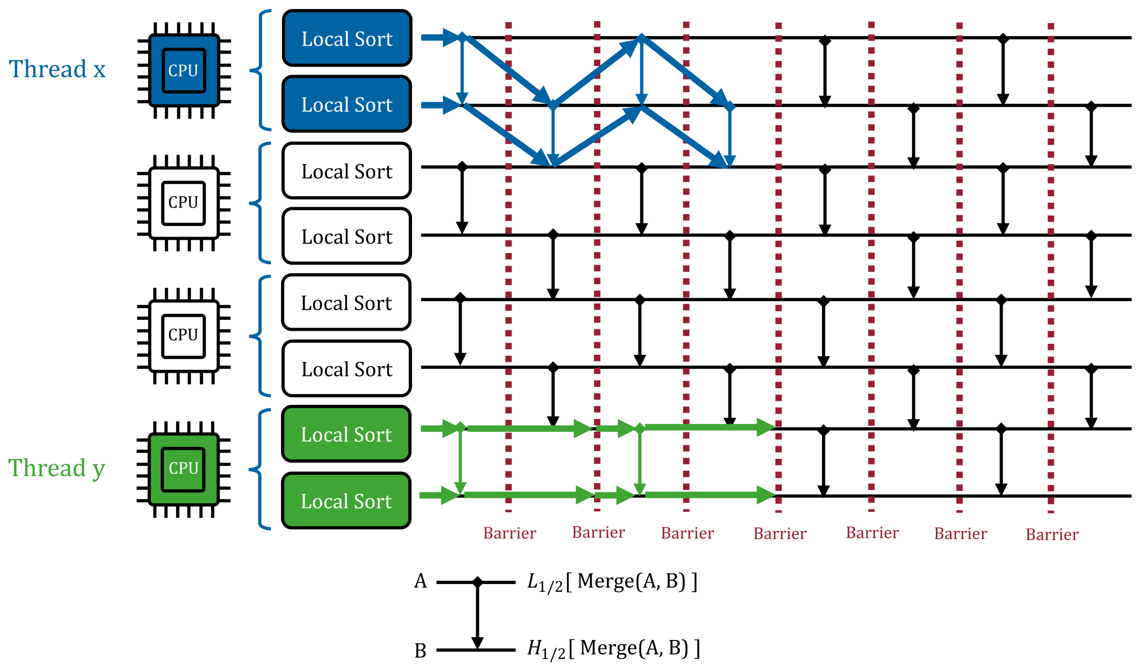

Figure 6.16: Barrier-synchronized parallel odd-even transposition sort example.

All individual segments are locally sorted and, then, merged together into larger sorted sequences in the already described odd-even transposition fashion. As previously seen, a series of barrier synchronization constructs, between each stage of the algorithm, are necessary to ensure that, for example, no even-numbered merge operation can occur without all preceding odd-numbered merge operations finishing first.

Algorithm 4 illustrates the high-level implementation. It is important to point out that due to the odd-even iterations nature of this algorithm, there are threads that will not perform any useful work or that will execute less work at some stages. Lines 8 and 9 illustrate this behavior, a thread at an even-numbered stage will not merge the last segment with any other segment. Hence, proceeding to the barrier immediately and wait for the rest of the threads to finish the merge computations.

Algorithm 4 Blocking barrier odd-even transposition sort implementation.

```
 1: procedure BlockingBarrierThread(&barrier)
 2:     for segment : assigned_segments
 3:         sort(&segment)
 4:     wait(&barrier)                                    ▷ Wait for all threads
 5:
 6:     for each stage
 7:         for segment_1, segment_2 : assigned_segments
 8:             if (&segment_1 == &last_segment)
 9:                 break
10:             merge(&segment_1, &segment_2)
11:         wait(&barrier)                                ▷ Wait for all threads
```

The unbalanced computational nature of the algorithm creates room for experimentation. Next section demonstrates how we map the algorithm to our proposed non-blocking model and how this helps accelerate unbalanced computations from different threads.

6.1.2.2 Non-blocking Parallel Odd Even Transposition Sort

This section presents our NBD^{2}BS multithreaded implementation of the odd-even transposition sort, figure 6.17. The input data is divided into segments of arbitrary size and evenly partitioned among all execution threads, but the synchronization is now driven by data. For example, once two segments are merged in their respective stage, both segments are *posted* as ready for their next stage.

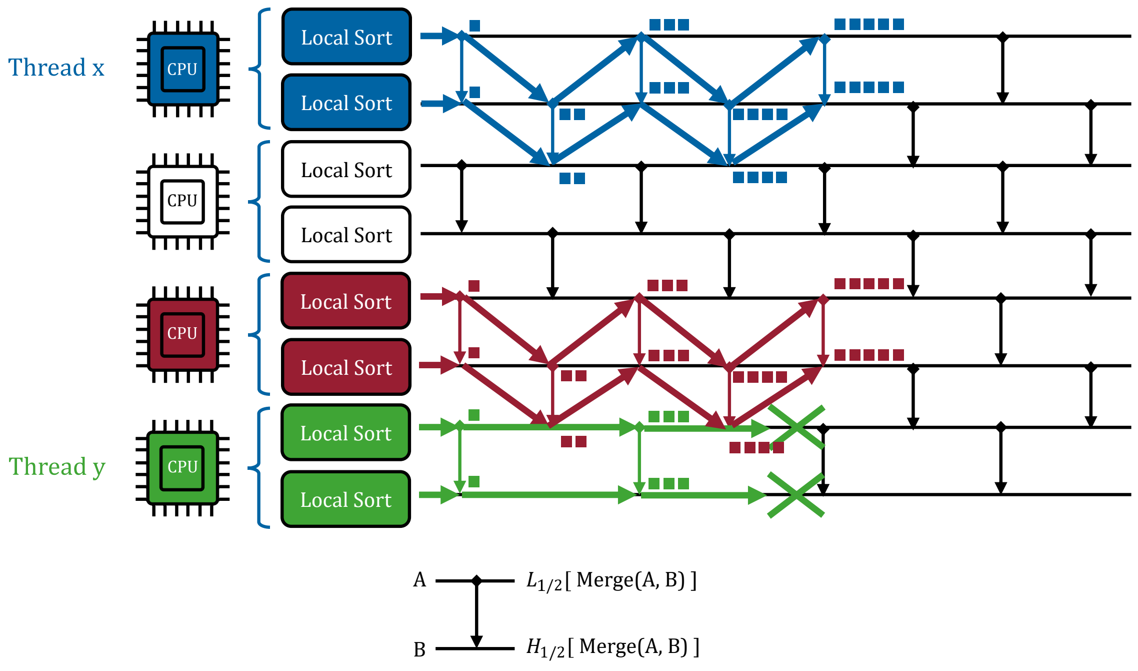

Figure 6.17: Non-blocking parallel odd-even transposition sort example.

Figure 6.17 illustrates the non-blocking thread progression under the NBD^{2}BS model. For example, the blue thread makes progress even though, at some point, the green thread suffers from delays. It is important to point out the data dependencies exhibited by the green thread and the red thread. From the figure, the green thread in its second stage waits for the segments to be advanced to the third stage by the red thread.

Algorithm 5 shows the high-level implementation of our non-blocking odd-even transposition sort. Notice how extra logic is necessary to handle the odd-numbered and the even-numbered stages separately, due to the unbalanced nature of the computations of the different threads.

In line 11, if a thread happens to be in an odd-numbered iteration, the very first segment should not be merged and, hence, move it forward to the next stage. Same for the last segment, line 13, it needs to be posted for the next stage and the thread can also continue to the next stage. Lines 17 and 19 show how each thread waits for its respective segments, which are going to be merged in line 22. Finally, lines 24 and 25 post the recent merged segments to be used in the next algorithmic stage.

It is important to point out that this implementation does not balance the amount of work among the execution threads, but makes it more robust against unexpected context-switches due to its peer-to-peer synchronization scheme.

Algorithm 5 Non-blocking odd-even transposition sort implementation.

```
procedure NonBlockingThread(atomic &segment_stage[number_of_segments])
    my_stage = 0

    for segment : assigned_segments
        sort(&segment)
        fetch_and_add(&segment_stage[segment], 1)          ▷ Post segment for next stage
    my_stage++

    for each stage
        for segment_1, segment_2 : assigned_segments
            if (&segment_1 == &second_segment)
                fetch_and_add(&segment_stage[first_segment], 1)       ▷ Post for next stage
            if (&segment_1 == &last_segment)
                fetch_and_add(&segment_stage[last_segment], 1)       ▷ Post for next stage
                break

            while (my_stage != load(&segment_stage[segment_1]))     ▷ Wait for segment
                yield_cpu()
            while (my_stage != load(&segment_stage[segment_2]))     ▷ Wait for segment
                yield_cpu()

            merge(&segment_1, &segment_2)

            fetch_and_add(&segment_stage[segment_1], 1)  ▷ Post segment for next stage
            fetch_and_add(&segment_stage[segment_2], 1)  ▷ Post segment for next stage
        my_stage++
```

6.1.2.3 Performance Analysis

The experimental setup is equal to the previous section. This is, our experiments sort a total of 1024 KiB of data and each element is a 32 bit integer. The data is partitioned into 256 segments of 4 KiB each, resulting in an odd-even sorting network of 256 stages. The first important detail to notice is that the odd-even transposition network is larger than the bitonicsort network. Again, every time we refer to the non-blocking implementation of odd-even transposition sort, it refers to the implementation based on our NBD^{2}BS model. The sequential implementation behavior is shown for reference purposes.

Under ideal conditions (figure 6.18), the performance of both implementations (barrier-based and non-blocking) is similar. However, our non-blocking implementation, based in the NBD^{2}BS model, excels under high load scenarios. Figure 6.19 shows the performance behavior when the system is under different stress rates. The performance remains similar when 8 out of our 16 CPUs are already busy, but the barrier-synchronized implementation still exhibits a bigger performance cliff when the system is stressed more than 50%.

An unusual characteristic of odd-even transposition sort is that sorting network is long enough that even the performance of our non-blocking implementation suffers more than its sequential counterpart. Figures 6.20 and 6.21 illustrate the scalability of both implementations under high load cricumstances to better understand this phenomena.

Figure 6.20 describes a scenario with 50% stress rate, i.e. half of our available cores are already running other tasks. At this point, odd-even transposition sort still benefits from exploiting multithreading up to the number of available CPU cores. Then, both implementations start to degrade in performance but the NBD^{2}BS version of the algorithm degrades gracefully and still performs better than its sequential counterpart.

Figure 6.21 shows a scenario where 75% of the CPU cores are already running other tasks. Both multithreaded versions of the algorithm scale poorly while the number of used threads increase and the sequential implementation achieves better performance.

