NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF ⟨COMPUTER SCIENCE⟩

RAFAEL GUERREIRO CUSTÓDIO

Bachelor in Computer Science

IMPLEMENTATION OF THE LOW-COST WORK STEALING ALGORITHM FOR PARALLEL COMPUTATIONS

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
December, 2022

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
⟨COMPUTER SCIENCE⟩

IMPLEMENTATION OF THE LOW-COST WORK STEALING ALGORITHM FOR PARALLEL COMPUTATIONS

RAFAEL GUERREIRO CUSTÓDIO

Bachelor in Computer Science

Adviser: Hervé Miguel Cordeiro Paulino
Associate Professor, NOVA University Lisbon

Examination Committee:

Chair: Doutor António Maria Lobo César Alarcão Ravara
Associate Professor, FCT-NOVA

Rapporteur: Doutor Luis Miguel Barros Lopes
Associate Professor, FCUP

Adviser: Hervé Miguel Cordeiro Paulino
Associate Professor, FCT-NOVA

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
December, 2022

Implementation of the Low-Cost Work Stealing Algorithm for parallel computations

Acknowledgements

This project was not made alone. Most likely none is. I would never even have been able to even begin this project if I had not learned from all the professors of the course. They spend their time and patience passing their knowledge to so many students and even go out of their way to give explanations off schedule. My thesis adviser Hervé Miguel Cordeiro Paulino, who was always patient with me and always gave me advice in the areas where I lacked it the most. I would also like to thank NOVA FCT for all the tools they provide to assist in our education.

The colleagues, friends, I met along the way, who I shared projects with and provided assistance every time I needed. Lastly, I would like to thank my family, who gave me the means to seek an higher education and who never fell short to help or motivate me.

*“Success is walking from failure to failure with no loss of enthusiasm.
” (Winston Churchill)*

Abstract

For quite a while, CPU's clock speed has stagnated while the number of cores keeps increasing. Because of this, parallel computing rose as a paradigm for programming on multi-core architectures, making it critical to control the costs of communication. Achieving this is hard, creating the need for tools that facilitate this task.

Work Stealing (WSteal) became a popular option for scheduling multithreaded computations. It ensures scalability and can achieve high performance by spreading work across processors. Each processor owns a double-ended queue where it stores its work. When such deque is empty, the processor becomes a thief, attempting to steal work, at random, from other processors' deques. This strategy was proved to be efficient and is still currently used in state-of-the-art WSteal algorithms. However, due to the concurrent nature of the deque, local operations require expensive memory fences to ensure correctness. This means that even when a processor is not stealing work from others, it still incurs excessive overhead due to the local accesses to the deque. Moreover, the pure receiver-initiated approach to load balancing, as well as, the randomness of the targeting of a victim makes it not suitable for scheduling computations with few or unbalanced parallelism.

In this thesis, we explore the various limitations of WSteal in addition to solutions proposed by related work. This is necessary to help decide on possible optimizations for the Low-Cost Work Stealing (LCWS) algorithm, proposed by Paulino and Rito, that we implemented in C++. This algorithm is proven to have exponentially less overhead than the state-of-the-art WSteal algorithms. Such implementation will be tested against the canonical WSteal and other variants that we implemented so that we can quantify the gains of the algorithm.

Keywords: Scheduling algorithms, Work stealing, Parallel computing, Load balancing, Low-Cost Work Stealing

Resumo

Já faz algum tempo desde que a velocidade dos CPUs tem vindo a estagnar enquanto o número de cores tem vindo a subir. Por causa disto, o ramo de computação paralela subiu como paradigma para programação em arquiteturas multi-core, tornando crítico controlar os custos associados de comunicação. No entanto, isto não é uma tarefa fácil, criando a necessidade de criar ferramentas que facilitem este controlo.

Work Stealing (WSteal) tornou-se uma opção popular para o escalonamento de computações concorrentes. Este garante escalabilidade e consegue alcançar alto desempenho por distribuir o trabalho por vários processadores. Cada processador possui uma fila duplamente terminada (deque) onde é guardado o trabalho. Quando este deque está vazio, o processador torna-se um ladrão, tentando roubar trabalho do deque de um outro processador, escolhido aleatoriamente. Esta estratégia foi provada como eficiente e ainda é atualmente usada em vários algoritmos WSteal. Contudo, devido à natureza concorrente do deque, operações locais requerem barreiras de memória, cujo correto funcionamento tem um alto custo associado. Além disso, a estratégia pura receiver-initiated (iniciada pelo recetor) de balanceamento de carga, assim como a aleatoriedade no processo de escolha de uma vitima faz com que o algoritmo não seja adequado para o scheduling de computações com pouco ou desequilibrado paralelismo.

Nesta tese, nós exploramos as várias limitações de WSteal, para além das soluções propostas por trabalhos relacionados. Isto é um passo necessário para ajudar a decidir possíveis otimisações para o algoritmo Low-Cost Work Stealing (LCWS), proposto por Paulino e Rito, que implementámos em C++. Este algoritmo está provado como tendo exponencialmente menos overhead que outros algoritmos de WSteal. Tal implementação será testada e comparada com o algoritmo canónico de WSteal, assim como outras suas variantes que implementámos para que possamos quantificar os ganhos do algoritmo.

Palavras-chave: Algoritmos de scheduling, Work stealing, Computações paralelas, Balanceamento de carga, Low-Cost Work Stealing

Contents

List of Figures

List of Tables

Acronyms

CAS Compare-And-Swap x, 7, 11, 14, 15, 23, 24, 25, 26, 28, 44, 46, 47, 53, 60, 77

DAG Directed Acyclic Graph 8, 16, 24

LCWS Low-Cost Work Stealing vi, vii, ix, x, xi, xii, 3, 4, 14, 16, 17, 19, 23, 24, 28, 29, 32, 34, 37, 41, 43, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 81, 88, 89, 90, 92, 93, 94, 96, 97, 99, 101, 103, 105, 107, 108, 109, 111, 112, 113, 115, 116, 117, 119, 120, 121, 123, 125, 127, 129, 131, 133, 135, 137, 139, 141, 143, 145, 147, 149, 151, 153, 155, 157, 159, 161, 163, 165, 167, 169, 171, 173, 175, 177, 179, 181, 183, 185, 187, 189, 191, 193, 195, 197, 199, 201, 203, 205, 207, 209, 211, 213, 215, 217, 219, 221, 223, 225, 227, 229, 231, 233, 235, 237, 239, 241, 243, 245, 247, 249, 251, 253, 255, 257, 259, 261, 263, 265, 267, 269, 271, 273, 275, 277, 279, 300

NBWS Non-Blocking Work Stealing viii, 4, 5, 11, 14, 25

PBBS Problem Based Benchmark Suite 4, 24, 26, 28, 43, 44

RWS Randomized Work Stealing 8, 9, 20

SBLCWS Signal-Based Low-Cost Work Stealing ix, x, xi, 29, 31, 32, 33, 34, 37, 41, 42, 49, 51, 53, 54, 55, 56, 57, 58, 59, 60, 61, 67, 69, 71, 73, 75, 77, 88, 89, 90, 91, 92, 93, 94, 95, 96, 98, 100, 102, 104, 106, 108, 109, 110, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 124, 126, 128, 130, 132, 134, 136, 138, 140, 142, 144, 146, 148, 150, 152, 154, 156, 158, 160, 162, 164, 166, 168, 170, 172, 174, 176, 178, 180, 182, 184, 186, 188, 190, 192, 194, 196, 198, 200, 202, 204, 206, 208, 210, 212, 214, 216, 218, 220, 222, 224, 226, 228, 230, 232, 234, 236, 238, 240, 242, 244, 246, 248, 250, 252, 254, 256, 258, 260, 262, 264, 266, 268, 270, 272, 274, 276, 278, 286, 293, 309, 318

TSO Total Store Ordering 15

WShare Work Sharing ix, x, xi, xii, 2, 3, 17, 18, 19, 37, 38, 41, 42, 43, 44, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 89, 90, 91, 93, 94, 95, 97, 98, 101, 102, 105, 106, 108, 109, 110, 112, 113, 114, 116, 117, 118, 120, 121, 122, 125, 126, 129, 130, 133, 134, 137, 138, 141, 142, 145, 146, 149, 150, 153, 154, 157, 158, 161, 162, 165, 166, 169, 170, 173, 174, 177, 178, 181, 182, 185, 186, 189, 190, 193, 194, 197, 198, 201, 202, 205, 206, 209, 210, 213, 214, 217, 218, 221, 222, 225, 226, 229, 230, 233, 234, 237, 238, 241, 242, 245, 246, 249, 250, 253, 254, 257, 258, 261, 262, 265, 266, 269, 270, 273, 274, 277, 278, 309, 318

WSteal Work Stealing vi, vii, ix, x, xi, xii, 2, 3, 4, 5, 7, 8, 9, 11, 12, 13, 14, 16, 17, 18, 19, 24, 28, 37, 41, 43, 46, 47, 48, 49, 50, 51, 52, 61, 77, 78, 79, 80, 81, 87, 91, 95, 99, 103, 107, 111, 114, 118, 123, 127, 131, 135, 139, 143, 147, 151, 155, 159, 163, 167, 171, 175, 179, 183, 187, 191, 195, 199, 203, 207, 211, 215, 219, 223, 227, 231, 235, 239, 243, 247, 251, 255, 259, 263, 267, 271, 275

1

Introduction

1.1 A Parallel Computation

Nowadays, the processor's number of cores keeps on growing and software needs to adapt to harness these untapped resources. The rise of fine-grained parallelism as a paradigm for programming on multi-core architectures has made it critical to control the costs of communication. On top of that, accesses to local cache in hierarchical memory systems can be much faster than to non-local memory and systems with *Weak Memory Ordering* model can require expensive synchronization to deal with operations on concurrent data structures. The user's increasing demand for efficiency also contributed to the development of this field. However, parallel computing can be difficult to achieve, creating a need for tools that help make parallelization of code easier.

There are two main approaches to parallel processing. *Data parallelism* is very common since it is generally easier to implement. Relies on splitting the computation by dividing data across multiple processors, executing the same operation on each partition of data.

On the other hand, *task parallelism* assumes that multiple tasks can be executed in parallel by different processors. This type of parallelism is thought to be harder to implement due to lacking regularity: threads run different instructions leading to different execution schedules and memory access patterns.

The generation of tasks and their mapping/scheduling to processes is very important in parallel algorithms.

When generating tasks, the granularity of a task is very critical, as most scheduling algorithms are optimized for tasks whose grain is not too fine and not too coarse. Task generation can be categorized into static and dynamic. Using the former method, all tasks and task dependencies are known *a-priori* (data and recursive decomposition techniques often lead to static task generation).

In the latter method, tasks are generated during the execution of the algorithm (typically decomposed using exploratory or speculative decomposition), creating the need for a sophisticated scheduler.

1.2 Scheduling Parallel Computations

The scheduling of a parallel computation must address several concerns:

C1 independent tasks must be assigned to different processes;

C2 tasks on the critical path (the longest path of tasks that depend on one another) need to be assigned as soon as they are available because they are required to be executed serially;

C3 interaction between processes has to be minimized by mapping tasks with dense interaction to the same process.

A problem arose concerning task parallelism with dynamic task generation. There was a need for an efficient scheduling algorithm that must ensure enough tasks, within reasonable limits, are computed concurrently to keep processors busy while trying to maintain related tasks on the same processor. Two scheduling paradigms have emerged to address these types of computations: *Work Sharing (WShare)* and *Work Stealing (WSteal)*.

In WShare, whenever a processor generates new tasks, the scheduler attempts to transfer them over to other processors, hoping that the work has been distributed to underutilized processors. This leads to a high communication overhead between processors as there is constant process migration.

The scheduling paradigm that this paper will focus on is WSteal, which contrasts with WShare. Instead of sharing tasks when these are generated, processors only try to acquire work when they don't have any, leading to less process migration. To achieve this each processor owns a **deque** (double-ended queue) that stores the tasks it owns during the course of the execution of the algorithm. If a process has work, it obtains the bottom task in the deque and begins its execution, until it either completes or stalls, after which the process repeats this procedure. When the deque becomes empty (processor runs out of work), the processor becomes a thief and starts looking for work, targeting possible victims (processes that have work), attempting to steal a task from the top of their deque. This stealing phase ends when the processor successfully steals work or when work is added to its own deque, after which the process starts the task's execution [4].

1.3 The Problem

With WSteal rising in popularity, it became widely used and the target of various studies. As such, multiple flaws and limitations have since been pointed out.

One big limitation that was previously mentioned (Concern 2 of Section 1.2) is the fact that the *critical path* must be executed serially and, because of such, for certain workloads, most of the work is generated by a single processor. This can lead to work generation being higher than the steal rate, resulting in that processor's pool becoming a bottleneck [5].

Another problem is related to data locality, a factor when dealing with modern CPU architectures that are designed with multiple memory hierarchies. Due to the lack of data locality, WSteal does not perform as efficiently as it is possible.

In addition, when working with fine-grained applications, it has been proven that for certain irregular graphs, it is more advantageous to steal more than one task at a time [13]. Applications with low parallelism also struggle due to workers being most of the time trying to balance the load. Furthermore, thieves often attempt to steal other workers without tasks due to the random selection of their victims. In this scenario, it would be preferable to use a sender-initiated approach which would result in much more efficient execution.

Lastly, WSteal is also limited in the sense that the use of synchronization can not be completely eliminated from the implementation of concurrent deques.

1.4 Proposal

Rito and Paulino proposed the *Low-Cost Work Stealing (LCWS)* [20], an extension of [3], implemented using a lock-free split deque, instead of the usual concurrent deque, having both a private part and a public part, allowing the removal of most of the unnecessary synchronization overheads, while maintaining the same guarantees of WSteal.

It resembles WShare to a certain degree, due to workers being involved during the stealing phase (apart from being the victim). Furthermore, the algorithm is very flexible and easily adaptable, which allows further experimentation with different variants derived from it.

We propose a C++ implementation of the algorithm, along with several variants and optimizations that further improve performance.

1.5 Contributions

Our major contributions are:

1. A practical implementation of the LCWS algorithm.
2. Testing both the LCWS algorithm and the canonical WSteal, in addition to other variants, to compare their performances.
3. The gains of each optimization added to the implementation.

1.6 Document Structure

The remaining of this document is structured as follows: in Chapter 2, we start by exploring the intricacies of locks and how they can be used. Then we examine the main WSteal algorithm and how parallel computations are modeled. In addition to the main

WSteal algorithm, we also look at the Non-Blocking Work Stealing (NBWS). It's followed up by related work, each describing the problem each contribution tries to fix/mitigate and possible drawbacks.

In Chapter 3, we focus on the LCWS by Paulino and Rito, which we implement in C++, also analyzing the lock-free split deque's specifications. Furthermore, we describe the implementation steps of the different variants we created as well as the various testing/comparisons that we ought to do. Then, in Chapter 4, we evaluate all the different implemented algorithms via the use of the Problem Based Benchmark Suite (PBBS) for testing. Lastly, in Chapter 5, we conclude this thesis by mentioning our major contributions and possible paths for future work.

2

Background and Related Work

In this chapter, we will introduce some notions in Section 2.1 regarding synchronization and how it can be achieved. We will also explore the canonical Work Stealing (WSteal) algorithm in Section 2.2 and, the Non-Blocking Work Stealing (NBWS) in Section 2.3 and other variants that arose to fix a particular problem or simply to boost the performance in Section 2.4 and Section 2.5.

2.1 Lock-based and Lock-free Synchronization

Lock-based and lock-free synchronization are well-established topics, studied in most computer science courses, and as such, this section is based on the Book [19].

Processes in a multiprocess program interact with each other and as such, a set of rules and mechanisms must be enforced on the implementation of sequencing properties of statements issued by processes so that all different possibilities of the execution of the multiprocess program are correct.

A lock is one of such mechanisms whose function is to enforce limits on resource accesses by allowing only one thread to own the lock. It provides two methods: *acquire* and *release*. Acquire allows a thread to take ownership of a lock at a time. If another thread owns the lock, the thread requesting it blocks until the lock is released. Release simply makes a thread let go of the lock's ownership.

Locks have drawbacks related to the blocking that a thread does if it tries to acquire an already owned lock as well as threads which, for any reason, pause execution while other processes have to wait for it to end its operations to acquire the lock. Apart from these flaws, working with locks further requires the implementation of mechanisms that guarantee starvation-freedom and deadlock-freedom progress conditions. Starvation of a thread implies that this thread is continuously not making progress. An example of this is the scenario mentioned in Section 1.3, where the workload is mostly composed by the *critical path*, meaning that a single thread will do most of the work while the others starve. On the other hand, a *deadlock* happens when two processes can't execute as they are waiting for each other's resource(s). One such resource is storage and this type of

situation can happen when limiting space in certain parallel computations. In a scenario where two threads have already allocated most of their needed storage but suddenly there is no more storage left and neither can be satisfied. Burton shows [7] how this can be prevented by determining how much storage would have been available to a task in a sequential system, and to ensure that at least as much storage is available to the task when it is executed in a distributed system.

While locks are a useful and easy way to ensure the correct execution of multiprocess programs, the lack of efficiency due to blocking and possible faults inspired the implementation of lock-free algorithms. These algorithms aim at new progress conditions, unique to lock-free object implementations.

A lock-free implementation of an object implies that its operation's code does not have any *critical sections* (sections that can only be executed by a process at a time). One problem is that it allows the base operations generated by invocations to be interleaved. As this is exactly what locks are trying to prevent, there is a need for stronger progress conditions:

1. *Obstruction-freedom* relates to concurrency, introducing the notion that each time an operation is invoked is executed in isolation, it terminates in a finite number of steps.

2. *Non-blocking* (or *lock-freedom*) guarantees that at least some thread is making progress. Individual threads may starve but *infinitely often*, operations ran by some processors will conclude in a finite number of steps. *Infinitely often* is a term used in Probability Theory, which can be explained briefly by the following example: The chance of someone tossing a coin 100 times and it always landing heads is 0 (7.8886091e-31). This can be said when this experiment is happening in finite space (the sum of probabilities is finite). If said person were to do this experiment infinitely, then the sum of probabilities is infinite. This means that the event where the coin lands heads 100 times in a row is bound to happen and will do so *infinitely often*.

3. A *wait-free* implementation ensures that any process can complete any operation in a finite number of steps. It requires no fairness or liveness conditions and provides fault-tolerance, meaning that no process can be prevented from completing an operation by halting failures of other processes or arbitrary variations in their execution speed.

These three progress conditions define a hierarchy between them. All *wait-free* implementations are *non-blocking* and all *non-blocking* implementations are *obstruction-free*.

There are three families of solutions when it comes to achieving synchronization between processes in algorithms:

1. Atomic read/write shared registers;

2. Safe, regular and atomic registers;

3. Built-in primitives of multiprocessors.

Atomic read/write shared registers: even though these registers are shared among processors, their design ensures that all operation invocations appear as if they have been executed sequentially.

Safe, regular and atomic registers: safe register's read operation, when concurrent with write operations, can return **any** value that the register may contain.

A regular register is a stronger safe register that addresses this property differently as such that when a read is concurrent with write operations, it can either return the value previous to these writes or return any value corresponding to one of the writes. These two registers are an approach to implement atomicity while not relying on underlying atomic objects.

An atomic register is the strongest and provides total ordering on all operations.

Built-in primitives of multiprocessors: most shared memory multiprocessors offer primitives specially designed to deal with synchronization. These primitives are: Let R be a shared register initialized with value 0

1. *test&set* sets R's value to 1 and returns the old value. *reset* resets R to its initial value.

2. *swap(v)* writes *v* to R and returns the previous value.

3. *compare&swap(old, new)* receives two arguments and returns a boolean. The *new* value is only written if R == *old*, and in this case, it returns *true*, otherwise, it would return *false*.

4. *fetch&add* simply increments R by one and returns value. Some variants of this instruction return the value before the increment and others also receive an argument and increment according to that argument.

Although *test&set* and *swap(v)* might seem to do the same thing, the main difference is that the first one is a 1-bit comparand while the second is 32-bit.

Another important hardware instruction that can be used is the *memory fence*. It simply enforces reads and writes to memory in order, meaning that instructions might have to wait for a previous instruction to finish. This instruction along with Compare-And-Swap (CAS) are used in WSteal algorithms.

2.2 The Work Stealing Algorithm

As mentioned in Section 1.1, multithreaded computing is a way to utilize unused computing resources. The branch this paper focuses on is task parallelism which comes down to parallelizing tasks across different threads. To do so during execution, a scheduler is needed to handle task distribution efficiently. The WSteal algorithm is one such scheduling strategy. It aims to reduce process migration in the sense that threads only seek to communicate when they don't have any work left.

Robert Blumofe and Charles Leiserson presented the first provably good *Randomized Work Stealing (RWS)* scheduler for well-structured multithreaded computations with dependencies [5]. This type of computation includes both backtrack search and divide-and-conquer computations, as well as data flow computations in which threads may stall due to a data dependency. The randomized nature of the scheduler they present comes from the fact that the victim chosen by a processor is done so uniformly at random. Their work proved optimal worst-case bounds on expected execution time, space required and total communication cost.

2.2.1 Modeling a Parallel Computation

To better understand how WSteal works, we first need to understand how parallel computations are modeled in WSteal. This section is based on the Paper [6].

Computation is modeled as a Directed Acyclic Graph (DAG) which is composed of a set of threads, each one being a sequential ordering of unit-time instructions. Instructions are then connected by *dependency edges*, establishing partial ordering in which an instruction must execute before another. In order to execute a thread, a chunk of memory (referred to as *activation frame*) is allocated, used by the instructions to store the values on which they compute.

During the course of execution, a thread may generate other threads that operate concurrently with each other and the parent thread. Parent threads connect to their spawned children by *spawn edges*, arranging the threads as a spawn tree. Apart from *dependency edges*, *join edges* also exist for the cases where there is a producer/consumer relation between threads, meaning that a thread produces a data value that another one consumes.

A multithreaded computation does not model detection or the resolution of these *join dependencies* and, as such, must be designed in the implementation. A multithreaded computation in which all *join edges* from a thread connect to ancestors of the thread in the activation tree is called *strict*. This strictness guarantees that a thread cannot be invoked before all its arguments are produced. A multithreaded computation can also be *fully strict*, implying that all *join edges* from a thread connect to the thread's parent.

The work of the computation is defined by the total number of instructions, while the *critical path* length is determined by the longest directed path in the DAG.

2.2.2 Busy-Leaves Algorithm

The RWS algorithm is based on a simple algorithm that uses a central deque. This algorithm is called the Busy-Leaves Algorithm. The schedules computed by this algorithm maintain the *busy-leaves* property at all steps for all *strict* computations.

2.2.2.1 Busy-leaves Property

Considering a *strict* computation, during the lifespan of a given thread T, there is at least one thread from the subcomputation rooted at T that is ready, meaning that leaves can't stall and are thus always ready.

Another relevant property is the *greedy* property. For any step of the execution of a *Greedy* schedule, if at least P instructions are ready, then P instructions execute, and if fewer than P instructions are ready, then all execute.

When both these properties are combined, it is possible to derive execution schedules that exhibit both linear speedup and linear expansion of space.

2.2.2.2 Algorithm Procedure

The algorithm operates in steps and at each step, the algorithm has previously computed and executed the steps that precede the current step. When operating in the current step, the algorithm does not use information about instructions not yet executed or threads that still have not been spawned. A single global thread pool is used to maintain all living threads, which is uniformly available to all processors. All spawned threads are added to the pool and when a processor needs work, a ready thread is removed from it.

The algorithm begins with all processes idle and the root thread in the global thread pool. At the start of each step, a process can either be idle or has a thread to work on. Idle processes thus try to remove a ready thread from the pool. A process that had or obtained work executes the next instruction of that thread and, in general, executes an instruction per step until it either spawns, stalls or dies.

The downsides of this algorithm are its inefficiency, in most cases, in computing execution schedules. In addition, it lacks scalability due to the single global thread pool in which all processors must compete for access. These flaws are where the WSteal algorithm comes in, being a distributed online scheduling algorithm that is both efficient and scalable.

2.2.3 Randomized Work-Stealing

The previous centralized thread pool of the Busy-Leaves algorithm is now distributed across processors, with each processor maintaining a *ready* deque data structure of threads. As has been explained in Section 1.2, threads can only be inserted from the bottom but can be removed from both ends. Processors treat this data structure like a call stack,

pushing and popping threads from the bottom while threads that are migrated to other processes are removed from the top.

A processor obtains work by removing the bottom-most thread from its deque and starts working on it, executing its instructions until it either spawns, stalls, dies or enables a stalled thread.

1. If a thread spawns another thread, the child is placed on the bottom of the deque and the processor begins working on it. This strategy called *work-first* task scheduling policy is the usual approach and can lead to the parent thread being stolen while the processor is working on the child. However, another fairly common strategy is the *help-first* task scheduling policy, which is supported by several implementations like SLAW [12] and instead, the processor keeps working on the parent thread and lets another thread work on the child, and in case this does not happen, it will work on the child later.

2. In the case of a thread stalling or dying, the processor checks the deque for work. If it contains any work, then the bottom-most thread is removed and the processor begins working on it. If it is empty, the processor starts stealing work.

3. If a thread enables another, the now-ready thread is placed at the bottom of the deque of the enabler thread's processor. Supposing the multithreaded computation is *fully strict*, the previously stalled thread must be the enabler's parent.

These rules are important to ensure critical structural properties, such as the *busy-leaves property*.

As the algorithm begins, all *ready* deques are empty, apart from one processor which contains the root thread of the multithreaded computation. All other processors start work stealing.

This work-stealing phase operates as follows. A processor becomes a thief, attempting to steal work from another processor (the victim), which is chosen uniformly at random. It starts by querying the *ready* deque of the victim, and if it's not empty, the thief removes the top-most thread and begins to work on it. In the event that it is empty, the thief tries to select another victim at *random*. A problem arises when multiple thieves attempt to steal, simultaneously, the same victim, creating the need to define a model to cope with this issue. The *atomic-access* model is such a model and it assumes that concurrent accesses to the same data structure are serially queued by an adversary. So when concurrent steal requests are made to the same deque, in one time step, one of them is satisfied while the others are queued by an adversary. It is the adversary who decides which requests will be satisfied and which will have to wait.

This procedure implies synchronization between processors and thus communication overhead. Assuming P processors and *critical-path* length C, the total communication cost during the execution of a *fully strict* computation is $O(PC(1 + N)S)$, with N being the

maximum number of *join edges* from a thread to its parent and S the largest size of any activation frame.

2.3 Non-Blocking Work Stealing Algorithm

Arora et al. [3] later implemented a *NBWS* algorithm that minimizes the number of *CAS* operations that processes are required to perform. It does so because a CAS operation is only necessary when the deque has one or fewer items, being the instruction only required to reach consensus. The non-blocking nature comes from the fact that the algorithm uses a *lock-free* deque and because of it processes need to perform a yield system call between every pair of consecutive steal attempts. A yield system call is made so that a process gives up CPU time without suspending, giving other processes, with the same or higher priority, a chance to run. These calls are needed to prevent the kernel from potentially starving a process.

This algorithm was proven to have high performance in multiprogrammed environments, which previous traditional thread schedulers did not. As the algorithm is *non-blocking*, process faults also do not block but can cause the loss of items.

2.4 Optimizations of Work Stealing

Ever since WSteal became widely adopted, several studies have been conducted that greatly impacted the algorithms performance-wise. These studies focused on different areas that we categorized into the four following sections.

2.4.1 Overflow problem of NBWS

Chase and Lev propose a new variant of the WSteal algorithm (Dynamic Circular Work-Stealing) [9] that intends to correct the underlying overflow problems of Arora's NBWS algorithm. This problem arises from the fact that the synchronization protocol heavily relies on the use of *fixed-sized* arrays which are prone to *overflows*, even more so in multiprogrammed environments for which the algorithm is designed.

Their algorithm fixes this problem by making use of a *cyclic* array for the deque. This way, the array can be used more efficiently. In the event that a push operation discovers that the array is full, it simply copies the array's contents to a bigger array, not needing to alter the indexes *top* and *bottom* (that indicate both ends of the deque).

It also successfully fixes the overflow problem with linear space complexity while not requiring garbage collection for memory management. However, the *pushBottom* operation always requires reading the *top* variable, which can increase the number of *cache misses*, lowering performance. To mitigate this issue, the author proposes keeping a local upper bound on the size of the deque and only reading its the *top* when the upper bound indicates that an array expansion is necessary.

2.4.2 Locality

Locality concerns frequent access to the same value or storage location by the same CPU. Because modern CPU architectures are designed with multiple memory hierarchies, whose access speeds differ, locality becomes an exploitable source to further improve performance.

Blumofe's WSteal algorithm does not take full advantage of this. Even though it tends to execute tasks as if they were in sequential order, and it is believed that there is inherent data locality in sequential execution, the algorithm is not cache-friendly.

Researched solutions can be divided into three main branches. The first requires programmers to assign *locality hints*, during runtime, to spawned tasks. The second one focuses on distributing data across the local memories of a distributed shared memory system. Threads in the computation are scheduled on the nodes that hold the data they require. The last branch is based on the proximity of executing threads on the same process on the computation graph.

The WSteal algorithm achieves good data locality as it executes tasks that are close in the computation graph on the same processor but for some applications, tasks that access the same data are not close in the computation graph.

Acar et al. [1] describes a *heuristic modification* to WSteal, designed to allow locality between tasks that are distant in the computation graph. It does so by allowing each task to be given an *affinity* for a processor, and having processors prioritize tasks that have an *affinity* for them. To achieve this, each processor has an additional queue called *mailbox* that stores pointers to tasks that have an *affinity* for the processor. However, the delays induced by the insertions on this extra queue made it so that the study lacks bounds on expected execution time and space requirements. Furthermore, a known issue is that when a task's structure is reused in a computation step, the existing copies from the previous step, present either in the deque or the *mailbox*, need to be marked as *invalid*.

SLAW [12] is a locality-aware WSteal algorithm where processors are grouped together forming multiple clusters. Each cluster contains multiple processors and a *mailbox* to store incoming tasks sent from other clusters. Tasks are spawned in the cluster of the processor that owns the task's parent unless a *locality hint* is specified by the programmer. When a processor runs out of work, it will attempt to steal from its cluster's *mailbox*. If it also has no work, it will try to steal from the processors in its cluster. Only after not being successful in these places will it try to steal from other clusters' *mailboxes*. The framework also gives the option to disable cross-cluster steals thus introducing a trade-off between cross-cluster load imbalance and counterproductive steals.

2.4.3 Reducing Scheduling Overhead

2.4.3.1 Task Granularity

Task granularity is related to the amount of work that the task performs between communication/synchronization events. In *fine-grained* parallelism, the computation is broken down into a large number of small tasks, facilitating load balancing. In case the granularity becomes too fine, it's possible that the synchronization overhead between tasks will take longer than the computation. As for *coarse-grained* parallelism, there is a small number of large tasks, making it harder to balance the load efficiently.

One way to reduce task scheduling overhead is to avoid creating tasks that are too *fine-grained*. This strategy comes with the risk that tasks may become too *coarse-grained*, and thus, load balancing becomes a problem again.

In the Paper [12], a cut-off approach is proposed to avoid creating sub-tasks that are deep in the task spawn tree. This policy will cause some threads to become idle if the tree is unbalanced. Loop parallelism also uses a chunking technique to reduce overhead.

Even though one should avoid the creation of *fine-grained* tasks, the scheduler proposed in Paper [24] enables the efficient execution of *fine-grained* tasks. It prevents the tedious manual coarsening required by the popular WSteal algorithms that keep novice and general-purpose programmers from parallelizing their code. Another difference from other adaptive dynamic schedulers is the fact that no additional state is maintained to infer system load, nor does it make irrevocable serialization decisions. These qualities allow the algorithm to scale well while providing excellent load balancing at a much lower overhead cost.

The author makes two key points on which his algorithm is based. The first one revolves around the fact that a processor that is pushing work onto its local deque is likely to be an unneeded overhead if other workers are busy with their work. In such a scenario, it would have been better if the worker performed some work before checking the overall system load to yet again decide if it would push work into its deque. Such a procedure would have resulted in the avoidance of unnecessary deque instructions. However, an approach where a worker checks the system load to infer if other workers are busy without incurring expensive overhead is not obvious.