Odd-even transposition sort is not as robust in performance as bitonicsort exhibited. The main culprit is that the sorting network is composed by a larger number of algorithmic stages and their unbalanced computational nature: one thread does not execute any work in all even-numbered stages.

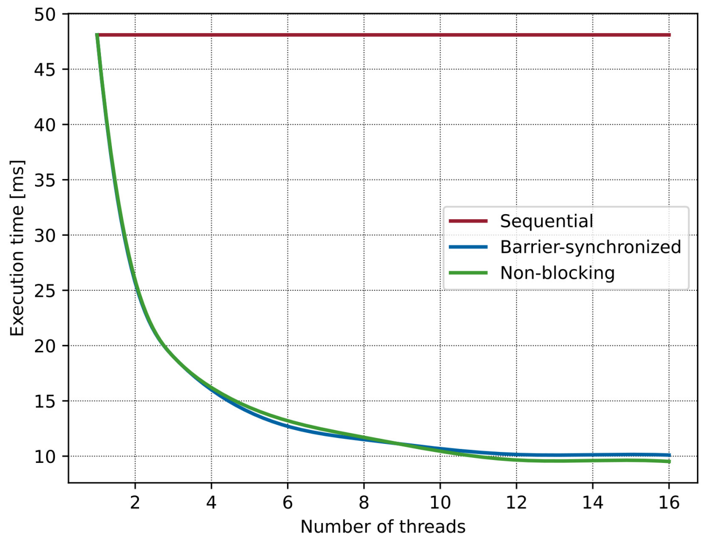

Figure 6.18: Odd-even sort performance behavior in terms of number of threads.

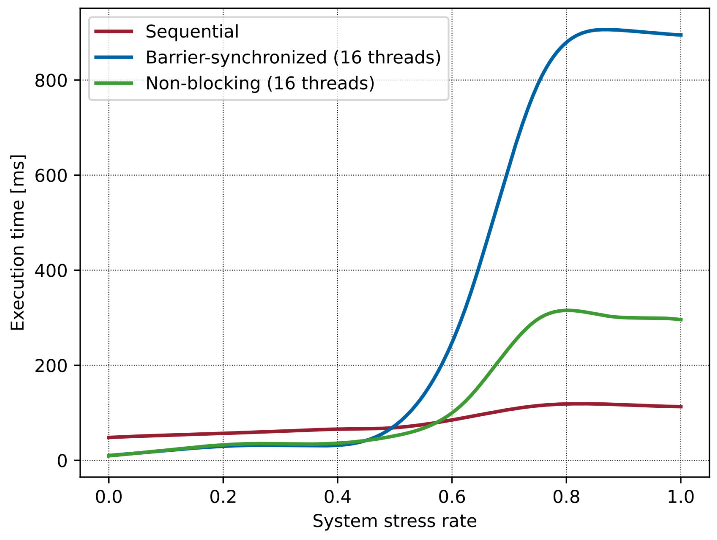

Figure 6.19: Odd-even sort performance behavior in terms of system stress rate.

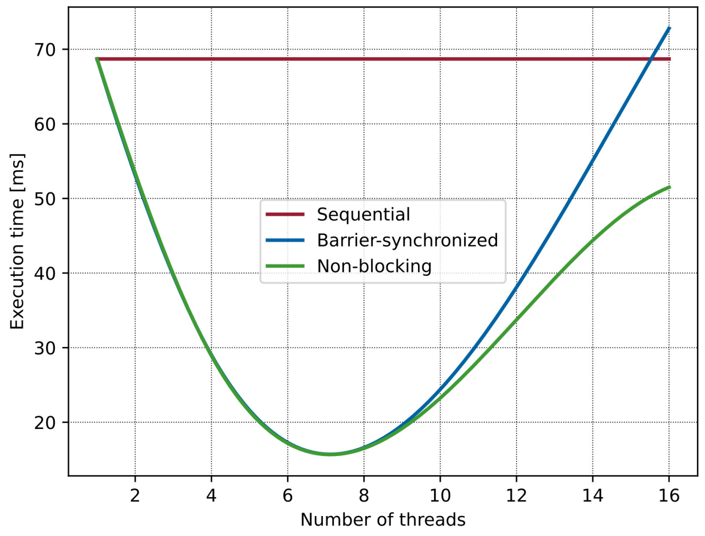

Figure 6.20: Odd-even sort performance robustness at 50% system stress rate.

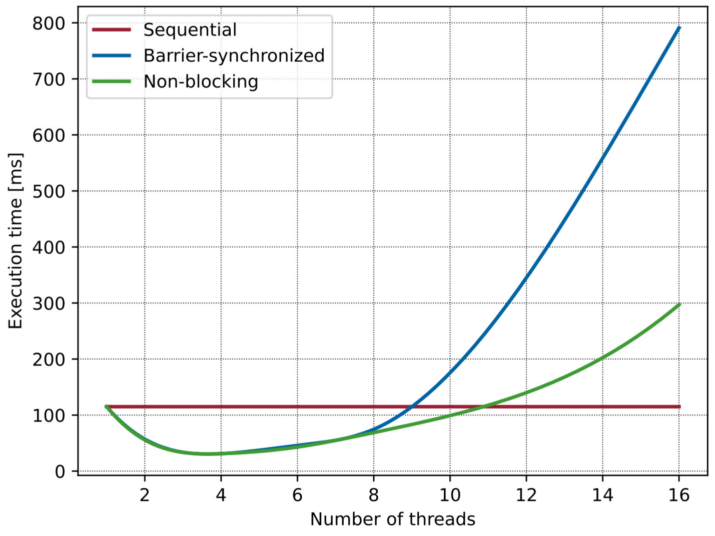

Figure 6.21: Odd-even sort performance robustness at 75% system stress rate.

6.2 Parallel Signal Processing

In the field of signal processing, multiple algorithms can also take advantage of the many CPU cores of modern systems to accelerate their computations. Examples of signal processing applications are, but not limited to: image reconstruction, filtering, audio and speech recognition, radar and communications processing, computer vision, and convolutional neural networks in machine learning [28, 46, 49, 59, 60, 85, 92, 96, 100]. Consequently, efficient and robust parallel implementations are of paramount importance.

The most important operation of these signal processing applications is the well-known *Fourier Transform*. The Fourier transform is an important mathematical transformation that decomposes a signal in its frequency components. This is, the transformation converts a signal from the *time* domain (signal amplitude as a function of time) to the *frequency* domain (a composition of the amplitude of each frequency component of the signal).

In this dissertation, the Fourier transform operation is just briefly discussed since our main goal is to apply our novel NBD^{2}BS model to provide thread preemption robustness and, hence, performance robustness to this important operation. See [12] for a thorough introduction to the Fourier transform.

For simplicity, only discrete-time signals with finite number of samples are considered. In other words, the actual operation to be performed is the *Discrete Fourier Transform* or DFT, for short. Arguably, J. W. Cooley and J. W. Tukey devised the most important algorithm to date to compute the DFT [19]. This algorithm is known as the Cooley-Tukey *Fast Fourier Transform* (FFT) and it is the next experimental vehicle for our NBD^{2}BS model.

6.2.1 The Core Operation: Fast Fourier Transform

The Cooley-Tukey algorithm is a widely used FFT implementation. The main idea is to divide the input sequence into smaller sequences and recursively applying the FFT on them. There are different ways to implement it, but one of the most common is the radix-2 FFT, which means that the algorithm works on input sequences whose size is a power of 2 and, then, factoring an N-point sequence into subsequences of size $N/2$, $N/4$, and so on, until each subsequence has a size of length 2. Hence, the time complexity is bounded by $O(N \log N)$ steps.

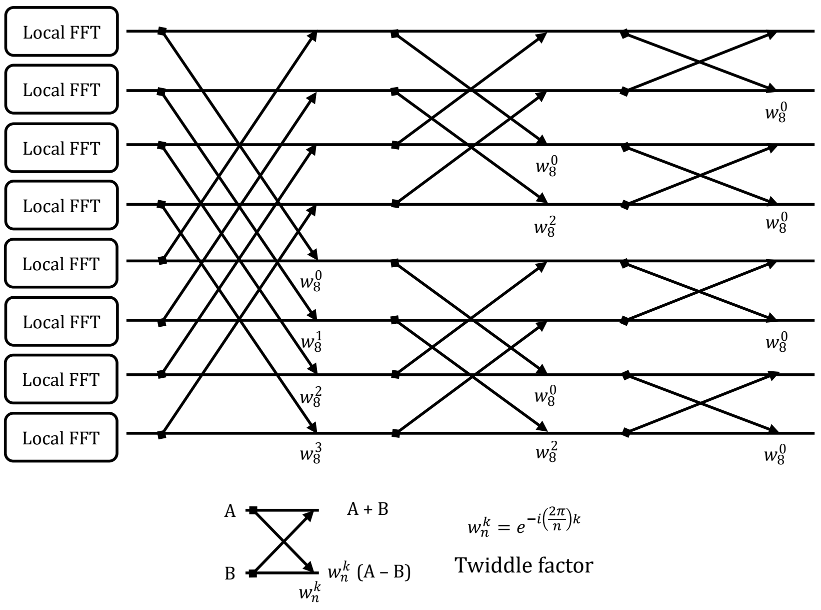

Figure 6.22: FFT algorithm pattern for an arbitrary sequence divided into segments.

Figure 6.22 illustrates the structure of the algorithm. There are two type of algorithmic structures, which main difference is the order in which the operations are performed; one is decimation in time and the other one is decimation in frequency [12]. This example follows a decimation in frequency approach to solve the FFT and, also, an *out-of-core* approach. The out-of-core approach means that the input sequence is large enough that in order to compute the FFT, it is better to distribute it across multiple execution units where local FFTs will take place. Finally, the results will be combined together following the same Cooley-Tukey algorithmic structure [15, 16, 62, 95].

The main operation in the Cooley-Tukey algorithm is known as the *butterfly* operation and it is defined at the bottom of figure 6.22, where the high-part is expressed as the sum of the two inputs and the low-part is determined as the difference of the two inputs multiplied by a factor. A *twiddle* factor is used to represent a given frequency component for a given sample and it is written as $W_N^k = e^{-2\pi ik/N}$. Finally, after $O(N \log N)$ stages, the final FFT result will be produced.

The next sections show our two parallel implementations of the out-of-core FFT. First, the traditional implementation using a barrier synchronization construct between the stages of the algorithm and a novel non-blocking parallel implementation using our proposed NBD^{2}BS model.