This is when the second insight is introduced. It provides a lightweight *heuristic* to infer system load during runtime. It introduces a new logical state in which tasks may be *postponed*, residing in memory private to the worker who owns them. A worker frequently compares its deque size with a *threshold*. If it is below it, the worker pushes work onto the deque, otherwise pushing tasks is *postponed* and the worker keeps focusing on doing his work. Furthermore, the operation of checking the deque's size is not *atomic* nor requires any sort of communication as long as the size checks are frequent (reading a stale value does not influence efficiency). The author concluded that the frequency of these checks

should be large, relative to the program's parallelism, to avoid increasing the *critical path* of lazy schedules.

In summary, Lazy Scheduling is much less susceptible to parallelism granularity. This allows the overexposure of parallelism with minimal performance degradation, due to the very low overhead, in the usual case when generated tasks are executed by the worker that spawned them, compared to existing approaches.

2.4.3.2 Stealing Multiple Items

Stealing one item at a time has been shown sufficient to optimize computation along the *critical path*, up to a certain degree [5, 3].

Larry Rudolf attempted to implement a load balancing scheme for task allocation on parallel machines [21]. This scheme works as such that, the amount of work that a thread has, dictates the *frequency* as to when it tries to distribute its load with another processor. This procedure chooses another processor at random and attempts to balance the workloads, migrating work from the heavier workload to the lighter one. The drawback is that the process owning the queue needs to use a strong communication operation for every *pop* or *push*. As these operations are much more common, the algorithm becomes less effective when compared with other traditional algorithms.

Hendler and Shavit, having both these approaches in mind, sought out to implement a new generalization of Arora's NBWS algorithm (the Non-Blocking Steal-Half Work), allowing processes to steal up to half of the items of a given queue at a time while preserving the non-blocking properties as well as minimizing the CAS instructions [13]. It can do so by utilizing an extended deque data structure, which differs from the usual deque by being implemented as a *cyclic array* and containing a structure that defines the range of items (approximately half) that can be stolen atomically.

Their analysis proves that their algorithm provides better load balancing, being the expected load of any process very close to the overall system average.

2.4.3.3 Removing Memory Fences

Another strategy for improving WSteal was to remove *memory fences*. This instruction was necessary for correctness when removing a task, but incurs in expensive overhead. One of the main objectives of Low-Cost Work Stealing (LCWS) is to eliminate most of the unnecessary synchronization overheads, which includes *memory fences*.

The *sender-initiated* and the *receiver-initiated* algorithms of Paper [2] eliminate these *memory fences* from the common scheduling path, by making use of private deques. They were specifically designed for *fine-grained* parallel computations, due to requiring explicit communication between processors when balancing load. The small task size implies that calls to the scheduler are frequent, allowing most of the polling for communication to be done by the scheduler without interruption.

These algorithms only require the implementation of a simple non-concurrent deque data structure and support the *steal-half* load balancing policy. By eliminating concurrency from local deque operations, they enable the implementation of more sophisticated task scheduling techniques.

They differ in the way they communicate and function.

In the *receiver-initiated* algorithm, communication happens using two different cells. The *request* cell is where thieves use a CAS instruction to write their *identifiers* when making a steal request to the victim. The *transfer* cell stores the answer (work) of the victim and a thief needs to repeatedly read this cell until the answer is available, which happens when the victim checks its *request* cell. Additionally, the *status* cell indicates whether a victim has work to steal.

In the *sender-initiated* algorithm, there is only one cell, used both for communication and status. When a processor becomes idle, it writes the value *null* in its cell. It then waits until a busy processor attempts to deliver work to it. To guarantee that processors do not share work concurrently, the sender writes the constant *incoming* on the receiver's cell. This is only needed if processors can only send one item.

The analysis of both these algorithms shows them to be competitive with optimal bounds for executing *fine-grained* parallelism on modern multi-core architectures. Their approach further enables the design and implementation of sophisticated task coalescing and scheduling techniques that accelerate uncommon problems.

Another successful attempt to remove the *memory fence* was pursued in the Paper [16], making use of a bounded *Total Store Ordering (TSO)* model. This model refers to memory ordering which translates to the order of accesses to computer memory by the CPU. This model guarantees that the sequence of instructions that appear in memory for a given processor is identical to the sequence in which they were issued by the processor. This model allows a stored value to be buffered before being stored in memory. This allows a later load from a different address to be satisfied by the value in the buffer while it is not written in memory.

The paper then introduces its *bounded TSO* model, in which a restriction was added, placing a *fixed bound* on the size of the abstract store buffer. This model permits the scenario in which a worker does not emit a *memory fence*. A thief can use knowledge of the store buffer's capacity to infer when it can safely steal a task by bounding the number of task removals hidden in the worker's store buffer. If a thief is not certain that he can safely remove a task, he simply aborts, preventing the state in which a task is removed more than once. This approach also opens a path for removing *memory fences* from concurrent algorithms because it circumvents the *impossibility result* introduced in Paper [4].

Much like the LCWS's deque, Paper [11] makes use of a lock-free split deque to minimize the number of memory fences. Their deque only requires the use of these instructions when reclaiming tasks from the public part of a deque. When a thief detects that the shared portion is empty, it alters the victim's flag *splitreq*. This flag is checked on every *push* and *pop* operation to ensure that its value is always in a dedicated *cacheline*, making it so that there is no further communication until the flag is set. A owner of a deque can also alter the flag *allstolen* that the thieves check before attempting to steal tasks, resulting in a slight performance gain.

Their experiments show that their algorithm is competitive with the framework *Wool*, gaining near-optimal speedup for several benchmarks.

2.4.4 Latency-Incurring Computations

Steffan Muller and Umut Acar implemented an algorithm focusing on an issue overlooked by many [17]. They took into account that parallel applications nowadays not only consist of processing batch workloads but also take input from the user, as well as access secondary storage or the network, or perform remote procedure calls.

They present a WSteal algorithm, using an extended DAG model, that hides latency. In this variant, workers can have multiple deques, of which only one is used at a time. When a task spawns other tasks, they are also added to the active deque. In the scenario where a task suspends, it is tagged with the *event* on which it is waiting. The scheduler polls each *event* per invocation. Facing the suspension of its current task, the worker will first check if the active deque has work. If it does, it simply computes the next task as usual. If it does not, the worker proceeds to switch its active deque with another that he owns and has work, eventually returning to execute the pending tasks of the switched out deques.

When a worker has no *ready* deques (deques with at least one task that is not suspended), it begins the stealing phase, attempting to steal a random *ready* deque (ignores *non-ready* deques) from a random target, after which, if successful, the thief starts a fresh deque, with the newly acquired task, that also becomes the currently active deque. This procedure has a flaw due to the fact that at an execution step, many suspended tasks may resume, leaving their owner to handle them by itself, which will hurt performance. To address this issue, the scheduler partitions the resumed tasks according to the deques that own them. Each deque will then spawn a function that will, in a parallel loop, execute the resumed tasks.

This implementation efficiently schedules latency-incurring computations without stalling workers when a task induces latency and suspends. Even though the authors show that the scheduler can incur an overhead for latency that falls on the *critical path*, their prototype gives evidence that their approach is practical and has the potential to improve the performance of this type of computation, while not penalizing computations that don't induce any latency.

2.5 Hybrid Work Stealing and Work Sharing Solutions

As it has been mentioned in the proposal, our LCWS implementation includes optimizations combining Work Sharing (WShare) and WSteal. Other solutions employing this strategy have been carried out.

Jeeva Paudel et al. presents a hybrid parallel task-placement strategy in *X10* [18]. *X10* is a programming language designed for parallel computing using the *Partitioned Global Address Space* model. It primarily relies on WSteal within its shared-memory abstraction for load balancing, not employing any strategy in a distributed-memory setting. As such, the authors pondered the question that WShare complemented with WSteal could alleviate load balance and improve performance.

They introduce a *heuristic* to detect load imbalance in the system and start WShare. Workers are grouped in *nodes*, and both are considered *starved* if the worker has failed to steal work N times (in their prototype, N is equal to the number of workers per *node*). Then, any new spawned tasks in a *loaded* (not *starved*) *node* can be launched at a *starved node*. This *heuristic* does not require any communication between *nodes* to decide whether to transfer a task.

Although migrating tasks is a way to keep all workers busy, thus increasing performance, excessive migration can incur excess overhead through loss of locality and the associated costs of migration. Because of this, the paper introduces the notion of *place-flexible* tasks capable of offsetting these underlying costs. One task can be considered as such if:

1. the processor's cache is warm for that task because related data is already in its local memory;
2. the task is local to that process (the parent task originally migrated to that process);
3. the task's granularity is coarse enough to overcome the cost of migration;
4. the task encapsulates the data necessary for its computation.

The opposite of these tasks is *place-sensitive* tasks that have relatively low computation and do not provide a substantial improvement in resource utilization that makes them worth migrating.

This runtime scheduler restricts migration between nodes using WShare to only *place-flexible* tasks (these tasks have to be identified by the programmer with an annotation on the source code), while *X10*'s WSteal scheduler maps all *place-sensitive* tasks in a locality-guided manner at each *node*.

The key result of this paper is that using this simple *heuristic* to identify load imbalances in a system and then combining WShare and WSteal strategies improves the application's performance relative to a system that does not use this hybrid approach.

Lopes *et al.* [15] have also experimented with hybrid scheduling strategies, in the context of the pSystem parallel programming system for both shared and distributed architectures [23]. An initial heuristic simply switched between WSteal and WShare when the amount of work in a worker's deque was, respectively below or above a given threshold. A second, more refined heuristic, attempted to keep a worker's amount of work between two pre-defined thresholds. Whenever, such amount dropped below the lower threshold, the worker would become a thief. Conversely, whenever it surpassed the upper threshold, the worker would begin to spread work following a WShare strategy.

3

Proposed Solution

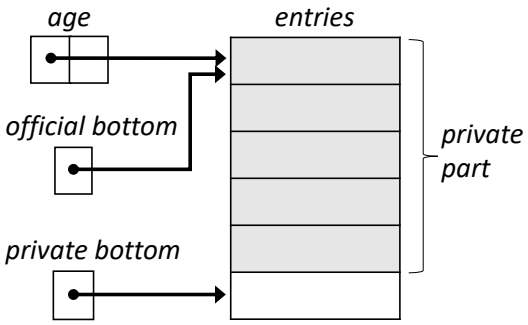
In this chapter we start by introducing the Low-Cost Work Stealing (LCWS) algorithm. It is a new scheduler that attempts to achieve the same results as Work Stealing (WSteal), but minimizes the communication costs of deque accesses to improve performance. We describe the differences that set it apart from the regular WSteal algorithm. We further describe the steps taken during the implementation of the lock-free split deque as well as of the algorithm. Afterwards, we proceeded to implement variants of the LCWS, one of which introduces Work Sharing (WShare), as a means to try and improve performance. Throughout the chapter, we will mention workers and processors, which mean the same thing. We present various snippets of pseudocode to simplify understanding the algorithms. The C++ implementations can be found in the Annex I.

3.1 The Low-Cost Work Stealing Algorithm

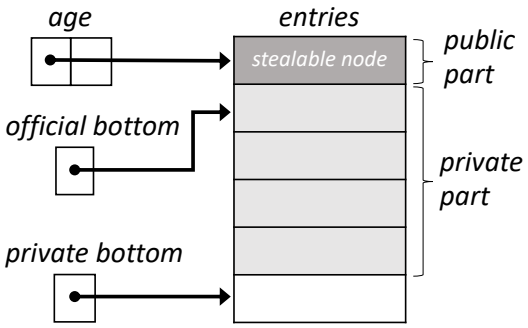
As mentioned in Section 1.4, LCWS emerges as a new work-stealing algorithm, seeking to fix the underlying synchronization problems of the canonical WSteal. It employs private deques to reduce these overheads and requires a different procedure when it comes to stealing work because the deques are split into two sections. This makes it so that the algorithm resembles WShare with WSteal due to both workers and thieves being involved in the load balancing process.

In LCWS, each processor owns a *lock-free split* deque (illustrated in Figure 3.1), which is divided into a public part and a private one, as well as a flag (called *targeted*) that stores a boolean value. This private part is thus only accessible to the owner of the deque, preventing the need for synchronization during local operations. The notion that thieves steal from the top of the deque still stands. This is because the upper portion of the deque comprises the public part, while the private part corresponds to the rest of the deque. A worker may have to seek work in the public part of it's own deque, competing with thieves, requiring synchronization. We will come back to this scenario after explaining how thieves are able to steal work.

A processor becomes a thief whenever it runs out of work. However, now that the



(a) A Split Deque with no stealable nodes.



(b) A Split Deque with one stealable node.

Figure 3.1: The split deque built from an array of nodes (named *entries*), and featuring a state composed of 3 variables:
age comprising fields *top*, that points to the Split Deque's top-most node, and *tag*, to ensure correctness;
official bottom points to the node below the bottom-most one of the Split Deque's public part, and, lastly,
private bottom points to the empty slot below the Split Deque's bottom-most node. Taken from Paper [20].

deque is divided into two, running out of work implies that both sections are empty. Following the common *stack-based* execution model, workers start removing tasks from the bottom of the deque, which comprises the private part. Once this segment is empty, they compete for tasks with thieves by attempting to take work from the public segment. Once all the work has been removed from the deque, the worker becomes a thief.

The theft phase works similarly to Randomized Work Stealing (RWS). A thief chooses a processor at random and attempts to steal from it. Just like before, he tries to remove work from the top of the deque. However, in this case, the top of the deque is separated from the rest. It may have work and so the thief either succeeds in the removal of it or ends up getting raced by another thief or the owner itself. The other possibility is that there is no work to be found there. Although this segment of the deque is empty, the private part may still have work. But thieves cannot access this section. Furthermore, tasks are always pushed to the bottom of the deque, meaning that the public part would always be found to be empty if there were no other mechanisms involved.

Algorithm 1 LCWS Algorithm. Taken from Paper [20].

```
 1: procedure SCHEDULER
 2:     while computation not terminated do
 3:         if self.targeted then
 4:             self.spdeque.UPDATEBOTTOM( )
 5:             self.targeted ← FALSE
 6:         end if
 7:         if VALIDNODE(assigned) then
 8:             enabled ← EXECUTE(assigned)
 9:             if LENGTH(enabled) > 0 then
10:                 assigned ← enabled[0]
11:                 if LENGTH(enabled) = 2 then
12:                     self.spdeque.PUSH(enabled[1])
13:                 end if
14:             else
15:                 assigned ← self.spdeque.POP( )
16:                 if assigned = RACE then
17:                     assigned ← self.spdeque.POPBOTTOM( )
18:                 end if
19:             end if
20:         else
21:             self.WORKMIGRATION( )
22:         end if
23:     end while
24: end procedure
25: procedure WORKMIGRATION
26:     victim ← UNIFORMLYRANDOMPROCESSOR( )
27:     assigned ← victim.spdeque.POPTOP( )
28:     if assigned = EMPTY then
29:         victim.targeted ← TRUE
30:     end if
31: end procedure
32: function VALIDNODE(node)
33:     return node ≠ EMPTY
            and  node ≠ ABORT
            and  node ≠ NONE
34: end function
```

Upon not finding work, thieves are able to *notify* the owner of the deque on which befell the steal attempt. Workers that are not idle check, in every cycle, if they have been notified. If they have, they can move work stored privately to the public part of the deque. This indirectly gives rise to the reason why workers may also need to remove work from the public part of their deques. In a scenario where a processor makes work publicly available to others, it continues to work, popping tasks from the bottom of its deque. When this segment runs out of work, the task that was transferred (to the public part) in the past might have ended up not getting stolen. This means that there is no reason for the processor to become a thief yet. He may remove this task, but now there is an extra cost. This operation is not local anymore, because it is shared with other processors. Communication is necessary and the owner may in fact be raced by thieves on the removal of this task. Only after a worker is unsuccessful in removing work from

Algorithm 2 The Split Deque implementation. Taken from Paper [20].

```
    privateBottom ← 0                                          ▷ private field
    entries ← {}                                               ▷ private read-write, public read-only
    officialBottom ← 0                                         ▷ private read-write, public read-only
    age ← {0, 0}                                               ▷ public field
 1: procedure PUSH(node)
 2:     pBot ← self.privateBottom
 3:     self.entries[pBot] ← node
 4:     self.privateBottom ← pBot + 1
 5: end procedure
 6: procedure POP
 7:     pBot ← self.privateBottom
 8:     if pBot = self.officialBottom then
          return RACE
 9:     end if
10:     pBot ← pBot − 1
11:     node ← self.entries[pBot]
12:     self.privateBottom ← pBot
13:     return node
14: end procedure
15: procedure POPTOP
16:     oldAge ← self.age
17:     oldBottom ← self.officialBottom
18:     if oldBottom ≤ oldAge.top then
          return EMPTY
19:     end if
20:     node ← self.entries[oldAge.top]
21:     newAge ← oldAge
22:     newAge.top ← newAge.top + 1
23:     if CAS(age, oldAge, newAge) = SUCCESS then
24:         return node
25:     end if
26:     return ABORT
27: end procedure
28: procedure UPDATEBOTTOM
29:     pBot ← self.privateBottom
30:     oBot ← self.officialBottom
31:     if pBot > oBot then
          oBot ← oBot + 1
32:     end if
33:     self.officialBottom ← oBot
34: end procedure
```

Algorithm 3 The Split Deque implementation (continuation). Taken from Paper [20].

```
35: procedure popBottom
36:     oBot ← self.officialBottom
37:     if oBot = 0 then
          return EMPTY
38:     end if
39:     oBot ← oBot − 1
40:     self.officialBottom ← oBot
41:     node ← self.entries[oBot]
42:     oldAge ← age
43:     if oBot > oldAge.top then
          return node
44:     end if
45:     self.officialBottom ← 0
46:     self.privateBottom ← 0
47:     newAge.top ← 0
48:     newAge.tag ← oldAge.tag + 1
49:     if oBot = oldAge.top then
50:         if CAS(age, oldAge, newAge) = SUCCESS then
51:             return node
52:         end if
53:     end if
54:     self.age ← newAge
55:     return EMPTY
56: end procedure
```

the deque's public part does it become a thief. The fact that there was such an attempt means that the private part was already empty and we can be certain that it really does not have work.