6.2.1.1 Barrier-synchronized Parallel Fast Fourier Transform

This section presents our barrier-synchronized multithreaded implementation of the out-of-core Cooley-Tukey FFT, figure 6.23. The input is divided into segments of arbitrary size and evenly partitioned among all execution threads.

At first, the FFT is applied to individual segments only and, then, combined together using the defined butterfly operation in $O(\log n)$ Cooley-Tukey-like stages. Then, a barrier synchronization construct enforces that no thread executes past a given stage until all threads finish their operations at said stage. For example, from figure 6.23, the blue thread and the green thread are forced to advance to each stage together. Algorithm 6 shows the high-level implementation, notice that lines 4 and 9 represent the barrier synchronization.

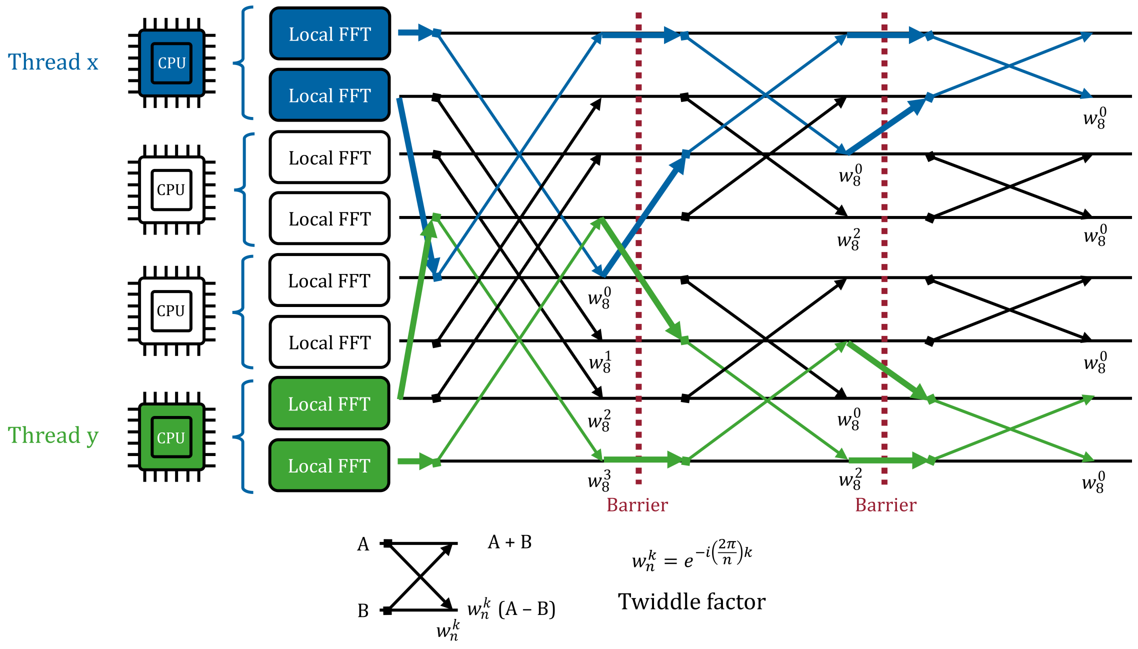

Figure 6.23: Barrier-synchronized parallel FFT example.

Algorithm 6 Blocking barrier FFT implementation.

```
1: procedure BlockingBarrierThread(&barrier)
2:     for segment : assigned_segments
3:         fft(&segment)
4:     wait(&barrier)                                   ▷ Wait for all threads
5:
6:     for each stage
7:         for segment_1, segment_2 : assigned_segments
8:             butterfly(&segment_1, &segment_2)
9:         wait(&barrier)                               ▷ Wait for all threads
```

6.2.1.2 Non-blocking Parallel Fast Fourier Transform

This section presents our NBD^{2}BS multithreaded implementation of the out-of-core Cooley-Tukey FFT, see figure 6.24 for reference.

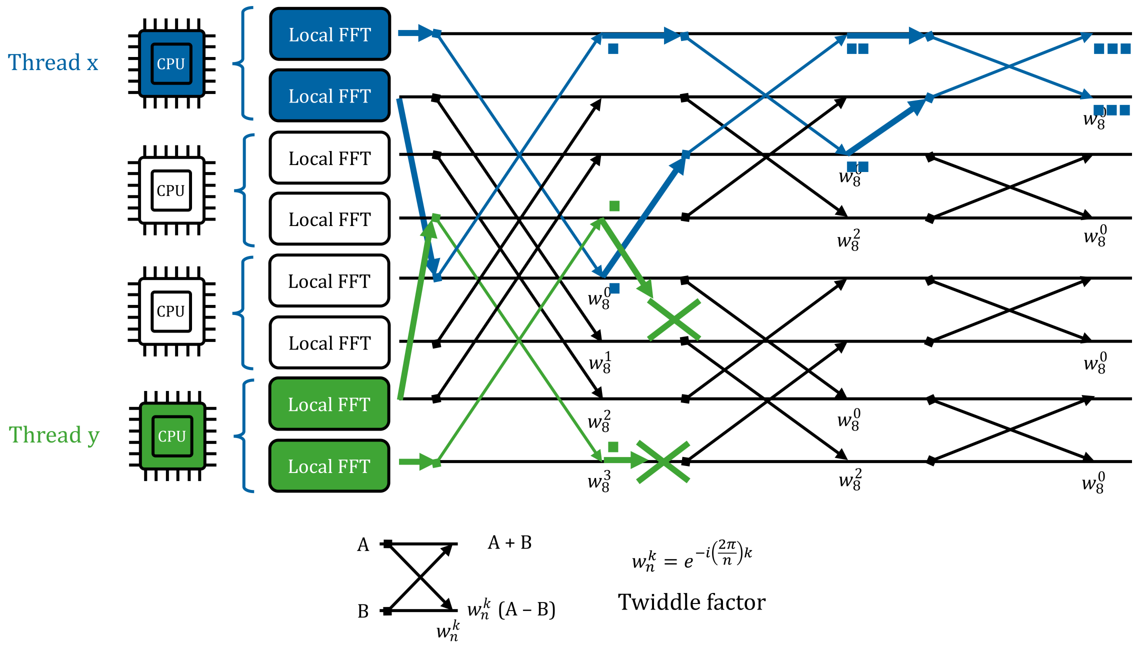

Figure 6.24: Non-blocking parallel FFT example.

The main aspect of this implementation is that the synchronization is now driven by data. Once a thread is done with a segment, the segment will be posted to its respective next stage regardless of the progress of the rest of the threads. From figure 6.24, notice that the blue thread can proceed to further stages, even though the green thread suffers from delays.

The high-level implementation is presented in algorithm 7. It is important to point out that each thread only waits for the segments it needs in its current stage (lines 11 and 13). Hereafter, once a thread finishes a butterfly operation with a given pair of segments, those segments are posted for use in the next stage (lines 6, 18, and 19). Finally, if a thread finishes all its respective operations, it will move forward to the next stage (line 20) and, then, it waits for its respective segments.

Algorithm 7 Non-blocking FFT implementation.

```
procedure NonBlockingThread(atomic &segment_stage[number_of_segments])
    my_stage = 0

    for segment : assigned_segments
        fft(&segment)
        fetch_and_add(&segment_stage[segment], 1)          ▷ Post segment for next stage
    my_stage++

    for each stage
        for segment_1, segment_2 : assigned_segments
            while (my_stage != load(&segment_stage[segment_1]))    ▷ Wait for segment
                yield_cpu()
            while (my_stage != load(&segment_stage[segment_2]))    ▷ Wait for segment
                yield_cpu()

            butterfly(&segment_1, &segment_2)

            fetch_and_add(&segment_stage[segment_1], 1)  ▷ Post segment for next stage
            fetch_and_add(&segment_stage[segment_2], 1)  ▷ Post segment for next stage
        my_stage++
```

6.2.1.3 Performance Analysis

The experimentation consists of an input sequence consists in 128 Ki-samples of a pair (real and imaginary) of 32 bit floating point elements. The data is partitioned into segments of 128 elements and, thus, resulting in a butterfly network of 10 stages. Every time we refer to the non-blocking implementation of the FFT, it refers to the implementation based on our NBD^{2}BS model. The sequential implementation behavior is shown for reference purposes.

In an ideal scenario, figure 6.25 shows that both implementations exhibit the same behavior, mainly due to the small amount of computational stages. Figure 6.26 illustrates the tolerance to different system stress rates. This behavior has been consistently observed, the parallel barrier-synchronized version of the FFT suffers a big performance cliff towards high load scenarios. Our NBD^{2}BS FFT shows better robustness and performance than its barrier-based counterpart by a factor of 2.

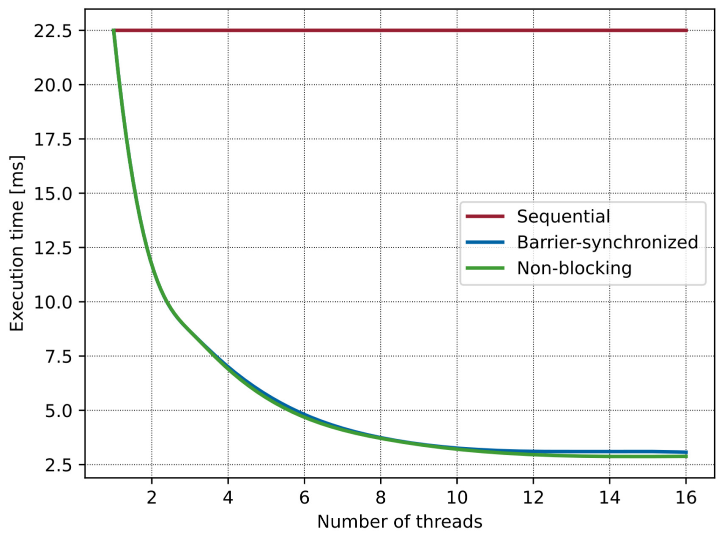

Figure 6.25: FFT performance behavior in terms of number of threads.

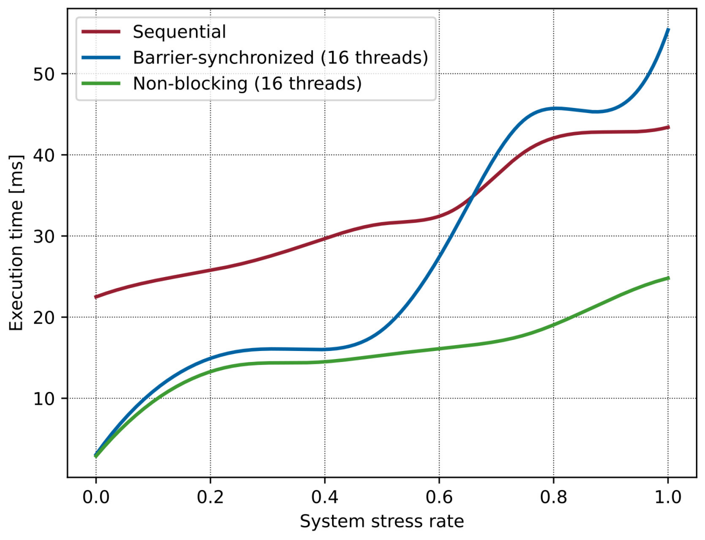

Figure 6.26: FFT performance behavior in terms of system stress rate.