The aforementioned lock-free split deque is thus the foundation on which LCWS is built. Algorithm 2 and 3 depict a possible implementation for such deque, where each instance of the data structure comprises a state composed of four fields:

entries - array of nodes ready to be executed.

privateBottom - index below the bottom-most node of the deque.

officialBottom - index below the bottom-most node of the deque's public part.

age - structure composed of two fields: *top*, the index of the top-most node, and *tag*, necessary to ensure correction (by avoiding the famous ABA problem [10] related to the Compare-And-Swap (CAS) instruction).

The deque offers an interface of five methods, whose brief description follows:

push (*node*) - Pushes *node* into the bottom of the split deque's private part.

pop - Removes and returns the bottom-most node of the deque's private part. If it is empty, returns a special value: RACE.

updateBottom - Transfers the top-most node of the deque's private part to the bottom of the public part, not returning any value.

popBottom - Removes and returns the bottom-most node of the deque's public part. If it is empty, returns a special value *EMPTY*.

popTop - Attempts to remove and return the top-most node of the deque's public part. If it is empty, returns a special value *EMPTY*. If the operation aborts, it has no effect and returns *ABORT*.

The deque meets the relaxed semantics presented in Paper [3] on any good set of invocations.

All sets of invocations are good because the *push*, *pop*, *updateBottom*, and *popBottom* are never invoked concurrently due to only the owner of the deque having access to invokes such instructions.

The algorithm achieves bounds on the expected total synchronization of

$$\mathcal{O}(PT\infty(CAS + MFence))$$

where P is the number of processors, $T\infty$ is the critical path length, CAS is the number of CAS operations and $MFence$ is the number of memory fences). When compared with the canonical WSteal, LCWS's number of synchronization operations and notifications (altering of the *targeted* flag) grows linearly with the span of the Directed Acyclic Graph (DAG) while the canonical WSteal grows exponentially.

3.2 A C++ Implementation

One of our goals is to implement the LCWS algorithm and evaluate its gains when compared with the WSteal algorithm. In order to capture data regarding the performance of these algorithms, we rely on the Problem Based Benchmark Suite (PBBS)[1]. This benchmark suite features a wide range of benchmarks designed to make possible the comparison of different algorithms and implementations, regardless of the programming language they are coded in. Each benchmark can be further divided due to the presence of multiple data sets that are being provided. The suite's Internet page provides all the information the user may need to understand how to run the benchmarks. Lastly, one big advantage of using this collection of benchmarks is that the framework's own scheduler, responsible for load balancing the jobs that make up each benchmark, already uses WSteal. This means that it is fairly easy to replace the algorithm with another variant.

To accomplish our adaptation of the pseudo-code in Algorithm 2, we also need to implement the lock-free split deque, following the pseudo-code in Figure 3.1. We sought out different implementations of regular lock-free deques in C++ so that one can be used as a starting point.

[1] Available from `https://cmuparlay.github.io/pbbsbench`

3.2.1 Choosing a Base Regular Deque

Our search for a C++ implementation of a regular lock-free deque focused on works related to the papers reviewed in Section 2.4.

Starting with Non-Blocking Work Stealing (NBWS)'s deque [3], from which the others are usually extended, it supports only three methods: *pushBottom*, *popBottom*, *popTop*. For the implementation to be concurrent, it makes use of CAS instructions. The paper fully describes all the methods with pseudo-code but no implementation in C++ could be found.

Following up with Dynamic Circular Work-Stealing's deque [9] that only intends to fix NBWS's overflow problems. As mentioned in Section 2.4.1, it functions like a circular array not held down by a fixed size. It provides all the methods of the NBWS's deque in a similar fashion (the main difference is that no variable *tag* exists) and a new private method *casTop*. This method simply depicts a CAS instruction, receiving the arguments *old* and *new*, overriding the *top* value with new if $top = old$. The paper contains pseudocode in Java. However, due to the popularity of the algorithm, several implementations in C++ can be found (with improvements regarding the Paper [14])

- `https://github.com/ssbl/concurrent-deque`
- `https://github.com/ConorWilliams/ConcurrentDeque`
- `https://github.com/taskflow/work-stealing-queue`

Also implemented as a cyclic array, the deque from the Non-Blocking Steal-Half Work algorithm [13] differs from the other deques by containing a structure (*stealRange*) that defines the range of items that can be stolen by another process. It contains three elements:

tag - a timestamp indicating the last update of this structure. Necessary to overcome the ABA problem inherent to the CAS instruction.

top - pointing to the top-most item of the deque.

stealLast - pointing to the last item to be stolen and can represent the special value *null*, indicating that there are no elements to steal.

The paper provides high-level pseudo-code for all the deque's methods with no C++ implementation to be found.

The framework *Lace*, which makes use of a split deque, also provides high-level pseudo-code of the deque's methods in the Paper [11], as well as the C library implementation [2]. It features three variables with minor name changes: *head*, *tail* and *split*, being all of these indexes. Furthermore, it has the three standard methods, as well as

[2] `https://github.com/trolando/lace`

three new ones. The *grow_shared* and *shrink_shared* methods are used to alter the size of the shared part of the deque. Only the owner of the deque can call these methods. The last method *pop_stolen* is the method the owner of the deque calls to notify that it does not have tasks to be stolen. Due to the framework being hard to compile and use as well as the minimal documentation, this deque was not tested.

The deques that remained were tested so that we could assess how many operations of pop and push they could perform during a specific time frame. These tests were conducted multiple times using different numbers of threads, as well as testing different rates regarding the number of pushes to pops. Our findings were that there weren't clear gains from a specific one due to all of them being very similarly coded. Their implementations were pretty much the same, apart from the way the logic was ordered. This led to making the necessary changes directly on the PBBS's own scheduler, which like we mentioned in Section 3.2, already uses work-stealing, easing the difficulty of adapting the existing deque.

3.2.2 Split Deque Implementation

As previously mentioned (Section 3.1 and can be seen in Figure 3.1), the lock-free split deque is implemented on top of the array *entries*, which stores nodes of type *WorkStealingJob*. They are coded in the PBBS and are basically functions that take no arguments and don't return anything. In normal scenarios, these are the tasks that workers and thieves remove from the deques and, later, execute. However, when certain circumstances are met, there is also a need to return special codes that represent the outcome of failed removals. These are:

ABORT - returned when steal attempt fails due to the task being removed by another processor.

RACE - returned when a worker tries to remove a task from the private part of its deque but it is empty.

EMPTY - returned when there is a failure to remove work from the public part of a deque due to it being empty, for both the owner of the deque and other thieves.

In order to accommodate these special values, the functions that return a job no longer do. Instead, they return a simple wrapper containing both a *job* and an integer. By default, this integer stores a number that is not linked to any special value, being altered using constants that represent these values when their conditions are met.

This array is complemented by the *privateBottom* and *officialBottom* index variables, which hold integer values, and the *age* structure.

Age is the only variable implemented using *std::atomic*. This is required to ensure a correct flow of execution because other processors may also alter it, making necessary the use of CAS instructions to modify it.

Algorithm 4 PopBottom method of the split deque.

```
    privateBottom ← 0                                          ▷ private field
    entries ← {}                                               ▷ private read-write, public read-only
    officialBottom ← 0                                         ▷ private read-write, public read-only
    age ← {0, 0}                                               ▷ public field
 1: procedure POPBOTTOM
 2:     oBot ← self.officialBottom
 3:     if oBot = 0 then
 4:         wrap.error ← EMPTY
            return wrap
 5:     end if
 6:     oBot ← oBot − 1
 7:     self.officialBottom ← oBot
 8:     std :: atomic_thread_fence(std :: memory_order_seq_cst)
 9:     node ← self.entries[oBot]
10:     oldAge ← age
11:     if oBot > oldAge.top then
12:         self.privateBottom ← oBot
13:         wrap.job ← node
14:         return wrap
15:     end if
16:     self.officialBottom ← 0
17:     self.privateBottom ← 0
18:     newAge.top ← 0
19:     newAge.tag ← oldAge.tag + 1
20:     if oBot = oldAge.top && CAS(age, oldAge, newAge) = SUCCESS then
21:         wrap.job ← node
22:     else
23:         self.age ← newAge
24:         wrap.error ← EMPTY
25:     end if
26:     std :: atomic_thread_fence(std :: memory_order_seq_cst)
27:     return wrap
28: end procedure
```

By resorting to memory thread fences [3], we can make sure that atomic and non-atomic operations are properly sequenced. These are only needed when a worker attempts to remove work from its public deque, preventing the duplication of tasks, where both the worker and a thief would acquire the same job. This can occur if a thief were to attempt a steal at the right time. Instead of using memory fences, another approach could be to synchronize sequentially all atomic accesses, rather than all at once at the memory fence.

One thing that proved a bit challenging during the implementation was thinning the number of memory fences. Correct placement of the memory fences was not hard. The problem is when you try to make them less restrictive or try to remove them. Often a computation was able to execute successfully numerous times, providing accurate results. However, every so often it would do the opposite, resulting mostly in task duplication. Because the program does not crash, or does so at a later time after executing the task twice, it was hard to pinpoint what was at fault. One of the problems that the split

[3] https://en.cppreference.com/w/cpp/atomic/atomic_thread_fence

deque tackles is to reduce the number of communication overhead. To do so, we had to look for places to thin the number of memory fences. This then resulted in the problem mentioned, randomly providing wrong results in the future when testing some other thing.

The final result can be seen in Algorithm 4. There are two memory fences: one on line 9 and another on line 27. The first one is necessary because reading an outdated *age* may induce the worker into entering the *if* clause on line 12, where no CAS is used to prevent task duplication. Another thing to keep in mind is that we cannot simply synchronize *age* by altering the memory order used to load its value. We also need to make sure the changes we made to *officialBottom* are visible to other threads, creating the need for the current memory fence. The last memory fence needed is present on line 27. The code segment from line 17 to line 26 is very dangerous because we set officialBottom to 0 on line 17. This is due to *age* always ending up with its *top* set as 0, regardless of the outcome of the *if* clause on line 21. If the changes to *age* are made visible to other processors while the changes to *officialBottom* are not, thieves would be able to duplicate the tasks that were previously stored there.

3.2.3 The LCWS Implementation

We began our implementation of the LCWS algorithm with a version that is a direct C++ adaptation of the pseudo-code presented in Algorithm 1.

Seeing that PBBS's scheduler already uses WSteal, it already provides the basic structure that LCWS also shares. One thing that look weird was that there was not an actual uniformly random number generator from a C++ library used to choose the victim of thieves (line 26). This was done instead by a hashing function that takes an integer as its argument and by keeping an array of integers (called *attempts*, one for each processor). This integer is incremented by one for each steal attempt. This ensures that the formula used:

$$hash(id) + hash(attempts[id])$$

always returns a different number. The argument *id* refers to the index of the processor. This is a requirement because we do not want every processor to have the same *seed* (if they did, all processors would generate the same sequence of numbers). Lastly, to ensure that thieves are not permanently trying to steal work, the thread is allowed to sleep. Coupon collector's problem [4] is a problem in probability theory. It concerns the expected number of coupons that must be drawn, with replacement, before having drawn each unique coupon at least once. Using the formula

$$\Theta(n\log(n))$$

, we can confirm that after *number_of_deques* * *100* steal attempts, it should have attempted to steal from all other processors and failed, so it sleeps.

[4] https://en.wikipedia.org/wiki/Coupon_collector%27s_problem

Processors do not have a private deque as it would made no sense, since they have to steal from others. Instead all processors access a public vector of deques. They are able to know which deque is theirs because of a *thread_local* integer, the *thread_id*, that is equal to the index of their deque in the vector (acts as the keyword *self* that can bee seen across the algorithm). Any thief can easily attempt to steal from another deque just by having its index. The same applies to the mechanism governing the notifications thieves send to workers.

As we have mentioned before, workers periodically check if they were notified by thieves to make work public (line 3). This is done using a boolean flag, called *targeted*, that thieves modify upon failure to steal from the worker due to not having publicly available work, thus creating an *asynchronous* notification mechanism. Every processor needs one, so we just need a public vector of booleans. Thieves then simply use the same index they used to attempt the steal on the targeted vector to alter the flag (line 29). Even though multiple thieves can concurrently access this flag, all of them try to update the flag to the same state, meaning that no synchronization is required [22].

Upon checking its *targeted* flag, a worker simply invokes the *updateBottom* function and resets the flag (line 4 and 5). A problem with this approach is that thieves are reliant on the workers to finish their current task, which can be anywhere from start to finish, before checking the flag to further decide if it will transfer work to its public part or not. This can turn into a devastating disadvantage if the tasks are very coarse-grained or imply different execution times. This can be seen in Figure 3.2, where one worker is in the middle of executing a rather large task, while having another task stored in the private part of its deque. Meanwhile, a second processor has finished a smaller task and is attempting to steal from it, failing because there are no public tasks on the deque. It keeps notifying the worker to transfer the private task to the public segment of the deque, which won't happen.

3.2.4 Signal-Based Low-Cost Work Stealing

In pursuance of mitigating this problem where thieves are reliant on workers to make their work public, we created a second version of the LCWS algorithm. Signal-Based Low-Cost Work Stealing (SBLCWS) contrasts with the first algorithm when it comes to who checks the *targeted* flag. Thieves now have the freedom to *force* workers to update their public part of the deque at any time.

This is achieved by making use of signaling among threads, provided in the *pthreads* API via the *csignal* package from the Standard C++ Library. By linking a type of signal, meant for user applications (*SIGUSR1* or *SIGUSR2*), to a function, a thread may send this signal to another, forcing it to execute the linked function.