Figures 6.27 and 6.28 present the scalability of our implemented versions of the FFT algorithm under different circumstances. First, figure 6.27 illustrates the behavior when half of the available CPU cores are busy running another task. From the picture, both parallel versions of the algorithm scale similarly until they start using more than 8 execution threads. Both FFT algorithms degrade in performance, but they still benefit from the multithreaded parallelism.

On the other hand, figure 6.28 demonstrates the robustness of our non-blocking NBD^{2}BS FFT implementation. In this scenario, the system is already stressed by 75%, i.e. 12 out of the 16 available CPU cores are running a different task. The non-blocking FFT outperforms its barrier-synchronized counterpart by more than a factor of 2 and its performance degrades gracefully when the system is under high loads.

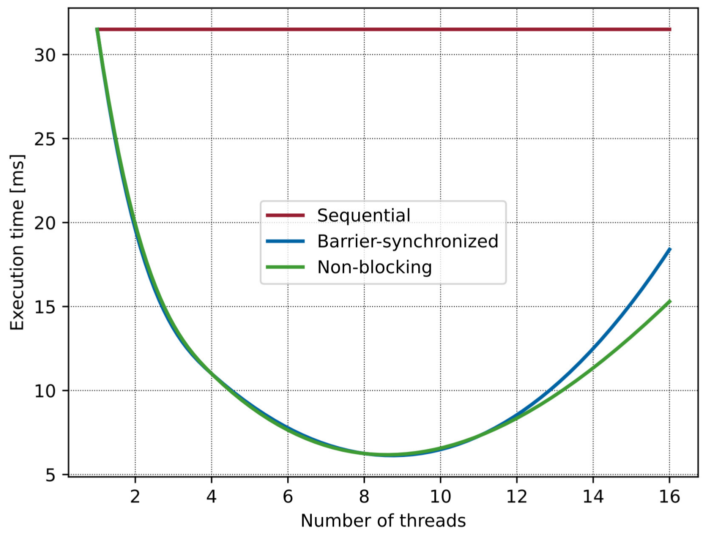

Figure 6.27: FFT performance robustness at 50% system stress rate.

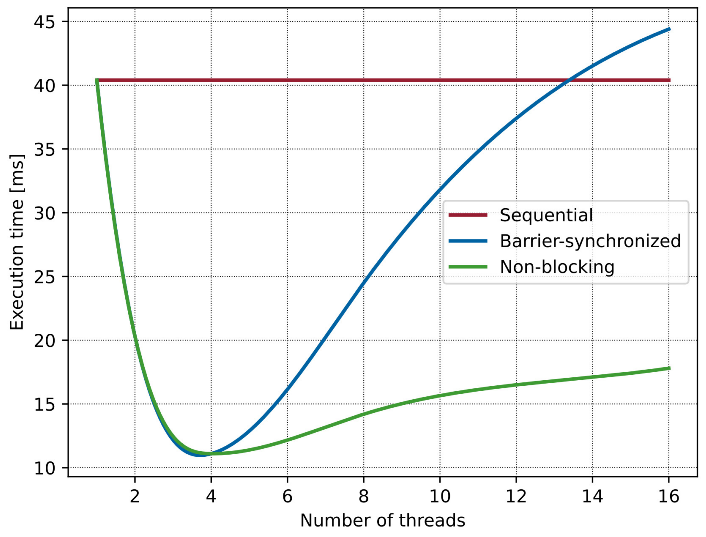

Figure 6.28: FFT performance robustness at 75% system stress rate.

Chapter 7

Road to Wait-free Barrier Synchronization

A problem well stated is a problem half-solved.

Charles Kettering

A practical definition of *wait-freedom* is that all execution threads eventually make progress. We pose the question: can a barrier-synchronized parallel algorithm achieve this level of progression?

In the contributed non-blocking model, NBD^{2}BS, one delayed thread can at most block just another thread at a given computational stage. So, there is still no guarantee that each individual thread will make progress at each synchronization point.

What if it is possible to provide a thread with the ability to unblock itself? A task-stealing thread can do work from another thread and, ideally, steal a task that will unblock itself and continue execution beyond the barrier synchronization point.

7.1 A Task-stealing Multithreading Model

Task-stealing is often referred as a load balancing technique and it is used as a means for efficient parallelism. The premise of load balancing is to prevent that no processing unit is doing too much and no processing unit is doing too little. Lately, it has been a primary focus of attention since researchers look for more ways to efficiently scale parallel applications [6, 17, 51, 54, 65, 72, 97]. This is also possible thanks to the work in concurrent data structures, particularly queues [3, 9, 10, 26, 33, 68, 93, 94], in order to provide fast thread communication to allow task-stealing.

Our model allows a thread to steal a *pending* task from another thread while waiting for synchronization, similar to termination detection barriers [44]. For example, whenever a thread reaches a *barrier synchronization* or waits for a given *data post*, it inspects the execution queues of the rest of the threads and executes pending tasks if any.

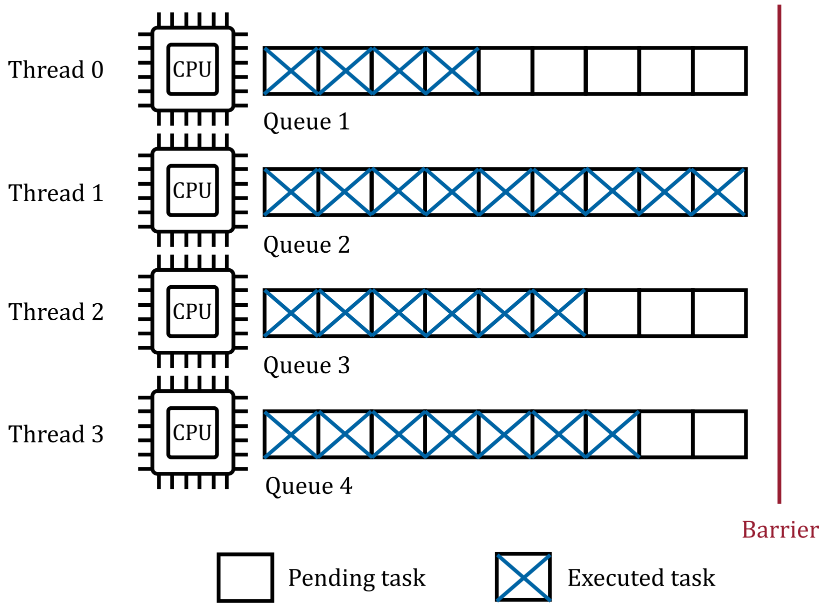

Figure 7.1: Task-stealing model.

Figure 7.1 helps to illustrate our task-stealing model explained below:

- Each thread has its own execution queue and each queue is visible to all threads.

- Each thread produces its own tasks at the beginning of each algorithmic stage, but each queue can be consumed by any thread that is waiting for synchronization.
- Threads steal tasks only from other threads that are in the same or in a previous algorithmic stage. For barrier-synchronized algorithms, it occurs naturally. Whereas for NBD^{2}BS algorithms, it needs to be enforced in order to protect the integrity of the data dependency flow.

Notice that figure 7.1 corresponds only to barrier synchronization semantics, where all threads are executing their own tasks and only *thread 1* is able to steal tasks from other threads while waiting at the barrier. However, both, task-stealing barrier-synchronized algorithms and task-stealing NBD^{2}BS algorithms have different thread progression guarantees.

Lemma 7.1. *All task-stealing barrier-synchronized parallel algorithms are obstruction-free.*

Proof. By contradiction. Assume, without loss of generality, one thread gets stuck before reaching the barrier at a given computational stage. By the obstruction-freedom definition, eventually this thread will run in isolation for a sufficient duration. Even if it happens that other threads stole some or all of its tasks, it will reach the barrier and will allow all threads to make progress. □

Lemma 7.2. *All task-stealing NBD^{2}BS parallel algorithms are wait-free.*

Proof. By contradiction. Assume, without loss of generality, one thread gets stuck before reaching the barrier at a given computational stage, but it was able to produce its tasks. Following from the NBD^{2}BS definition, this thread can block at most one different thread. However, this other thread can steal the task that is currently blocking it. Hence, each thread now is able to make progress regardless of the delays of others. □

It is important to point out that algorithms with stealing-barriers cannot guarantee stronger progression guarantees since system-wide progress still depends on all threads reaching the barrier. On the other hand, in order for NBD^{2}BS algorithms to be wait-free, each thread needs to produce their tasks at the beginning of each algorithmic stage.

7.2 Case Study: Can Bitonicsort Achieve Wait-freedom?

Two more ways to parallelize bitonicsort are presented. First, the common parallelization using a barrier synchronization, but every barrier is now considered a *stealing-barrier* where each thread steal tasks while waiting at the barrier. Second, the NBD^{2}BS multithreaded implementation of bitonicsort where each thread can steal tasks when waiting for a particular segment of data to be posted in the thread current execution stage. Finally, we present the performance analysis of our task-stealing model.

7.2.1 Stealing-barrier Parallel Bitonicsort

The algorithm is the same as the one presented in section 6.1.1.1, but adding the task-stealing capabilities to each of the execution threads. Figure 7.2 helps to illustrate the idea. The input sequence is partitioned into smaller segments of arbitrary size and assigned to a particular execution thread.

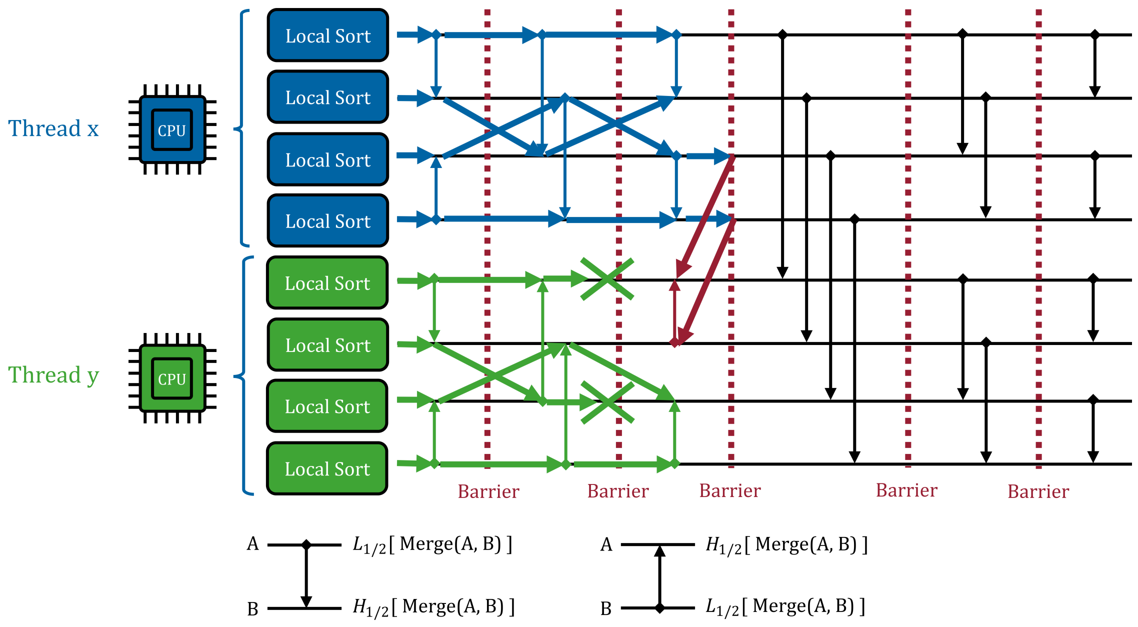

Figure 7.2: Stealing-barrier parallel bitonicsort example.

All threads produce their own tasks like a *local sort* or a *merge operation* between two segments at the beginning of each stage, consume its own tasks, and, finally, wait for the rest of the threads at each barrier synchronization point. However, if a given thread gets delayed, another thread can consume the delayed thread tasks while waiting at the barrier.

From figure 7.2, the blue thread steals a task from the green thread since it is already waiting at the barrier. Note that the blue thread cannot continue execution even though it steals a task, this is because the green thread has not reached the barrier yet. Algorithm 8 shows the high-level implementation.

Algorithm 8 Stealing-barrier bitonicsort implementation.

```
 1: procedure StealingBarrierThread(&barrier, &queues[number_of_threads])
 2:     for segment : assigned_segments
 3:         queues[thread_id].push({ sort(&segment) })
 4:     queues[thread_id].execute_all()
 5:     steal_and_wait(&barrier, &queues)        ▷ Try to steal while waiting for all threads
 6:
 7:     for each stage
 8:         for segment_1, segment_2 : assigned_segments
 9:             queues[thread_id].push({ merge(&segment_1, &segment_2) })
10:         queues[thread_id].execute_all()
11:         steal_and_wait(&barrier, &queues)    ▷ Try to steal while waiting for all threads
```

It is important to notice that some changes with respect to the original barrier-based implementation are those in lines 5 and 11. The behavior of the barrier differs, instead of just yielding the CPU to another potential thread waiting to be scheduled, a thread will check all the execution queues of the rest of the threads and it will try to steal a pending task to try to speed up the overall execution time.

Note that, as per our task-stealing model, each execution queue is produced by their owner threads (lines 3 and 9), but it can be consumed by any other thread if a task were to be stolen (lines 5 and 11). Hence, the implementation of each queue should be thread-safe and should handle its concurrency aspect internally.

7.2.2 Wait-free Parallel Bitonicsort

The wait-free parallel bitonicsort refers to the NBD^{2}BS implementation in section 6.1.1.2, but each execution thread now has the ability to steal tasks from any other thread. A blocked thread (potential victim) can give up tasks to its data-dependent counterpart thread (potential stealer). The stealer thread will unblock itself as the data, from a stolen task, will be now ready for upcoming computational stages and the victim thread will eventually make progress to the next stage since its tasks could have been already executed by other threads.

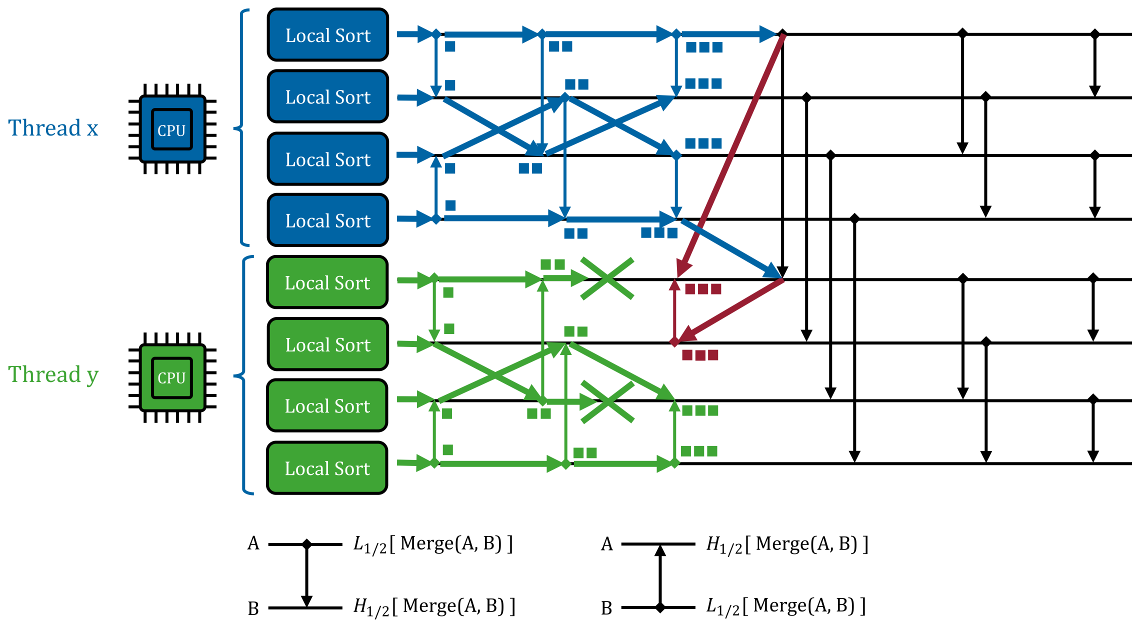

Figure 7.3: Wait-free parallel bitonicsort example.

Figure 7.3 illustrates how the wait-free parallel bitonicsort works. At some point, the blue thread gets stuck because of the data dependency that exhibits with the green thread. As opposed to the NBD^{2}BS implementation, the blue thread has the ability to unblock itself by stealing the missing merge operation that the green thread has left as a result of a delay. Then, the blue thread will post those data segments for the next algorithmic stage and both threads will be able to continue their execution.

Algorithm 9 shows the high-level implementation. Notice how it differs from the previous non-blocking bitonicsort in lines 16 and 18, where a thread will try to steal a task while it waits for a specific segment. The stealing approach is the same as in the barrier-based counterpart, where a thread will check all other execution queues to see if it can help execute other tasks. If that is the case, once the thread finish stealing, it does not need to wait any more since it would have executed the task that was blocking it from making progress.

Algorithm 9 Wait-free bitonicsort implementation.

```
 1: procedure WaitFreeThread(atomic &segment_stage[number_of_segments],
 2:                                   &queues[number_of_threads])
 3:     my_stage = 0
 4:
 5:     for segment : assigned_segments
 6:         queues[thread_id].push({
 7:             sort(&segment)
 8:             fetch_and_add(&segment_stage[segment], 1)      ▷ Post segment for next stage
 9:         })
10:     queues[thread_id].execute_all()
11:     my_stage++
12:
13:     for each stage
14:         for segment_1, segment_2 : assigned_segments
15:             while (my_stage != load(&segment_stage[segment_1]))     ▷ Wait for segment
16:                 steal()                                              ▷ Try to steal
17:             while (my_stage != load(&segment_stage[segment_2]))     ▷ Wait for segment
18:                 steal()                                              ▷ Try to steal
19:
20:             queues[thread_id].push({
21:                 merge(&segment_1, &segment_2)
22:                 fetch_and_add(&segment_stage[segment_1], 1)    ▷ Post seg. for next stage
23:                 fetch_and_add(&segment_stage[segment_2], 1)    ▷ Post seg. for next stage
24:             })
25:         queues[thread_id].execute_all()
26:         my_stage++
```

Again, it is important to point out that the execution queue of each thread internally manages its concurrency aspect and the algorithm does not show how it is handled. It is assumed that all threads correctly produce and consume tasks from the different execution queues.

7.2.3 Performance Analysis

The experimental setup is the same as in the previous bitonicsort section 6.1.1.3. Figure 7.4 shows the scalability of our non-blocking bitonicsort based on our NBD^{2}BS model, the parallel implementation of bitonicsort with a stealing-barrier (which is the central counter barrier with task-stealing), and the wait-free parallel bitonicsort (which is the non-blocking bitonicsort with task-stealing). Also, it is important to point out that each execution queue that is used handles concurrency in the simplest possible form. This is using a global lock that a thread needs to acquire in order to produce or consume a task.

Under ideal circumstances, where all CPUs are available to run all execution threads, our task-stealing model introduces a slight overhead; but both new task-stealing implementations scale as the input size increases.

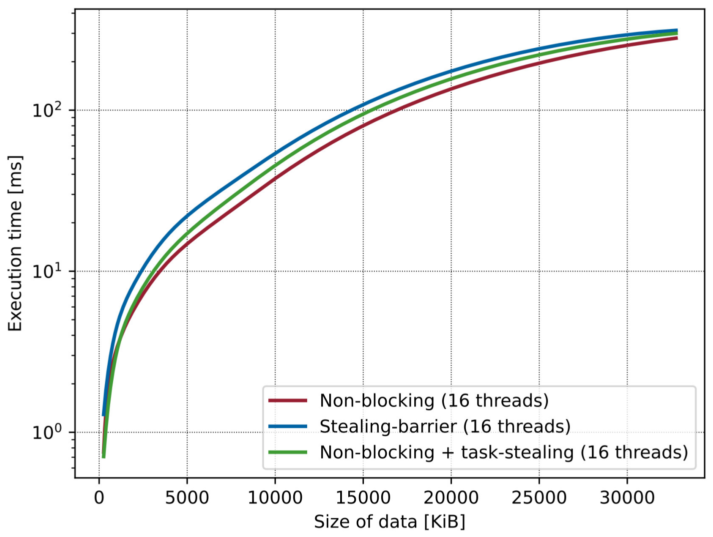

Figure 7.4: Task-stealing bitonicsort performance behavior in terms of input size.