This way, a thief can signal a worker, halting its current execution and triggering the invocation of the function that updates the deque's public part. After the function returns the worker simply resumes execution of what it was previously doing. Meanwhile, the

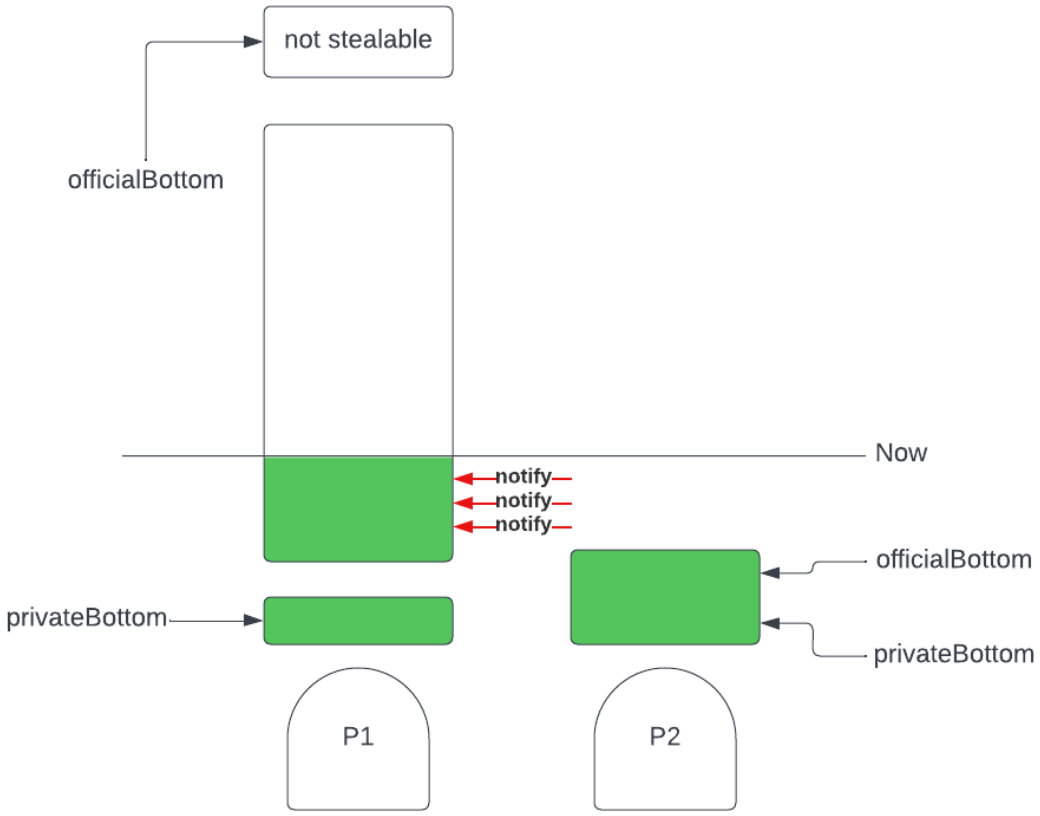


Figure 3.2: Scenario in which two processors are in the middle of a computation. Processor P1 is in the middle of executing a rather large task (when compared with the other tasks present) and has an extra task stored privately. Processor P2 is unable to steal the task and notifies P1 to make it public. P1 can't check the notification until he finishes the current task.

thief never stopped trying to steal work from others while this was happening. This mechanism preserves the notion that only the owner of the deque can operate on the private part of its deque, preventing the rise of concurrency problems.

Signals work great but one should be careful to prevent threads overflowing others with signals. They interrupt the work that threads are doing and bombarding a thread with signals is not good for progressing in the computation. However, one good thing is the fact that standard signals, such as the one used, may only set one signal as pending even when more are received, disregarding the rest. This does not imply that it cannot be a target of this signal flooding but while a thread is answering a signal, it may only have one other signal pending.

To implement this algorithm we require an extra array consisting of native handles extracted from the *std::threads* using *std::thread::native_handle*. These handles are required to invoke the function *pthread_kill*, which is exactly the function necessary for a thread to send a signal to another. It needs both a handle and the type of signal as arguments. One

Algorithm 5 Signal handler function of SBLCWS.

```
    staticDeques ← &deques                                   ▷ public field
    thread_id ← index                                        ▷ thread_local
1:  procedure SIGNALHANDLER(int signal)
2:      staticDeques[thread_id].UPDATEBOTTOM()
3:  end procedure
```

Algorithm 6 Pop method of the split deque.

```
    privateBottom ← 0                                        ▷ private field
    entries ← {}                                             ▷ private read-write, public read-only
    officialBottom ← 0                                       ▷ private read-write, public read-only
1:  procedure POP
2:      pBot ← self.privateBottom
3:      if pBot = self.officialBottom then
          return RACE
4:      end if
5:      pBot ← pBot − 1
6:      node ← self.entries[pBot]
7:      self.privateBottom ← pBot
8:      return node
9:  end procedure
```

detail that almost got overlooked is that the spawned *std::threads* are not the only threads running the computation. We also need to extract the native handle of the main thread, from which the others are created. To do so, we simply have to call *pthread_self*.

Next, we had to create a static function to be linked to the signal *SIGUSR1*. Due to the nature of a static function, we are required to create a static pointer to access the public vector of deques. This pointer is then initialized when *scheduler*'s is created. Processors are then able to access their own deque via the *thread_id* we already mentioned (Section 3.2.3), which happens to be static. The end result function that handles signals is very simple and can be seen in Algorithm 5.

The fact that the worker, after returning from the function invoked by the signal, resumes what it was doing implies that there could be dangerous outcomes in the scenario:

> When a worker is attempting to retrieve work and has only one task in the deque's private part, this worker might have already passed the conditional check of line 3 on the Algorithm 6 (that provides an implementation of the *pop* method of Algorithm 2), that verifies if there is indeed work to be popped. At such time, a thief may signal the worker to transfer work to the public part. This causes the worker to transfer the only task it currently possesses. However, after this update is over, the worker resumes what it was doing, proceeding to remove a task that it should no longer have. This may happen from anywhere in between lines 5 and 9, because *officialBottom* is the only affected variable by the signal, while being read just once (on line 3). A possible representation of this scenario can be visualized in Figure 3.3.

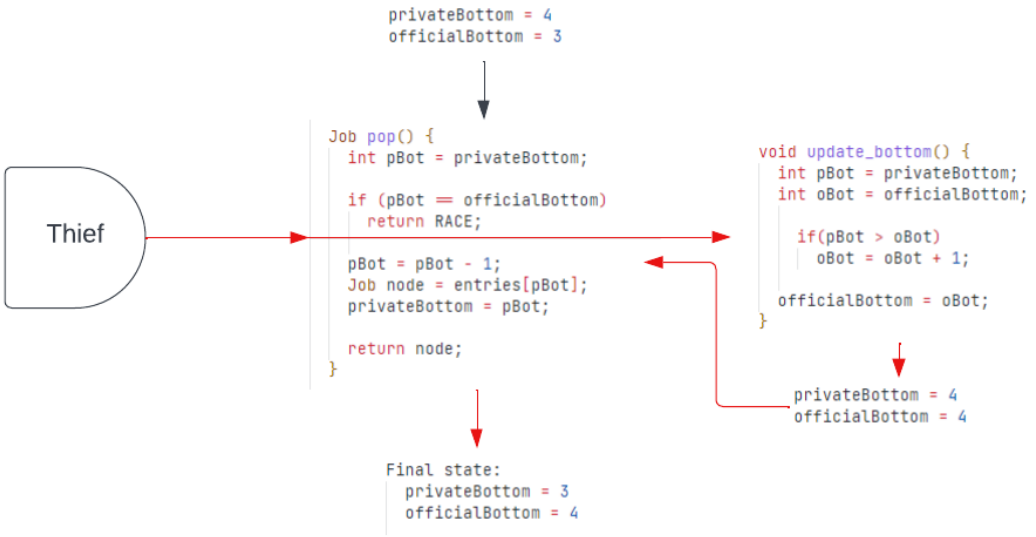


Figure 3.3: Scenario in which two processors are in the middle of a computation using SBLCWS. One processor is currently a thief and signals a worker in the middle of the execution of a *pop* method, right after the conditional check to make sure there is a task stored privately. The worker is thus interrupted and executes *updateBottom*, successfully making the single task public. It returns to previous workflow and removes the task that is no longer private, resulting in an incorrect state where *privateBottom*'s value is lower than *officialBottom*.

To prevent this situation from happening, we could either:

1. ensure that a worker can only transfer work to the public part if he has at least two tasks in its private part or

2. keep the current logic behind work transfers and adopt solutions to prevent incorrect deque states that arise.

Both solutions makes it so that the possible interactions between functions always ends in a correct state. We ended up implementing both, to see which provided better results.

Transferring work to the public part only when having two tasks: The easiest approach to implement is to impose the requirement of having minimum two tasks in order to transfer one to the public part. To accomplish this strategy, we had to alter the conditional check inside the update function to check if

$$privateBottom > officialBottom + 1$$

as shown on line 24 of Algorithm 7. Furthermore, we can reduce the amount of signals a worker receives by tampering with the *targeted* flag. In LCWS, such flag is reset as soon as the worker satisfied the request to update his public part of the deque. Now however,

Algorithm 7 Altered and added methods of the lock-free split deque for the SBLCWS.

```
    privateBottom ← 0                                          ▷ private field
    entries ← {}                                               ▷ private read-write, public read-only
    officialBottom ← 0                                         ▷ private read-write, public read-only
    age ← {0, 0}                                               ▷ public field
    targeted ← true
 1: procedure PUSH(node)
 2:     pBot ← self.privateBottom
 3:     self.entries[pBot] ← node
 4:     self.privateBottom ← pBot + 1
 5:     targeted ← false
 6: end procedure
 7: procedure POPTOP
 8:     oldAge ← self.age
 9:     oldBottom ← self.officialBottom
10:     if oldBottom ≤ oldAge.top then
          return EMPTY
11:     end if
12:     node ← self.entries[oldAge.top]
13:     newAge ← oldAge
14:     newAge.top ← newAge.top + 1
15:     if CAS(age, oldAge, newAge) = SUCCESS then
16:         self.targeted ← false
17:         return node
18:     end if
19:     return ABORT
20: end procedure
21: procedure UPDATEBOTTOM
22:     pBot ← self.privateBottom
23:     oBot ← self.officialBottom
24:     if pBot > oBot + 1 then
          oBot ← oBot + 1
25:     end if
26:     self.officialBottom ← oBot
27: end procedure
28: procedure CHECK
29:     return self.privateBottom > self.officialBottom + 1
30: end procedure
```

thieves can only send signals if the flag is turned off. The flag is turned on immediately before a thief sends a signal. The flag will remain active until:

1. a thief successfully steals the task (line 16);

2. the worker inserts another job on the deque (line 5).

These alone are sufficient to make sure that thieves can request a worker to make tasks public whenever needed. If a thief steals the task, there is a need to reset the flag so that another one can be made public. If it is not stolen, it will either remain there until it is eventually stolen or the owner removes it. The latter option is not a reason to reset the flag. This is because the fact that the worker is removing jobs that are public implies that

Algorithm 8 Stealing phase of the SBLCWS.

```
1:  procedure workMigration
2:      victim ← uniformlyRandomProcessor( )
3:      assigned ← victim.spdeque.popTop( )
4:      if assigned = empty then
5:          if !spdeque[victim].targeted & spdeque[victim].check( ) then
6:              victim.targeted ← true
7:              pthread_kill(thread_handles[victim], SIGUSR1)
8:          end if
9:      end if
10: end procedure
```

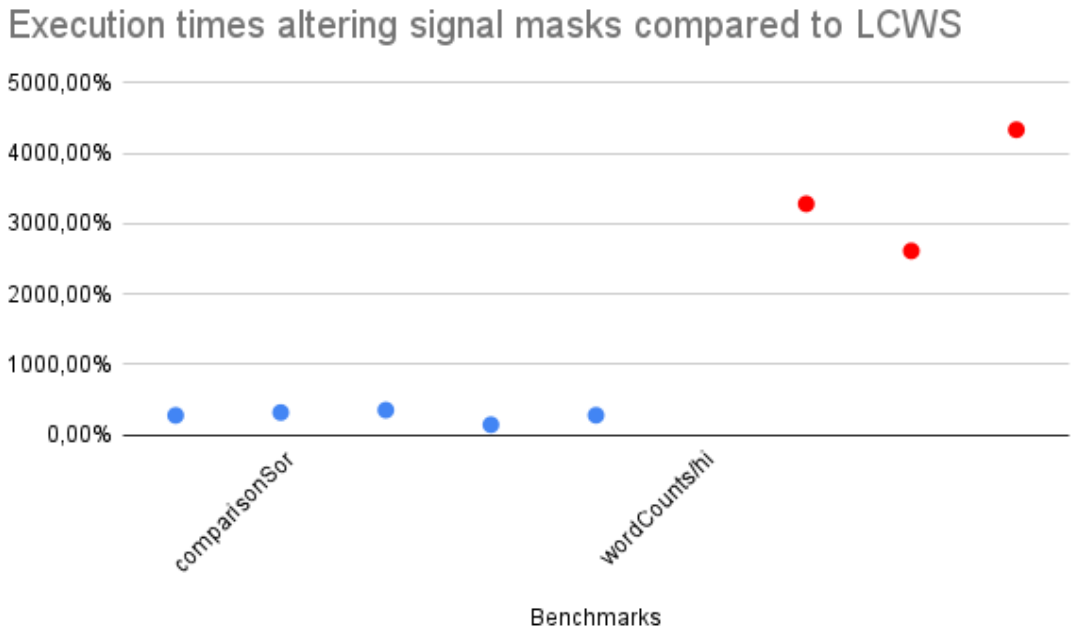


Figure 3.4: The graph shows how much bigger the execution times are by altering signal masks, compared to LCWS using 64 threads. The benchmarks are *Comparison Sort* and *Word Counts*.

the deque's private part is already empty. Therefore, there will only be a need to reset the flag when another task is pushed onto the deque. If there never was a task there in the first place, it means the worker also does not have any, being required to wait for a task to be pushed.

In order to better fit in these changes, the *targeted* flag was moved inside of the lock-free split deque instead of having a vector with all the flags inside the scheduler (previously stated in Section 3.2.3). An extra condition was also added to the stealing phase. There, we are able to see a new call to a procedure called *check*. It is implemented inside the deque on line 28 of Algorithm 7. This method helps further restrict signals by checking if the processor has at least two private tasks. There is no synchronization, in either the added method or the *targeted* flag, imposed which may result in outdated values.

Avoiding incorrect deque states When it comes to preventing the incorrect states that arise with the introduction of signals mentioned in Section 3.2.4, there are different solutions that can be employed to prevent or correct the wrong behavior that arises. A rather direct approach is one where the *signal mask* is altered before and after the *critical* section of the code. Modifying the *signal mask* to block a specific signal implies that said type of signal will not be received by the thread. Only after unblocking the signal will it resume accepting that type of signal. This solution, while logical, unfortunately, does not perform very well. Each thread having its own signal mask, which is changed every time it executes a *pop* on the deque, puts a toll on the machine, as this operation will always be the deque's second most used operation (*push* is naturally number 1). This results in performing always worse than any other solution, by a large margin. An example of such results can be seen in Figure 3.4.

Even though the results are not good, the solution itself provided some insight on how we should proceed. Whatever restriction or condition we force upon the *pop* method will always come down to at least adding two instructions to the method. So we stopped further pursuing this path. Instead of applying restrictions, signals are still received at all times.

To make this work, we have to correct the wrong state that results from the signal arriving during execution of the referred method. This is easily achievable by moving the decrement of the private index to before checking if the private part of the deque has work (line 3 of Algorithm 9). Along with this change, the condition itself has to be slightly altered. This is so that, instead of both indexes having an equal value, the private index's value has to be the lower one. This change does not directly impact performance when the *pop* successfully extracts work. However, when the opposite happens this is not the case. The algorithm still has to execute the decrement, which is "unneeded". A positive remark is that, always after an unsuccessful *pop*, the *popBottom* procedure is run. The latter method, by default, already provides a "reset" to the private index for two of the three possible outcomes the method is structured around. To guarantee the algorithm's correction, i.e. to provide for the last "reset" of the three, the assertion on line 14 of Algorithm 9 is needed to reset the private index.

These changes make it so that, when a *pop* fails, there is either one or two extra instructions, depending on the deque's state when executing *popBottom*. This mild downside is further diminished by the fact that thieves don't even execute these procedures in-between stealing attempts.

After applying these changes, that are indeed correct, the algorithm should work but it does not. Working with signals is not very easy. It is hard to pinpoint exactly where they occur. Even more so on a multithreaded program. This made it hard to debug why deques still ended in incorrect states. After much back and forth, trying all sorts of changes, one by one, to try and understand the error, only one final possibility remained:

The function handling the signal was reading incorrect values.