From now on, the experiments sorts 1024 KiB of data, where each element is a 32 bit integer. The data is partitioned into 256 segments of 4 KiB each, resulting in a sorting network of 36 stages.

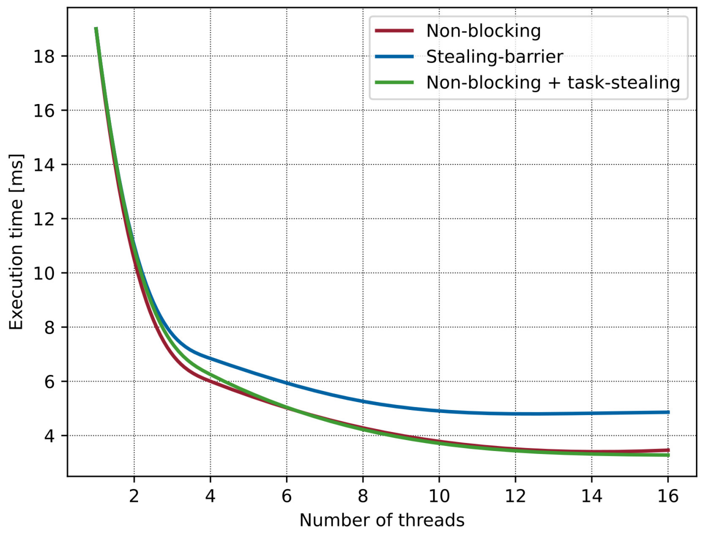

Figure 7.5: Task-stealing bitonicsort performance behavior in terms of number of threads.

Continuing with our ideal scenario, figure 7.5 shows how the same algorithms scale while modifying the number of execution threads. It is possible to see how the stealing-barrier introduces more overhead than the non-blocking with task-stealing bitonicsort. This is mainly because, in ideal circumstances, all threads may arrive at similar times at the barrier; but all of them check if they can steal a task to see if they can help with the execution of other threads.

On the other hand, figure 7.6 shows the behavior under stressed circumstances. Note that the implementations with task-stealing techniques, both stealing-barrier and NBD^{2}BS plus task-stealing, excel under low system stress rates.

The overall execution time may benefit from our task-stealing model. However, in scenarios with high system stress rates, both task-stealing implementations suffer big performance cliffs and they do not degrade gracefully as our original NBD^{2}BS bitonicsort implementation. The reason behind this is that each execution queue also becomes a hot-spot, where all execution threads are now contending to gain access to the multiple execution queues.

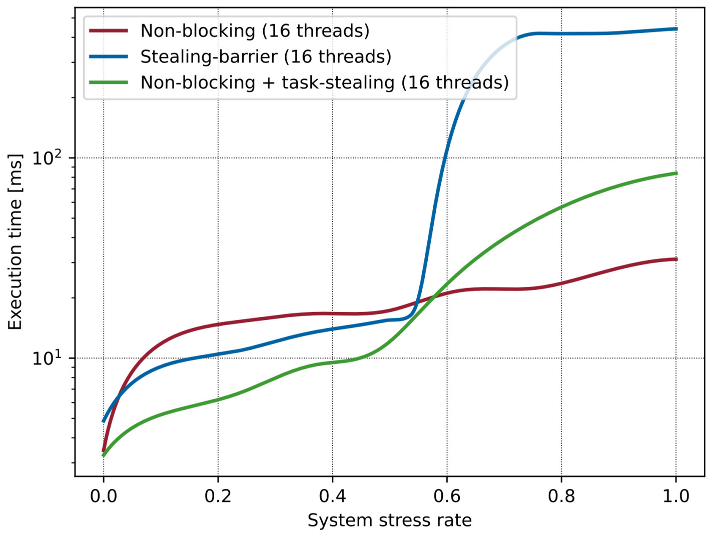

Figure 7.6: Task-stealing bitonicsort performance behavior in terms of system stress rate.

Figures 7.7 and 7.8 show the behavior of these algorithms at specific stress rate scenarios. First, when 8 out of 16 CPUs are already busy (50% system stress rate), figure 7.7, our novel NBD^{2}BS bitonicsort with task-stealing capabilities exhibits better performance robustness when using more threads.

Then, at 75% system stress rate (12 out of 16 CPUs are busy), figure 7.8 shows how the implementations with task-stealing queues degrade faster than our original contribution, the NBD^{2}BS bitonicsort. It is not hard to imagine a scenario where a thread is not only waiting for other threads or for a specific segment, it is also now waiting for a preempted thread to release the acquired lock of a particular execution queue.

As this model evolves, it is important to explain that part of the work to be performed towards wait-free barrier synchronization is not limited to task-stealing techniques only (as these experiments showed). High performance concurrent queues/deques, like lock-free and wait-free, are vital in order to scale this model and make it more robust against the over-subscribed nature of modern multicore systems.

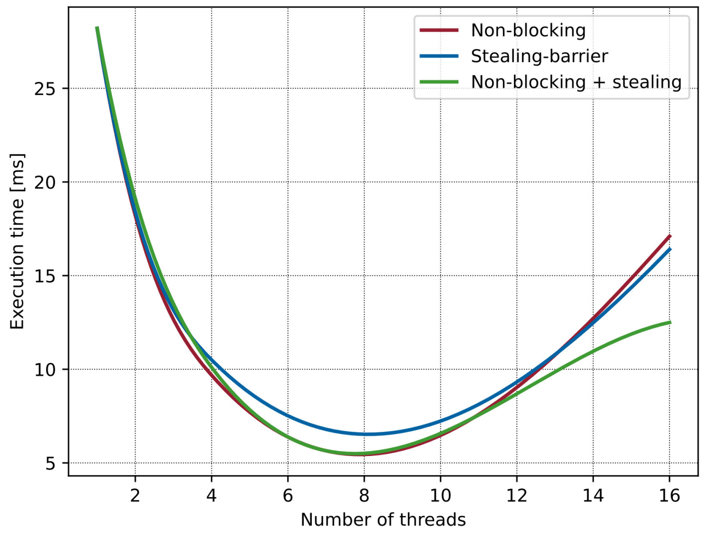

Figure 7.7: Task-stealing bitonicsort performance robustness at 50% system stress rate.

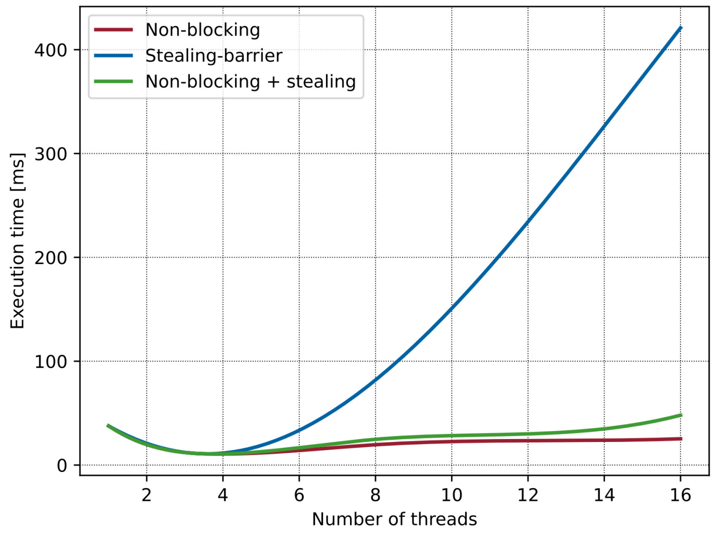

Figure 7.8: Task-stealing bitonicsort performance robustness at 75% system stress rate.

7.2.4 Wait-free Bitonicsort and Multiway Mergesort Comparative Analysis

This section presents a comparative analysis of our non-blocking bitonicsort with task-stealing capabilities and the state-of-the-art parallel multiway mergesort (MWMS) that was described in section 6.1.1.4. The experimental setup is, once again, similar to all previous experiments. This is, sorting 1024 KiB of data, where each element is a 32 bit integer. For bitonicsort, the data is partitioned into 256 segments of 4 KiB each, resulting in a sorting network of 36 stages.

As seen before, parallel multiway mergesort performs better in ideal circumstances, see figure 7.9. However, in situations where the system is already running other tasks, wait-free bitonicsort (which is the non-blocking implementation based on our NBD^{2}BS model plus task-stealing techniques) performs better. From figure 7.10, note that in scenarios where the system stress rate is less than 50%, our wait-free bitonicsort implementation is at least a factor of 2 faster than multiway mergesort. Task-stealing techniques can help boost the performance of parallel applications, notice the performance differences even in low system stress rates. The experiment is set up to run using the maximum available hardware (16 threads), but the performance of multiway mergesort degrades significantly even when only 2 out of 16 CPU cores are busy running other tasks.

It is important to notice from figure 7.10 that in high-load situations, where the system stress rate is greater than 50%, the wait-free bitonicsort implementation suffers from a big performance cliff due to high-level contention in each thread execution queue. This discontinuity in performance is similar to the one in multiway mergesort, suggesting that the context-switching becomes an important bottleneck in parallel applications.

In the end, the results are promising, they show that it is possible to design parallel algorithms that are robust to the high context-switch rate nature of modern multicore systems.

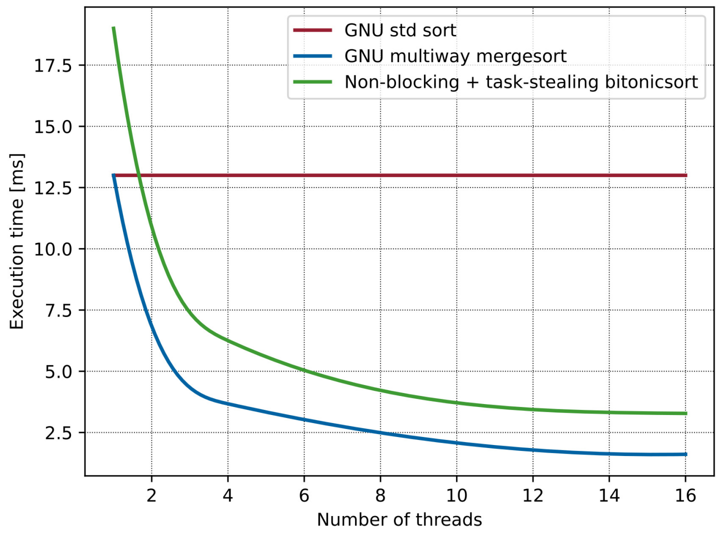

Figure 7.9: Wait-free Bitonicsort and Multiway Mergesort varying the number of threads.