Algorithm 9 Altered algorithm for the split deque with signals.

```
    privateBottom ← 0                                          ▷ private field
    entries ← {}                                               ▷ private read-write, public read-only
    officialBottom ← 0                                         ▷ private read-write, public read-only
    age ← {0, 0}                                               ▷ public field
 1: procedure POP
 2:     pBot ← self.privateBottom
 3:     pBot ← pBot − 1
 4:     if pBot < self.officialBottom then
          return RACE
 5:     end if
 6:     node ← self.entries[pBot]
 7:     self.privateBottom ← pBot
 8:     return node
 9: end procedure

10: procedure POPBOTTOM
11:     wrap
12:     oBot ← self.officialBottom
13:     if oBot = 0 then
14:         self.privateBottom ← oBot
15:         wrap.error ← EMPTY
          return wrap
16:     end if
17:     oBot ← oBot − 1
18:     self.officialBottom ← oBot
19:     std :: atomic_thread_fence(std :: memory_order_seq_cst)
20:     node ← self.entries[oBot]
21:     oldAge ← age
22:     if oBot > oldAge.top then
23:         self.privateBottom ← oBot
24:         wrap.job ← node
          return wrap
25:     end if
26:     self.officialBottom ← 0
27:     self.privateBottom ← 0
28:     newAge.top ← 0
29:     newAge.tag ← oldAge.tag + 1
30:     if oBot = oldAge.top && CAS(age, oldAge, newAge) = SUCCESS then
31:         wrap.job ← node
32:     else
33:         self.age ← newAge
34:         wrap.error ← EMPTY
35:     end if
36:     std :: atomic_thread_fence(std :: memory_order_seq_cst)
37:     return wrap
38: end procedure
```

One thing that definitely made us overlook that this type of problem was a possibility comes from how poor the *C++* documentation at times is. One such example is *std::atomic_signal_fence* ([5]) which, by looking at the single paragraph description, supposedly could be used to fix this issue. The way this problem was solved however, was by the usage of the *volatile* keyword. This problem comes from compiler optimizations. The compiler may reorder instructions so that the code is more efficient. In most scenarios, program correctness is maintained, unless some outside program tampers with shared resources. Turns out that even though these signals between threads are all part of the same program, their actions are not taken into account by the compiler when optimizing all the other methods. One of the reasons for the *volatile* keyword's existence is actually signal handling. By making both indexes volatile the algorithm now works as intended. The public index also needs to be altered because the possible changes caused by the signal handler may not be reflected immediately after it returns to the previous workflow.

3.3 Hybrid Work Sharing

Upon testing of all these receiver-initiated approaches, we attempted an hybrid one, mixing both LCWS and SBLCWS with WShare, employing a spreading mechanism similar to the one used in The Flexible Work Stealing and Sharing algorithm presented in Paper [20]. This mechanism works on a set of heuristics, from which we extracted two that were mentioned in the paper but were slightly altered.

3.3.1 Low-Cost Work Stealing with Work Sharing

Algorithm 10 presents the algorithm we developed that combines WSteal and WShare, having both a stealing phase and a spreading phase where workers try to give work to thieves. This new spreading phase is not executed on a loop like the stealing one, but instead *on demand*. It is enabled by a set of heuristics that dictate in which scenarios it should be triggered.

One characteristic that makes LCWS unique is that thieves notify workers to make jobs public. This is still the case in Algorithm 10 but there are some slight differences. Now, thieves do not necessarily notify their victim and instead choose another processor at random to notify (line 41). Workers still periodically check to see if they were notified (line 4) and if so, they will attempt to give work to thieves. Upon beginning the spreading phase (line 8 to 15), the worker chooses a processor at random (line 5). Before anything else, it has to make sure that the processor chosen is in fact in need of work (line 6). This means that the worker will only attempt to spread tasks to thieves. After this checkup is finished, it will attempt to make work public (line 7) and, if it does, it sends a notification back to the chosen thief (line 9). Meanwhile, the thief may have gotten lucky and succeeded in a steal attempt and this notification serves no purpose. Even though this thief

[5] https://en.cppreference.com/w/cpp/atomic/atomic_signal_fence

Algorithm 10 Hybrid WShare algorithm.

```
1: procedure SCHEDULER
2:     while computation not terminated do
3:         if VALIDNODE(assigned) then
4:             if self.spreading then
5:                 victim ← UNIFORMLYRANDOMPROCESSOR( )
6:                 if vitim.state == IDLE then
7:                     if self.spdeque.UPDATEBOTTOM( ) then
8:                         self.spreading ← HEURISTIC[0]
9:                         victim.state ← self.thread_id
10:                    else
11:                        self.spreading ← HEURISTIC[1]
12:                    end if
13:                else
14:                    self.spreading ← HEURISTIC[1]
15:                end if
16:            end if
17:            enabled ← EXECUTE(assigned)
18:            if LENGTH(enabled) > 0 then
19:                assigned ← enabled[0]
20:                if LENGTH(enabled) = 2 then
21:                    self.spdeque.PUSH(enabled[1])
22:                end if
23:            else
24:                assigned ← self.spdeque.POP( )
25:                if assigned = RACE then
26:                    assigned ← self.spdeque.POPBOTTOM( )
27:                end if
28:            end if
29:        else
30:            self.WORKMIGRATION( )
31:        end if
32:    end while
33: end procedure

34: procedure WORKMIGRATION
35:     if self.state ≠ IDLE then
36:         victim ← self.state
37:     else
38:         victim ← UNIFORMLYRANDOMPROCESSOR( )
39:     end if
40:     assigned ← victim.spdeque.POPTOP( )
41:     target ← UNIFORMLYRANDOMPROCESSOR( )
42:     if assigned = ABORT then
43:         target.spreading ← HEURISTIC[2]
44:         self.state ← IDLE
45:     else
46:         if assigned = EMPTY then
47:             target.spreading ← HEURISTIC[3]
48:             self.state ← IDLE
49:         else
50:             target.spreading ← HEURISTIC[4]
51:             self.state ← WORKING
52:         end if
53:     end if
54: end procedure

55: function VALIDNODE(node)
56:     return node ≠ EMPTY and node ≠ ABORT and node ≠ NONE
57: end function
```

no longer needs to acquire a task, others might. If, however, this thief was still looking for work, it is now aware that a worker notified him, removing the need to blindly look for processors with work publicly available (line 36 instead of line 38).

In order to accommodate these changes, each processor now possesses another two variables in addition to the lock-free split deque. Starting off with the *state* flag, it is used to keep track of the status of a worker. One can either be *WORKING* or *IDLE*. Even though its primary function is to store the state of a worker, it is also used to keep the identifier of a worker who attempted to spread work to the flag's owner, functioning like a notification. The second flag labeled as *spreading* flag simply has two possible values and so is implemented as a boolean. This flag acts as a replacement for the previous *targeted*, and indicates whether this worker should attempt to spread work. This value can be altered depending on the result of a previous spread attempt or by a thief that selects another worker at random and updates his flag, depending on the outcome of its last steal attempt. The resulting state arisen from such update is dictated by a set of heuristics that comprise all possible outcomes in these scenarios, linking a value to each one.

When a worker that is not empty of work and has the spreading flag set as true, it attempts to spread work to an idle processor (without work), choosing one at random. Like it has been mentioned, one possible scenario is that the processor chosen is not idle thus the worker will not spread work and, depending on the heuristics defined, will or not attempt a spread cycle after its current task. However when it does succeed with the spreading, the thief will have its *state* flag altered so that it points to the worker who spread work. This way, when the thief checks that its flag does not comprise the *IDLE* value, it will attempt to steal from the worker who it points to, instead of doing so at random.

Contrary to the *spreading* flag, which may only store two different values and so, allowed being implemented using a boolean variable, the *state* flag has $N + 2$ possible scenarios (N = number of processors). The logical thing to do was to implement it as an integer. Doing so, it allows a direct usage as an index for the vector of processors, necessary when a thief is the target of spreading. This leaves the *WORKING* or *IDLE* states to be assigned to any number. The easiest way to do this is to either use negative numbers or large enough values that will never be used as the number of processors. We chose the former.

3.3.2 Heuristics

As it has already been mentioned, heuristics are a set of values that dictate whether a worker will attempt to spread work depending on the outcome of two operations: stealing and spreading phase. They were inspired by the ones in Paper [20] and are tuples of five values: **(S1, S2, S3, S4, S5)**. Each value defines if a specific scenario is a reason to spread work. We will now describe what these five situations are.

The spreading routine has to address two possible scenarios, meaning that the heuristics have two preset values assigned to each of the scenarios. After randomly choosing a processor:

S1 it is idle;

S2 it is not idle.

The stealing phase has an extra nuance and thus comprises three possible scenarios and, following the same logic, the heuristic has three preset values used for each. After attempting to steal work and, yet again, choosing another random processor whose spreading flag will receive the update:

S3 the steal attempt may abort due to being raced by another thief;

S4 the attempt failed because there is no publicly available work but the worker indeed has work in the deque's private part;

S5 it succeeded;

The union of the values associated with all these scenarios thus creates an heuristic: (S1, S2, S3, S4, S5). The possible values for each of these are:

TRUE - indicates that the processor should spread work;

FALSE - indicates that the processor should not spread work;

3.3.2.1 Tit-for-tat heuristic

The values for this heuristic are (F, F, T, T, T). This heuristic is rather basic as what it basically dictates is that whenever there is a steal attempt, the thief will set the *spreading* flag of another random processor as *TRUE*, regardless of the steal's outcome. When it comes to spreading, a worker will only attempt to spread one time and resets the flag to *FALSE* in all scenarios. The conjunction of these make it so that the number of spreading attempts will always be lower or equal to the number of steal attempts. It can, and most likely will, be lower because we can not forget that multiple thieves may target the spreading flag of the same processor until he begins the spreading phase.

3.3.2.2 Continuous tit-for-tat heuristic

The values for this heuristic are (T, F, T, T, T). This heuristic provides the same behavior as before when it comes to the stealing scenarios. Regardless of the outcome, a thief will always set a processors *spreading* flag to true. The difference is related to the success of spreading work. When a worker succeeds in spreading work, the flag remains as *TRUE*. This implies that as long as the worker has work to spread and keeps randomly selecting *IDLE* processors, it will keep attempting to spread work. This does not mean that it will

Algorithm 11 Signal handler function.

staticScheduler ← &*scheduler* ▷ public field

```
1: procedure SIGNALHANDLER(int signal)
2:     staticScheduler->SPREAD()
3: end procedure
```

spread all its work in one go because workers only get one spreading cycle in between executing jobs. This heuristic makes it so that there are more spreading attempts than the former.

3.4 Signal-Based Low-Cost Work Stealing with Work Sharing

Just like for the LCWS algorithm, we experimented with introducing signals into the WShare solution. We partially replicated the changes made to LCWS when transformed into SBLCWS. The first change is that we no longer need the static pointer that points to the vector of deques. Previously, we wanted to invoke *updateBottom*, which is implemented in the lock-free split deque. Now we want to call a *spreading* routine. Instead, we want to keep a static pointer that points to the scheduler itself. The resulting function responsible for handling signals can be seen in Algorithm 11. The *thread_local id* is no longer there as it is only needed inside the spread function and not required when invoking the procedure itself.

When it comes to the changes made to the lock-free split deque in SBLCWS, we removed all the restrictions imposed by the *targeted* flag while keeping all the changes that made both versions of SBLCWS work. It would not make sense to restrict making jobs public just because there are already public jobs. Mixing WShare with WSteal is meant to allow workers to make proactive transfers. Workers also will not be interrupted non-stop as we still use the *check* function, meaning a worker will need to have two private tasks to be signaled. This is also a good thing, seeing that we removed most restrictions that were stopping signals. Workers are always able to keep one task for themselves private (apart from the one that they might be executing), even if they get targeted numerous times.

We kept the *check* function, because it is called during a stealing phase. Just like previously in LCWS, we inserted signals to allow transfers during execution of tasks. They also allowed multiple transfers to happen during a full cycle of the algorithm instead of just one. What this means here is that there is no reason for the heuristic *Continuous tit-for-tat heuristic*. Now that the spreading routine can only be invoked by thieves, there is no point in leaving the *spreading* flag turned on in between cycles. As such, thieves simply turn the flag on so that others cannot send any more signals. The worker that received the signal will then perform the spreading procedure and reset the flag.

Algorithm 12 Altered steal phase for the SBLCWS with WShare solution.

```
 1: procedure workMigration
 2:     if self.state ≠ IDLE then
 3:         victim ← self.state
 4:     else
 5:         victim ← uniformlyRandomProcessor( )
 6:     end if
 7:     assigned ← victim.spdeque.popTop( )
 8:     target ← uniformlyRandomProcessor( )
 9:     if !target.spreading then
10:         if assigned = abort then
11:             spread ← heuristic[2]
12:             self.state ← IDLE
13:         else
14:             if assigned = empty then
15:                 spread ← heuristic[3]
16:                 self.state ← IDLE
17:             else
18:                 spread ← heuristic[4]
19:                 self.state ← WORKING
20:             end if
21:         end if
22:         if spread & target.spdeque.check( ) then
23:             target.spreading ←TRUE
24:             pthread_kill(thread_handles[target],SIGUSR1)
25:         end if
26:     end if
27: end procedure
```

4

Evaluation

In this chapter, we will start by explaining what are our goals are and how we want to achieve them. This implies the methodology behind the tests we conducted for all the algorithms created and the canonical Work Stealing (WSteal) algorithm. These tests were composed of the benchmarks provided by Problem Based Benchmark Suite (PBBS), and so, we have listed all the ones used and what they do. The next sections in this chapter consist of comparisons of the algorithms, showing some key graphs that illustrate the logic behind the advantages and disadvantages of each algorithm. We also present tables of gains and losses, in terms of execution times, for these comparisons, across all machines used. We end the chapter by providing a table that should provide a mild suggestion of what algorithm is more suitable for each benchmark, disregarding the number of threads used.

4.1 Goals

We structured our evaluation according to the **Goal, Question, Metric** methodology [8]. As such, we began by defining the set of goals we sought to achieve:

1. Compare our implementation of Low-Cost Work Stealing (LCWS) against a implementation of the canonical WSteal algorithm;
2. Evaluate the gains of mixing Work Sharing (WShare) with WSteal compared with the single WSteal solutions;
3. Evaluate the relative performance of the signal-based versions against the original (signal-free) versions.

To reach each of these goals, for each one of them, we intend to answer the following questions:

- What is the performance of each algorithm under evaluation?
- What are the gains obtained by their optimizations?

Lastly, so that we can answer these questions, we defined a set of metrics that allow us to compare the algorithms:

- Execution time;
- Number of fences;
- Number of Compare-And-Swap (CAS) operations;
- Number of steals;
- Number of missed steals;
- Number of tasks removed from owned deque;
- Number of missed steals when there is private work (split deque algorithms);
- Number of tasks removed from public section of owned deque (split deque algorithms);
- Number of tasks transferred from private to public section of the deque (split deque algorithms);
- Number of successful steals that happened due to sender-initiated mechanism (WShare algorithms).

4.2 Methodology

The experiments conducted in order to capture the mentioned data made use of the PBBS. This benchmark suite features different types of algorithms, specifically designed to make possible comparisons between different algorithms or implementations. Furthermore, each one provides multiple data sets, allowing even broader diversity. It allows the user to define the number of threads (workers) to use during testing. These threads are managed by the Operating System, which handles all the scheduling and mapping of the threads to the hardware cores.

The benchmarks that we tested were:

- Integer Sort: sorts a sequence of integers. Tested on four data sets.
- Comparison Sort: sorts a sequence of elements of any (uniform) type using a given comparison function over the elements. Tested on five data sets;
- Remove Duplicates: returns the input sequence with the duplicates removed. Tested on three data sets;
- Histogram: given a sequence of integers and bucket-count n, generates a histogram of the integers in the range $[0{:}n]$. Tested on five data sets;

Table 4.1: Machines used in the collection of data.