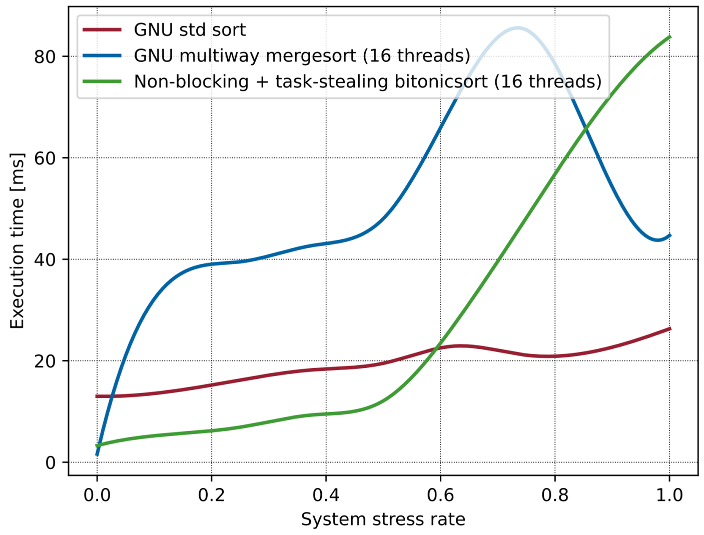

Figure 7.10: Wait-free Bitonicsort and Multiway Mergesort varying the system stress rate.

Chapter 8

Concluding Remarks

The proper method for inquiring after the properties of things is to deduce them from experiments.

Isaac Newton

Modern multicore CPUs present challenges when it comes to efficiently exploit the underlying parallelism that the hardware provides. These systems are general purpose processors that run not only one, but a large number of tasks and each CPU core becomes a resource that is shared among all those tasks that are just waiting to be executed.

The oversubscribed nature of multicore systems impacts the performance of a given task. Furthermore, tasks that are parallelized in multiple execution threads can also be severely impacted. The most common form of parallelism in multicore computing is the Single Program Multiple Data (SPMD) model, where multiple threads are executed simultaneously over disjoint subsets of data. Additionally, these threads are working cooperatively to accomplish a task, it is likely that, at defined moments in time, they need to communicate with each other. This particular moment in time, where threads perform the necessary data realignments and repartitionings due to data dependencies, is known as barrier synchronization. In other words, a thread must not continue its execution if it needs the computational results that are being produced by other threads. It is not hard to imagine a scenario where the expected parallelism significantly reduces if even just one thread, out of a group of N threads, is preempted from its CPU. The rest of the threads still have to wait in a barrier synchronization point until the preempted thread is assigned back to a CPU core to finish its respective computations and reaches the barrier.

This dissertation addressed the problem mentioned above with a novel model that we called *non-blocking data-driven barrier synchronization* or NBD^{2}BS, for short. The main idea is to exploit the memory access pattern of an algorithm to provide better thread progression guarantees, preemption robustness, and, thus, performance robustness; so that the algorithm degrades gracefully in its performance whenever the system is under high load/stress. The experimental results are promising, they show that an NBD^{2}BS parallel algorithm exhibits better performance in scenarios with high system stress rates by almost a factor of 5 than its barrier-synchronized counterpart.

Table 8.1 is almost identical to the one presented in section 4.3. However, there is an additional row: this is our contribution and now it is part of the work that has been done in this research field.

	Pros	**Cons**
Counter barriers	Practical	Blocking
Tree-based barriers	Low contention	Blocking
Dissemination barriers	Low contention	Blocking
Tournament barriers	Low contention	Blocking
Multiple disjoint barriers	Non-blocking	Complex control flow
Fuzzy barriers	Non-blocking	Application specific
Task-stealing barriers	Non-blocking	Complex runtime support
Speculative barriers	Non-blocking	Tradeoff: accuracy vs speed
NBD^{2}BS	Non-blocking	$\Theta(n)$ additional space

Table 8.1: New summary of related work in barrier synchronization.

The contributed model was successfully applied to a variety of algorithms, specially in the field of high performance sorting and signal processing due to the importance of these core computational tasks, which can serve and be attractive to a wide range of applications and academic research fields.

There is important work yet to be explored. The dissertation aims to motivate additional efforts in the future. For example, introducing the problem from a linear optimization point of view, where the number of threads can be adjusted depending on the current level of workload in the system to find optimal performance. Another example is to establish a probabilistic model on how to characterize the impact in performance of parallel multithreaded algorithms with different scheduler properties for CPU allocation. This could be particularly useful due to the non-deterministic nature of multithreaded algorithms working concurrently.

Finally, this dissertation also opens new research possibilities by introducing a model towards *wait-free* barrier synchronization in chapter 7. In other words, the overall task *always* makes progress regardless of any unexpected delays of individual execution threads. The reasoning behind is that, even if *N-1* threads (out of a total of *N* threads) are preempted, one thread can still perform the computational tasks of the rest of the threads by exploiting task-stealing techniques. Our experimental results are encouraging and motivate research towards more efficient methods.

Bibliography

[1] J. L. Abellán, J. Fernández, and M. E. Acacio. A novel hardware-based barrier synchronization for many-core cmps. In *Proceedings of the 7th ACM International Conference on Computing Frontiers*, volume 1. 2010.

[2] J. L. Abellán, J. Fernández, and M. E. Acacio. Efficient hardware barrier synchronization in many-core cmps. *IEEE Transactions on Parallel and Distributed Systems*, 23(8):1453–1466, 2011.

[3] D. Adas and R. Friedman. A fast wait-free multi-producers single-consumer queue. In *23rd International Conference on Distributed Computing and Networking*, pages 77–86, 2022.

[4] A. Aiken and D. Gay. Barrier inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 342–354, 1998.

[5] S. G. Akl. *The design and analysis of parallel algorithms.* Prentice-Hall, Inc., 1989.

[6] E. Aksenova, E. Barkovsky, and A. Sokolov. The models and methods of optimal control of three work-stealing deques located in a shared memory. *Lobachevskii Journal of Mathematics*, 40:1763–1770, 2019.

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

[8] N. S. Arenstorf and H. F. Jordan. Comparing barrier algorithms. *Parallel Computing*, 12(2):157–170, 1989.

[9] S. Arnautov, P. Felber, C. Fetzer, and B. Trach. Ffq: A fast single-producer/multiple-consumer concurrent fifo queue. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 907–916. IEEE, 2017.

[10] A. Barrington, S. Feldman, and D. Dechev. A scalable multi-producer multi-consumer wait-free ring buffer. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 1321–1328, 2015.

[11] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307–314, 1968.

[12] R. N. Bracewell and R. N. Bracewell. *The Fourier transform and its applications*, volume 31999. McGraw-Hill New York, 1986.

[13] E. D. Brooks. The butterfly barrier. *International Journal of Parallel Programming*, 15:295–307, 1986.

[14] D. Callahan, K. Kennedy, and J. Subhlok. Analysis of event synchronization in a parallel programming tool. In *Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 21–30, 1990.

[15] C. Calvin. Implementation of parallel fft algorithms on distributed memory machines with a minimum overhead of communication. *Parallel Computing*, 22(9):1255–1279, 1996.

[16] C. Calvin and F. Desprez. Minimizing communication overhead using pipelining for multi-dimensional fft on distributed memory machines. In *PARCO*, pages 65–72, 1993.

[17] D. Chase and Y. Lev. Dynamic circular work-stealing deque. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 21–28, 2005.

[18] M. Cintra, J. F. Martínez, and J. Torrellas. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *Proceedings of the 27th annual international symposium on Computer architecture*, pages 13–24, 2000.

[19] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

[20] F. Darema. The spmd model: Past, present and future. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 8th European PVM/MPI Users' Group Meeting Santorini/Thera, Greece, September 23–26, 2001 Proceedings 8*, pages 1–1. Springer, 2001.

[21] F. Darema. Applications environment for the ibm research parallel processor prototype (rp3). In *Supercomputing: 1st International Conference Athens, Greece, June 8–12, 1987 Proceedings*, pages 80–95. Springer, 2005.

[22] F. Darema, D. A. George, V. A. Norton, and G. F. Pfister. A single-program-multiple-data computational model for epex/fortran. *Parallel Computing*, 7(1):11–24, 1988.

[23] Z. Deng, J. Li, and J. Lin. A synchronization optimization technique for openmp. In *2021 IEEE 13th International Conference on Computer Research and Development (ICCRD)*, pages 95–103. IEEE, 2021.

[24] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha. Scalable work stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, 2009.

[25] M. N. Farooqi and M. Pericàs. Vectorized barrier and reduction in llvm openmp runtime. In *OpenMP: Enabling Massive Node-Level Parallelism: 17th International Workshop on OpenMP, IWOMP 2021*, pages 14–16. Cham: Springer International Publishing, 2021.

[26] S. Feldman and D. Dechev. A wait-free multi-producer multi-consumer ring buffer. *ACM SIGAPP Applied Computing Review*, 15(3):59–71, 2015.

[27] J. Feliu, A. Ros, M. E. Acacio, and S. Kaxiras. Speculative inter-thread store-to-load forwarding in smt architectures. *Journal of Parallel and Distributed Computing*, 173:94–106, 2023.

[28] J. A. Fessler. Model-based image reconstruction for mri. *IEEE signal processing magazine*, 27(4):81–89, 2010.

[29] C. H. Flood, D. Detlefs, N. Shavit, and X. Zhang. Parallel garbage collection for shared memory multiprocessors. In *Java Virtual Machine Research and Technology Symposium*, 2001.

[30] M. J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966.

[31] W. Gao, J. Fang, C. Huang, C. Xu, and Z. Wang. Optimizing barrier synchronization on armv8 many-core architectures. In *2021 IEEE International Conference on Cluster Computing (Cluster)*, pages 542–552. IEEE, 2021.

[32] A. Garza, C. A. Parra, and I. D. Scherson. Non-blocking technique for parallel algorithms with global barrier synchronization. In *2021 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 1759–1764. IEEE, 2021.

[33] O. Giersch and J. Nolte. Fast and portable concurrent fifo queues with deterministic memory reclamation. *IEEE Transactions on Parallel and Distributed Systems*, 33(3):604–616, 2021.

[34] S. Gopal, T. Vijaykumar, J. E. Smith, and G. S. Sohi. Speculative versioning cache. In *Proceedings 1998 Fourth International Symposium on High-Performance Computer Architecture*, pages 195–205. IEEE, 1998.

[35] R. Gupta. The fuzzy barrier: a mechanism for high speed synchronization of processors. *ACM SIGARCH Computer Architecture News*, 17(2):54–63, 1989.

[36] J. L. Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[37] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. *ACM SIGOPS Operating Systems Review*, 32(5):58–69, 1998.

[38] D. Hensgen, R. Finkel, and U. Manber. Two algorithms for barrier synchronization. *International Journal of Parallel Programming*, 17:1–17, 1988.

[39] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

[40] M. Herlihy. The multicore revolution: the challenges for theory. In *FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science: 27th International Conference, New Delhi, India, December 12-14, 2007. Proceedings 27*, pages 1–8. Springer, 2007.

[41] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *23rd International Conference on Distributed Computing Systems, 2003. Proceedings.*, pages 522–529. IEEE, 2003.

[42] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, 1993.

[43] M. Herlihy and N. Shavit. On the nature of progress. In *Principles of Distributed Systems: 15th International Conference, OPODIS 2011, Toulouse, France, December 13-16, 2011. Proceedings 15*, pages 313–328. Springer, 2011.

[44] M. Herlihy, N. Shavit, V. Luchangco, and M. Spear. *The art of multiprocessor programming*. Newnes, 2020.

[45] C. Hetland, G. Tziantzioulis, B. Suchy, M. Leonard, J. Han, J. Albers, N. Hardavellas, and P. Dinda. Paths to fast barrier synchronization on the node. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, pages 109–120. 2019.

[46] T. Highlander and A. Rodriguez. Very efficient training of convolutional neural networks using fast fourier transform and overlap-and-add. *arXiv preprint arXiv:1601.06815*, 2016.

[47] J. M. Hill and D. B. Skillicorn. Practical barrier synchronisation. In *Proceedings of the Sixth Euromicro Workshop on Parallel and Distributed Processing*, pages 438–444. IEEE, 1998.

[48] T. Hoefler, P. Gottschling, and A. Lumsdaine. Leveraging non-blocking collective communication in high-performance applications. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 113–115, 2008.

[49] G. Hopper and R. Adhami. An fft-based speech recognition system. *Journal of the Franklin Institute*, 329(3):555–562, 1992.

[50] A. Huang. Moore's law is dying (and that could be good. *IEEE Spectrum*, 52(4):43–47, 2015.

[51] J. H. M. Korndörfer, A. Eleliemy, A. Mohammed, and F. M. Ciorba. Lb4omp: A dynamic load balancing library for multithreaded applications. *IEEE Transactions on Parallel and Distributed Systems*, 33(4):830–841, 2021.

[52] A. Krishnamurthy and K. Yelick. Optimizing parallel programs with explicit synchronization. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 196–204, 1995.

[53] V. Krishnan and J. Torrellas. A chip-multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers*, 48(9):866–880, 1999.

[54] V. Kumar, S. M. Blackburn, and D. Grove. Friendly barriers: Efficient work-stealing with return barriers. *ACM SIGPLAN Notices*, 49(7):165–176, 2014.

[55] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications*, 1978.

[56] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 9:690–691, 1979.

[57] N. M. Lê, A. Pop, A. Cohen, and F. Zappa Nardelli. Correct and efficient work-stealing for weak memory models. *ACM SIGPLAN Notices*, 48(8):69–80, 2013.

[58] C. E. Leiserson and I. B. Mirman. How to survive the multicore software revolution (or at least survive the hype). *Cilk Arts*, 1:11, 2008.

[59] B. Liu, H. Liang, J. Wu, X. Chen, P. Liu, and Y. Han. Accelerating convolutional neural networks in frequency domain via kernel-sharing approach. In *Proceedings of the 28th Asia and South Pacific Design Automation Conference*, pages 733–738, 2023.

[60] W. Liu, Q. Liao, F. Qiao, W. Xia, C. Wang, and F. Lombardi. Approximate designs for fast fourier transform (fft) with application to speech recognition. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 66(12):4727–4739, 2019.

[61] B. D. Lubachevsky. Synchronization barrier and related tools for shared memory parallel programming. *International Journal of Parallel Programming*, 19:225–250, 1990.

[62] R. G. Lyons. Program aids analysis of fft algorithms. *EDN Magazine, Aug*, 6, 1987.

[63] P. Marcuello and A. Gonzalez. Clustered speculative multithreaded processors. In *Proceedings of the 13th International Conference on Supercomputing*, pages 365–372, 1999.

[64] J. F. Martinez and J. Torrellas. Speculative synchronization: Applying thread-level speculation to explicitly parallel applications. *ACM SIGOPS Operating Systems Review*, 36(5):18–29, 2002.

[65] S. McClure, A. Ousterhout, S. Shenker, and S. Ratnasamy. Efficient scheduling policies for {Microsecond-Scale} tasks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1–18, 2022.

[66] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, 1991.

[67] J. M. Mellor-Crummey and M. L. Scott. Synchronization without contention. *ACM SIGPLAN Notices*, 26(4):269–278, 1991.

[68] X. Meng, X. Zeng, X. Chen, and X. Ye. A cache-friendly concurrent lock-free queue for efficient inter-core communication. In *2017 IEEE 9th International Conference on Communication Software and Networks (ICCSN)*, pages 538–542. IEEE, 2017.

[69] A. K. Mohamed El Maarouf, L. Giraud, A. Guermouche, and T. Guignon. Combining reduction with synchronization barrier on multi-core processors. *Concurrency and Computation: Practice and Experience*, page e7402, 2023.

[70] R. C. Nanjegowda, O. R. Hernandez, B. M. Chapman, and H. Jin. Scalability evaluation of barrier algorithms for openmp. In *Proceedings of the 5th International Workshop on OpenMP, IWOMP 2009*, pages 42–52. 2009.

[71] A. Nicolau, G. Li, and A. Kejariwal. Techniques for efficient placement of synchronization primitives. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 199–208, 2009.

[72] P. Nookala, P. Dinda, K. C. Hale, K. Chard, and I. Raicu. Enabling extremely fine-grained parallelism via scalable concurrent queues on modern many-core architectures. In *2021 29th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 1–8. IEEE, 2021.

[73] A. Osterhaug. *Guide to parallel programming on sequent computer systems.* Prentice-Hall, Inc., 1989.

[74] L. Otten. LaTeX template for thesis and dissertation documents at UC Irvine. Available at `https://github.com/lotten/uci-thesis-latex`, 2012.

[75] M. Pedrero, R. Quislant, E. Gutierrez, E. L. Zapata, and O. Plata. Speculative barriers with transactional memory. *IEEE Transactions on Computers*, 71(1):197–208, 2020.

[76] G. F. Pfister and V. Norton. "hot spot" contention and combining in multistage interconnection networks. *IEEE Transactions on Computers*, 100:10, 1985.

[77] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings. 34th ACM/IEEE International Symposium on Microarchitecture. MICRO-34*, pages 294–305. IEEE, 2001.

[78] A. Rodchenko, A. Nisbet, A. Pop, and M. Luján. Effective barrier synchronization on intel xeon phi coprocessor. In *Euro-Par 2015: Parallel Processing: 21st International Conference on Parallel and Distributed Computing, Vienna, Austria, August 24-28, 2015*, pages 588–600. Springer Berlin Heidelberg, 2015.

[79] J. Sánchez-Curto and P. Chamorro-Posada. The inherent overlapping in the parallel calculation of the laplacian. *Journal of Computational Science*, page 101945, 2023.

[80] P. Sanders. Fast priority queues for cached memory. *Journal of Experimental Algorithmics (JEA)*, 5:7–es, 2000.

[81] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, 1995.

[82] J. Singler and P. Sanders. The gnu libstdc++ parallel mode: Benefit from multi-core using the stl.

[83] J. Sorensen and S. B. Baden. Hiding communication latency with non-spmd, graph-based execution. In *Computational Science–ICCS 2009: 9th International Conference Baton Rouge, LA, USA, May 25-27, 2009 Proceedings, Part I 9*, pages 155–164. Springer, 2009.

[84] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. *ACM SIGARCH Computer Architecture News*, 28(2):1–12, 2000.

[85] T. Sumanaweera and D. Liu. Medical image reconstruction with the fft. *GPU gems*, 2:765–784, 2005.

[86] L. Szustak. Strategy for data-flow synchronizations in stencil parallel computations on multi-/manycore systems. *The Journal of Supercomputing*, 74(4):1534–1546, 2018.

[87] G. Taubenfeld. *Synchronization algorithms and concurrent programming.* Pearson Education, 2006.

[88] The kernel development community. Linux Complete Fair Scheduler (CFS). Available at `https://docs.kernel.org/scheduler/sched-design-CFS.html` (2023).

[89] The kernel development community. Linux stress tool. Available at `https://linux.die.net/man/1/stress` (2023).

[90] The Open MPI Development Team. Open MPI: Open Source High Performance Computing. Available at `https://www.open-mpi.org/` (2023).

[91] The OpenMP Architecture Review Board. The OpenMP API specification for parallel programming. Available at `https://www.openmp.org/` (2023).

[92] N. Vasilache, J. Johnson, M. Mathieu, S. Chintala, S. Piantino, and Y. LeCun. Fast convolutional nets with fbfft: A gpu performance evaluation. *arXiv preprint arXiv:1412.7580*, 2014.

[93] J. Wang, Q. Jin, X. Fu, Y. Li, and P. Shi. Accelerating wait-free algorithms: Pragmatic solutions on cache-coherent multicore architectures. *IEEE Access*, 7:74653–74669, 2019.

[94] J. Wang, Y. Tian, and X. Fu. Equeue: Elastic lock-free fifo queue for core-to-core communication on multi-core processors. *IEEE Access*, 8:98729–98741, 2020.

[95] Y. Wang, Y. Tang, Y. Jiang, J.-G. Chung, S.-S. Song, and M.-S. Lim. Novel memory reference reduction methods for fft implementations on dsp processors. *IEEE Transactions on signal processing*, 55(5):2338–2349, 2007.

[96] Z. Wang, Y. Zhao, and J. Chen. Multi-scale fast fourier transform based attention network for remote sensing image super-resolution. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 2023.

[97] K. Yamashita and T. Kamada. Introducing a multithread and multistage mechanism for the global load balancing library of x10. *Journal of Information Processing*, 24(2):416–424, 2016.

[98] P.-C. Yew and N.-F. Tzeng. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Transactions on Computers*, 100(4):388–395, 1987.

[99] G. Zhang, F. Martínez, A. Tal, and B. Blainey. Busy-wait barrier synchronization using distributed counters with local sensor. In *OpenMP Shared Memory Parallel Programming: International Workshop on OpenMP Applications and Tools, WOMPAT 2003 Toronto, Canada*, pages 84–98. Springer Berlin Heidelberg, 2003.

[100] Y. Zhang and X. Li. Fast convolutional neural networks with fine-grained ffts. In *Proceedings of the ACM international conference on parallel architectures and compilation techniques*, pages 255–265, 2020.