Name	CPU	Cores/Threads	Memory
Intel 12-24	2 x Intel Xeon E5-2620 v2	12/24	64 GiB DDR3 1600 MHz
AMD 32-64	4 x AMD Opteron 6272	32/64	64 GiB DDR3 1600 MHz
Intel 16-16	2 x Intel Xeon E5-2609 v4	16/16	32 GiB DDR4 2400 MHz

- Word Counts: counts the number of each word in a string. Tested on three data sets;
- Inverted Index: returns an inverted index given a string of documents. Tested on two data sets;
- Classify: predicts labels for feature vectors based on a training vectors with attached labels. Tested on two data sets;
- Spanning Forest: generates a spanning forest of an undirected graph. Tested on three data sets;
- Breadth First Search: returns a breadth-first-search tree from a given vertex in a graph. Tested on three data sets;
- Maximal Matching: returns a maximal matching for an undirected graph. Tested on three data sets;
- Maximal Independent Set: returns a maximal independent set for an undirected graph. Tested on three data sets;
- Nearest Neighbors: returns the k nearest neighbors for points in 2D and 3D. Tested on six data sets;
- Convex Hull: generates the convex hull for a set of points in 2 dimensions. Tested on three data sets;
- Delaunay Triangulation: returns the Delaunay triangulation of points in 2D. Tested on two data sets;
- Delaunay Refine: adds points to a Delaunay triangulation in 2D, making the result have no small angles; Tested on two data sets;
- 2D Range Query: given a set of points and a set of rectangles, returns a count of the number of points in each rectangle. Tested on two data sets.

The benchmarks were ran for each of the algorithms, using different number of threads, following the pattern of being a power of two until the number of hardware threads of the machine. For this, three machines were used, each with different specifications and hardware depicted in Table 4.1.

Each benchmark is ran for 10 rounds, from which we extracted the geometric mean and standard deviation, regarding execution time. Geometric mean ensures that extreme values do not skew the average as much. In separate rounds of testing, we acquire all the other metrics related to the algorithms.

For the sake of readability and conciseness, we do not include graphs of all experiments in this chapter. Instead, to strengthen the narrative, we selected some graphs representative of the experiments and opted to present the complete set of graphs in Appendix A.

4.3 LCWS vs WSteal

In this section, we analyze and compare the performances of both algorithms. As we skim through the results, one thing becomes clear. The problems that the LCWS algorithm was meant to mitigate are indeed accomplished. The number of memory fences is extremely reduced. As we can see in Figure 4.1, all benchmarks result in an average ratio of 0.0025 memory fences compared to the WSteal algorithm. This average comprises all the different numbers of threads used so it does not reflect the average of each of them independently (the ratio increases along with the number of threads). A few outliers that surpass this number by quite a bit can be seen. The benchmarks were: *Classify*, *Maximal Independent Set*, *Delaunay Triangulation*.

When it comes to CAS operations, LCWS also greatly outperforms the WSteal algorithm. Results in Figure 4.2 show that there aren't as many clear outliers but there is a tendency for the ratio to hang around 0.3. It also clearly shows that LCWS starts off having considerably less operations than WSteal and this gap shrinks a little as we scale the number of threads.

Even though these numbers look positive, they do not necessarily translate into an increase in performance. Due to the nature of the LCWS algorithm, it features a reduced number of successful steals as shown in Figure 4.3. This disparity with the WSteal algorithm only becomes wider as the number of threads scales. Having the need to notify a worker to update his deque delays the actual stealing of work. Stealing is what makes the computation move forward. Delayed steals directly correlate to threads being idle and this affects the computation as a whole because it functions like a chain reaction.

Another problem related to the stealing of work is when a worker transfers work to the public part of the deque and it ends up not getting stolen. There is a hit in performance not only because of the transfer process but also because of the communication overhead implied by the removal of work from the public part of the deque. The amount of times this happens also scales with the number of threads.

The last problem linked to the scalability of threads is related to memory fences. In both algorithms, there are no fences present when it comes to stealing. This means that the increase in stealing attempts, directly implied by the increase in the number of threads, should not alter much the number of memory fences. While this remains true

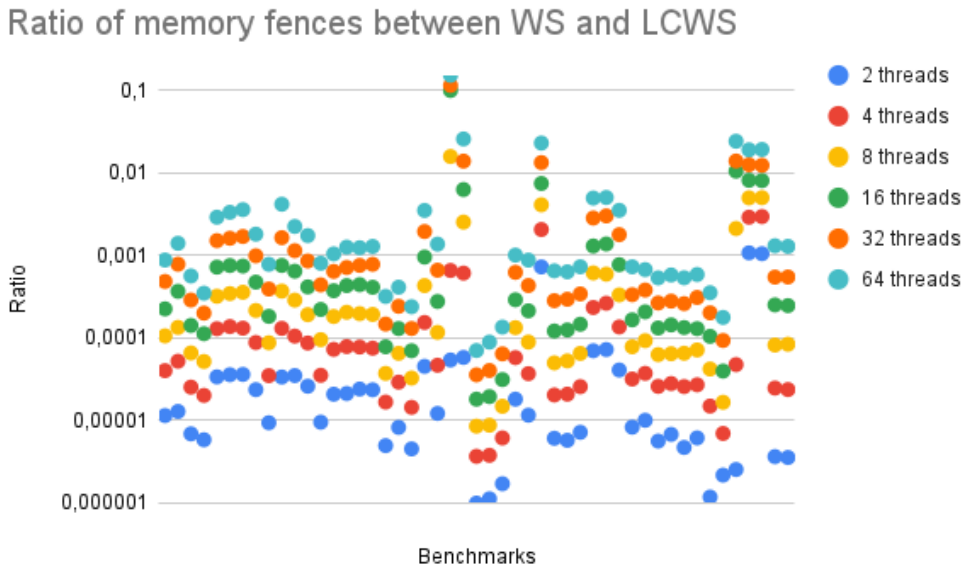


Figure 4.1: Ratio of memory fences between LCWS and WSteal algorithms ran in machine *AMD 32-64*.

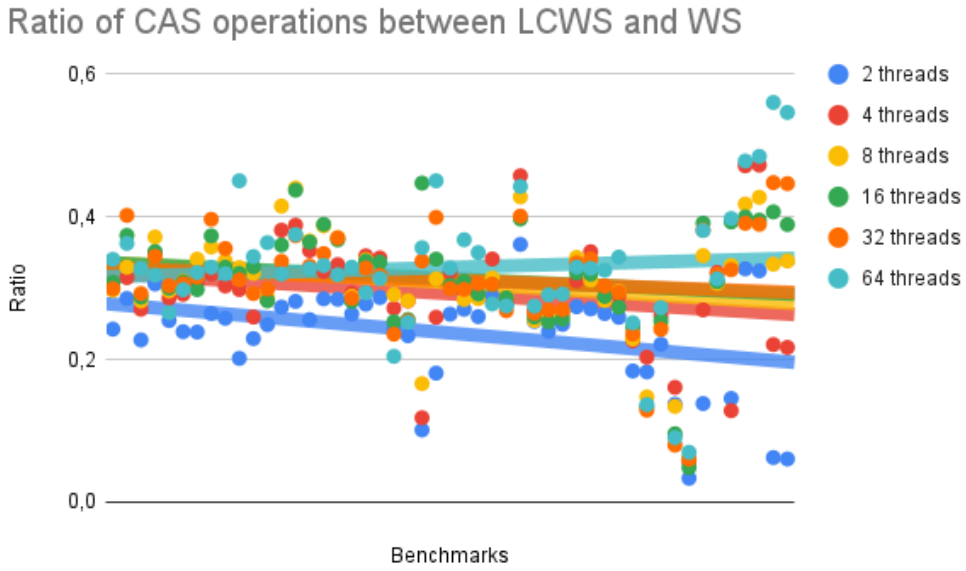


Figure 4.2: Ratio of CAS operations between LCWS and WSteal algorithms ran in machine *AMD 32-64*.

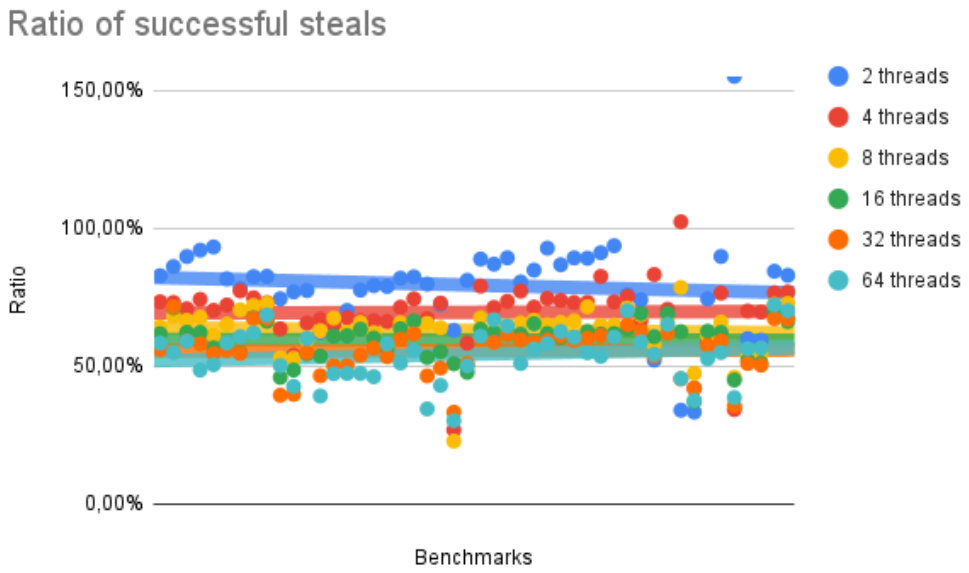


Figure 4.3: Ratio of successful steal operations between LCWS and WSteal algorithms ran in machine *AMD 32-64*.

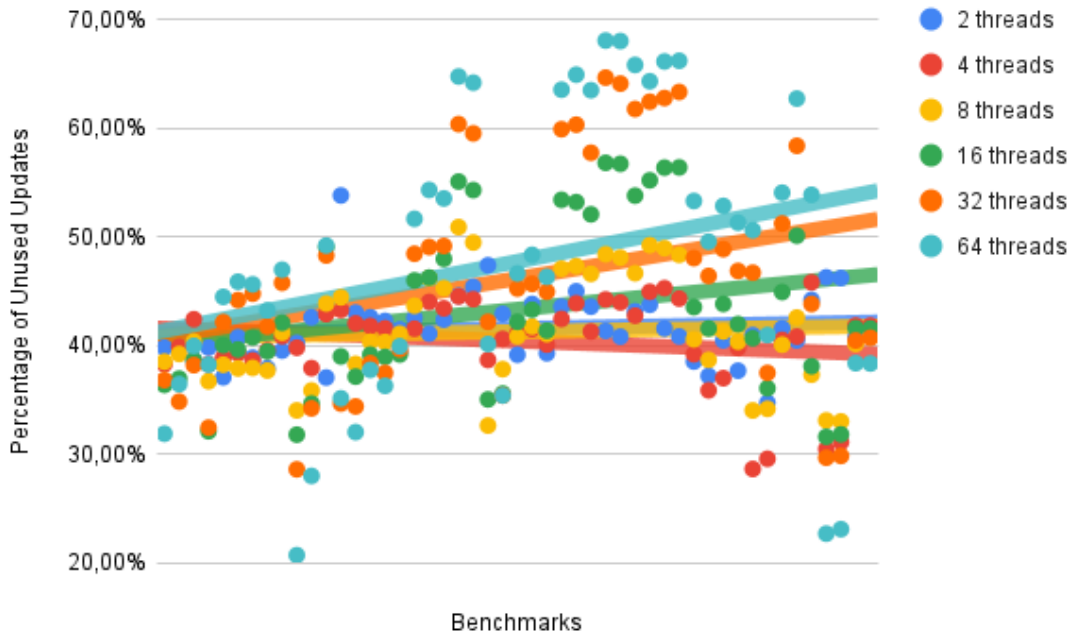


Figure 4.4: Percentage of unused updates of the deque's public part in LCWS ran in machine *AMD 32-64*.

for the WSteal algorithm, the same does not happen with LCWS. The increased number of steal attempts infers an increase in the already mentioned unused deque transfers (see Figure 4.4), and therefore, an increased number of accesses to the public part of the deque by the owner, which requires communication. Even though the number of memory fences remains almost unchanged for the WSteal algorithm, the number of memory fences doubles when the number of threads doubles for the LCWS algorithm (perceivable in Figure 4.1). Regardless, the canonical WSteal algorithm features immensely more memory fences than LCWS.

These problems explain the diminishing of the overall gains of LCWS over WSteal as the number of threads scales. LCWS starts out on top when it comes to computations using 1 thread and this is where the average of gains is the highest. As the number of threads starts increasing, this average begins to drop and by the time the number of threads is equal to the machine's number of cores, LCWS starts losing. Afterwards, computations become more unstable, exhibiting high deviations and resulting in big hits on performance. Many of the benchmarks performed slower than they did with fewer threads.

Tables 4.2, 4.3 and 4.4 show the gains and losses of the different machines across all benchmarks. It usually has smaller percentages of gains around 10% and may go in some outliers up to 30%. The problem is that there are some benchmarks where it loses by a lot. This can be seen on the *Histogram* and *Integer Sort* benchmarks, happening on all machines. The reasons for this to be happening is most likely that these benchmarks have very low execution times. LCWS has a rather slow start at balancing the load and so, it is not the best option for short computations. In the canonical WSteal, a worker starts with a task and begins executing it. As soon as it creates another task, thieves are able to steal it, even if the worker has not finished its execution. Then there will be more workers to be stolen from by processors who are still idle and so on like a chain reaction, until the load is decently distributed (a high percentage of processors are not idle). In LCWS, thieves are only able to steal after the worker finishes its current task. It will only make one task public per step of the computation (considering it has tasks and is notified every step). This will happen across all workers and delay the chain reaction. This slow start is is another reason for why it starts losing against WSteal across the majority of benchmarks when the number of threads starts being too high.

4.4 LCWS vs Signal-Based Low-Cost Work Stealing (SBLCWS)

4.4.1 Signal integration solutions

As we established before, we presented two different solutions when it comes to the integration of signals in the LCWS algorithm. Before we delve right into the comparison between signal and signal-free solutions, we are will explain the differences between the results gathered from both signal solutions, as they differ. Throughout this section we

					AMD 32-64			
Benchmark	**Data Set**	**1**	**2**	**4**	**8**	**16**	**32**	**64**
integerSort/parallelRadixSort	randomSeq_100M_int	-1,50%	-24,80%	9,30%	-4,70%	-0,40%	-1,20%	-108,10%
	exptSeq_100M_int	-1,70%	0,10%	2,10%	-1,00%	-0,40%	-69,90%	-120,20%
	randomSeq_100M_int_pair_int	4,20%	0,40%	0,70%	3,40%	-9,30%	-41,90%	-49,80%
	randomSeq_100M_256_int_pair_int	0,50%	-0,60%	-2,80%	21,70%	-10,40%	-4,40%	-35,10%
comparisonSort/sampleSort	randomSeq_100M_double	-3,00%	-2,00%	-2,90%	-1,10%	-2,50%	-1,30%	-11,00%
	exptSeq_100M_double	-2,50%	-1,90%	-0,30%	1,20%	1,60%	-3,60%	-11,60%
	almostSortedSeq_100M_double	-4,70%	-3,90%	-6,30%	-3,60%	-2,10%	-2,10%	-9,50%
	randomSeq_100M_double_pair_double	-2,50%	-3,40%	0,90%	2,20%	0,50%	-1,70%	-280,10%
	trigramSeq_100M	5,10%	5,70%	5,10%	4,50%	3,20%	-1,50%	-229,20%
removeDuplicates/parlayhash	randomSeq_100M_int	1,00%	-2,60%	8,20%	-2,80%	-17,90%	-33,90%	-201,60%
	exptSeq_100M_int	1,40%	-2,20%	5,90%	0,60%	-3,40%	-7,00%	-255,30%
	trigramSeq_100M	-1,10%	-4,70%	3,20%	0,90%	-2,80%	-13,80%	-389,80%
histogram/parallel	randomSeq_100M_256_int	-0,90%	-1,50%	-37,50%	-27,00%	-50,00%	-47,10%	-50,00%
	randomSeq_100M_100K_int	-1,20%	1,50%	0,20%	-9,30%	-21,30%	-102,80%	-53,40%
	randomSeq_100M_int	0,20%	-4,40%	12,90%	-7,60%	-3,20%	-44,80%	-114,80%
	exptSeq_100M_int	0,10%	-4,90%	7,30%	-2,30%	-17,80%	-34,40%	-159,30%
	almostEqualSeq_100000000	0,30%	-4,20%	3,60%	2,70%	-10,80%	-29,10%	-80,50%
wordCounts/histogram	trigramString_250000000	9,90%	8,30%	11,10%	11,60%	6,30%	4,60%	-129,70%
	etext99	10,70%	6,40%	14,60%	8,90%	7,10%	-14,60%	-176,00%
	wikipedia250M,txt	11,80%	6,60%	9,80%	12,80%	5,10%	-2,40%	-77,90%
invertedIndex/parallel	wikisamp,xml	7,80%	7,70%	10,00%	9,30%	10,90%	-1,20%	-105,10%
	wikipedia250M,txt	10,00%	8,50%	9,40%	10,20%	8,70%	9,90%	-40,90%
classify/decisionTree	covtype,data	1,20%	1,70%	3,50%	-1,60%	-17,00%	-48,10%	-163,30%
	kddcup,data	0,70%	0,40%	0,40%	-0,10%	0,60%	1,40%	-49,80%
spanningForest/ndST	randLocalGraph_E_10_20000000	-0,60%	0,70%	-2,40%	5,70%	-0,40%	5,70%	-5,70%
	rMatGraph_E_12_16000000	1,50%	3,70%	1,30%	8,60%	2,00%	-4,50%	3,10%
	2Dgrid_E_64000000	0,20%	-0,70%	-13,10%	-2,60%	-1,50%	-0,10%	0,60%
breadthFirstSearch/backForwa	randLocalGraph_J_10_20000000	19,20%	17,00%	25,30%	15,80%	20,90%	6,10%	-10,10%
	rMatGraph_J_12_16000000	13,80%	16,70%	18,50%	10,40%	18,40%	8,90%	-15,40%
	3Dgrid_J_64000000	10,90%	9,40%	2,50%	-8,10%	-18,20%	-56,70%	-363,80%
maximalMatching/incrementalI	randLocalGraph_E_10_20000000	16,20%	14,60%	21,00%	9,70%	8,40%	3,30%	-1,40%
	rMatGraph_E_10_20000000	14,00%	13,20%	7,70%	10,40%	6,70%	6,60%	-9,70%
	2Dgrid_E_64000000	10,30%	10,30%	14,60%	4,90%	7,00%	4,80%	-2,30%
maximalIndependentSet/increr	randLocalGraph_JR_10_20000000	14,00%	10,40%	17,60%	7,60%	1,30%	-5,60%	-167,80%
	rMatGraph_JR_12_16000000	12,50%	14,20%	11,30%	7,70%	4,40%	-3,90%	-270,30%
	3Dgrid_JR_64000000	13,70%	12,90%	0,80%	7,90%	6,30%	-6,40%	-120,30%
nearestNeighbors/octTree	2DinCube_10000000	7,40%	6,80%	12,70%	4,10%	6,40%	7,70%	-4,00%
	2Dkuzmin_10000000	7,80%	7,00%	9,20%	5,80%	10,50%	5,80%	-10,30%
	3DinCube_10000000	4,70%	4,20%	5,50%	3,00%	4,10%	3,50%	-8,70%
	3DonSphere_10000000	5,70%	5,00%	1,90%	3,20%	8,00%	4,30%	-9,40%
	3DinCube_10000000	2,30%	1,50%	7,00%	1,70%	-0,20%	0,70%	-16,40%
	3Dplummer_10000000	2,50%	1,10%	7,50%	3,40%	2,40%	1,70%	-2,90%
convexHull/quickHull	2DinSphere_100000000	2,90%	9,70%	-3,90%	-7,50%	1,50%	-7,80%	-13,20%
	2Dkuzmin_100000000	4,80%	3,10%	-15,00%	14,90%	-2,30%	-16,30%	-34,20%
	2DonSphere_100000000	3,00%	-1,30%	25,20%	10,60%	-1,40%	-8,00%	-11,90%
delaunayTriangulation/increme	2DinCube_10M	1,00%	-0,40%	7,30%	1,50%	-5,80%	-14,80%	-69,40%
	2Dkuzmin_10M	2,00%	0,10%	-0,30%	-5,60%	-6,00%	-14,80%	-62,00%
delaunayRefine/incrementalRe	2DinCubeDelaunay_5000000	4,80%	2,80%	-2,70%	2,10%	-1,10%	-3,60%	-55,10%
	2DkuzminDelaunay_5000000	4,30%	2,80%	1,80%	2,90%	-0,10%	-6,40%	-72,40%
rangeQuery2d/parallelPlaneSv	2DinCube_10M	7,80%	14,40%	13,00%	1,10%	3,90%	-1,80%	-58,10%
	2Dkuzmin_10M	10,70%	3,40%	2,80%	3,90%	2,50%	-0,70%	-70,20%

Table 4.2: Gains and losses of LCWS over WSteal when executed on machine *AMD 32-64*.

				Intel 12-24			
Benchmark	**Data Set**	**1**	**2**	**4**	**8**	**16**	**24**
integerSort/parallelRadixSort	randomSeq_100M_int	0,30%	0,80%	0,00%	0,00%	-4,70%	-4,70%
	exptSeq_100M_int	-0,20%	4,60%	0,90%	0,00%	-4,30%	-1,90%
	randomSeq_100M_int_pair_int	0,30%	0,80%	1,60%	-1,00%	-4,70%	-3,70%
	randomSeq_100M_256_int_pair_int	-0,20%	1,20%	0,40%	-0,70%	2,00%	0,00%
comparisonSort/sampleSort	randomSeq_100M_double	-0,10%	-0,50%	1,10%	0,50%	-13,10%	-0,20%
	exptSeq_100M_double	0,00%	1,00%	-0,50%	1,00%	-9,90%	-0,70%
	almostSortedSeq_100M_double	0,00%	-0,70%	1,00%	0,80%	-8,70%	-0,20%
	randomSeq_100M_double_pair_double	0,90%	0,90%	1,20%	1,20%	-12,80%	-3,80%
	trigramSeq_100M	10,20%	9,50%	8,50%	9,30%	-5,50%	-0,20%
removeDuplicates/parlayhash	randomSeq_100M_int	1,60%	1,40%	0,60%	0,30%	-11,60%	-2,70%
	exptSeq_100M_int	0,60%	-1,30%	-1,70%	0,50%	-12,70%	1,60%
	trigramSeq_100M	-0,20%	-3,10%	-9,50%	-23,90%	-50,10%	-51,30%
histogram/parallel	randomSeq_100M_256_int	0,00%	0,00%	-20,00%	-76,90%	-77,80%	-55,60%
	randomSeq_100M_100K_int	0,10%	-0,50%	-1,60%	-3,70%	-5,30%	-15,80%
	randomSeq_100M_int	-0,10%	-2,50%	-2,70%	-9,60%	-17,70%	-26,10%
	exptSeq_100M_int	1,00%	-1,60%	-3,60%	-8,80%	-22,10%	-23,30%
	almostEqualSeq_100000000	-1,50%	-0,30%	-1,60%	-4,40%	-19,30%	-21,10%
wordCounts/histogram	trigramString_250000000	8,80%	9,40%	9,00%	7,90%	-6,40%	0,30%
	etext99	11,20%	10,90%	11,50%	10,70%	-3,20%	0,70%
	wikipedia250M,txt	11,40%	11,90%	12,20%	11,50%	-1,70%	1,90%
invertedIndex/parallel	wikisamp,xml	4,80%	5,90%	4,70%	5,50%	-4,80%	2,40%
	wikipedia250M,txt	7,00%	7,30%	7,70%	6,50%	-4,20%	1,70%
classify/decisionTree	covtype,data	0,60%	1,00%	0,60%	-1,80%	-7,70%	-29,60%
	kddcup,data	0,10%	0,00%	-0,20%	0,60%	-7,80%	-1,00%
spanningForest/ndST	randLocalGraph_E_10_20000000	0,80%	0,70%	0,40%	0,60%	-4,00%	0,40%
	rMatGraph_E_12_16000000	-0,30%	-0,10%	-1,40%	1,00%	-1,20%	-2,40%
	2Dgrid_E_64000000	0,00%	-0,40%	1,20%	0,20%	-8,30%	-0,40%
breadthFirstSearch/backForwa	randLocalGraph_J_10_20000000	12,20%	12,10%	9,90%	13,80%	-3,70%	5,90%
	rMatGraph_J_12_16000000	13,60%	14,40%	11,60%	12,30%	-9,80%	4,70%
	3Dgrid_J_64000000	6,20%	5,30%	1,40%	-5,20%	-25,50%	-24,50%
maximalMatching/incrementalI	randLocalGraph_E_10_20000000	17,00%	16,50%	13,90%	13,90%	0,30%	9,30%
	rMatGraph_E_10_20000000	15,60%	14,30%	12,70%	12,00%	-0,60%	6,00%
	2Dgrid_E_64000000	13,40%	12,00%	10,80%	10,50%	-2,60%	5,30%
maximalIndependentSet/increr	randLocalGraph_JR_10_20000000	6,90%	8,30%	6,90%	5,00%	1,90%	-0,60%
	rMatGraph_JR_12_16000000	7,20%	7,50%	4,00%	6,50%	1,90%	0,80%
	3Dgrid_JR_64000000	9,80%	9,50%	8,40%	8,10%	1,50%	2,40%
nearestNeighbors/octTree	2DinCube_10000000	5,00%	5,10%	4,90%	3,20%	-5,40%	1,30%
	2Dkuzmin_10000000	4,50%	5,50%	4,70%	4,20%	-5,40%	3,10%
	3DinCube_10000000	5,20%	5,80%	4,30%	5,30%	-3,60%	2,70%
	3DonSphere_10000000	5,80%	5,90%	5,40%	6,40%	-5,30%	2,50%
	3DinCube_10000000	1,40%	2,20%	1,60%	3,00%	-8,00%	1,40%
	3Dplummer_10000000	2,30%	2,70%	3,10%	3,20%	-4,90%	2,60%
convexHull/quickHull	2DinSphere_100000000	0,60%	-6,10%	0,90%	4,30%	-5,70%	2,50%
	2Dkuzmin_100000000	1,10%	1,80%	1,00%	0,00%	-6,10%	-3,10%
	2DonSphere_100000000	0,40%	1,60%	1,10%	2,40%	-5,00%	-6,10%
delaunayTriangulation/increme	2DinCube_10M	0,60%	-0,10%	-1,50%	-3,40%	-19,30%	-10,50%
	2Dkuzmin_10M	0,30%	-0,70%	-1,30%	-3,90%	-17,50%	-10,80%
delaunayRefine/incrementalRe	2DinCubeDelaunay_5000000	2,70%	1,80%	0,20%	0,40%	-1,10%	-4,60%
	2DkuzminDelaunay_5000000	2,80%	1,50%	0,30%	-0,10%	0,20%	-3,50%
rangeQuery2d/parallelPlaneSv	2DinCube_10M	6,70%	6,40%	4,00%	4,10%	-10,30%	3,90%
	2Dkuzmin_10M	7,40%	6,20%	7,70%	5,50%	-12,50%	2,60%

Table 4.3: Gains and losses of LCWS over WSteal when executed on machine *Intel 12-24*.

				Intel 16-16		
Benchmark	**Data Set**	**1**	**2**	**4**	**8**	**16**
integerSort/parallelRadixSort	randomSeq_100M_int	0,00%	4,50%	0,20%	-14,70%	-28,40%
	exptSeq_100M_int	1,90%	0,20%	0,60%	-6,70%	-18,20%
	randomSeq_100M_int_pair_int	0,10%	-3,00%	-9,00%	-25,30%	-27,40%
	randomSeq_100M_256_int_pair_int	-0,20%	-25,60%	-53,60%	-75,10%	-77,20%
comparisonSort/sampleSort	randomSeq_100M_double	0,00%	-0,10%	0,60%	1,00%	1,10%
	exptSeq_100M_double	0,30%	0,50%	0,60%	0,70%	0,10%
	almostSortedSeq_100M_double	-1,40%	-1,10%	-0,30%	0,30%	-1,80%
	randomSeq_100M_double_pair_double	0,60%	4,40%	0,40%	2,00%	0,60%
	trigramSeq_100M	6,10%	6,20%	5,70%	5,30%	4,80%
removeDuplicates/parlayhash	randomSeq_100M_int	1,10%	1,90%	-0,10%	-3,10%	-9,70%
	exptSeq_100M_int	0,60%	-0,60%	2,00%	-0,90%	-14,60%
	trigramSeq_100M	-0,10%	-3,00%	-7,50%	-24,50%	-47,10%
histogram/parallel	randomSeq_100M_256_int	0,00%	-37,90%	-48,50%	-46,20%	-17,40%
	randomSeq_100M_100K_int	-0,60%	-0,40%	-6,20%	-6,50%	-45,80%
	randomSeq_100M_int	-0,10%	3,20%	-11,20%	-13,20%	-13,00%
	exptSeq_100M_int	-0,20%	1,40%	-5,00%	-17,10%	-17,90%
	almostEqualSeq_100000000	0,20%	0,40%	-1,60%	-9,30%	-8,00%
wordCounts/histogram	trigramString_250000000	9,80%	9,30%	3,20%	-2,70%	-18,50%
	etext99	12,30%	13,40%	12,30%	10,40%	2,30%
	wikipedia250M,txt	12,40%	12,10%	8,30%	1,40%	-12,00%
invertedIndex/parallel	wikisamp,xml	-3,70%	2,00%	-0,20%	-1,60%	-3,90%
	wikipedia250M,txt	6,50%	6,20%	8,70%	5,10%	3,00%
classify/decisionTree	covtype,data	0,30%	0,70%	0,10%	0,90%	-21,10%
	kddcup,data	-0,70%	0,30%	-1,90%	0,60%	0,40%
spanningForest/ndST	randLocalGraph_E_10_20000000	0,80%	1,70%	0,60%	0,90%	-3,70%
	rMatGraph_E_12_16000000	0,50%	-32,70%	-0,70%	5,40%	6,00%
	2Dgrid_E_64000000	0,90%	0,50%	0,30%	-3,30%	-3,30%
breadthFirstSearch/backForwa	randLocalGraph_J_10_20000000	15,30%	14,60%	14,20%	-9,50%	-1,10%
	rMatGraph_J_12_16000000	17,40%	16,90%	16,20%	10,90%	4,40%
	3Dgrid_J_64000000	10,80%	7,50%	2,20%	-6,50%	-6,70%
maximalMatching/incrementall	randLocalGraph_E_10_20000000	22,40%	22,10%	19,60%	20,60%	-0,80%
	rMatGraph_E_10_20000000	19,90%	19,80%	16,80%	13,40%	8,80%
	2Dgrid_E_64000000	17,90%	17,00%	13,60%	7,20%	5,20%
maximalIndependentSet/increi	randLocalGraph_JR_10_20000000	10,10%	3,90%	8,90%	7,50%	-2,80%
	rMatGraph_JR_12_16000000	8,30%	7,00%	8,80%	2,40%	0,00%
	3Dgrid_JR_64000000	12,80%	-0,50%	9,50%	5,00%	2,50%
nearestNeighbors/octTree	2DinCube_10000000	6,20%	7,50%	8,30%	-0,50%	4,50%
	2Dkuzmin_10000000	6,30%	6,30%	3,50%	1,80%	5,70%
	3DinCube_10000000	3,80%	4,10%	4,40%	5,00%	3,00%
	3DonSphere_10000000	1,90%	3,10%	2,60%	-1,50%	-8,00%
	3DinCube_10000000	7,80%	6,20%	7,50%	3,10%	5,80%
	3Dplummer_10000000	4,90%	7,50%	7,50%	9,00%	6,90%
convexHull/quickHull	2DinSphere_100000000	2,10%	6,00%	8,50%	5,40%	22,70%
	2Dkuzmin_100000000	3,40%	3,70%	2,40%	3,40%	-4,50%
	2DonSphere_100000000	1,20%	1,00%	1,30%	2,10%	-3,00%
delaunayTriangulation/increme	2DinCube_10M	2,10%	0,20%	0,40%	-2,70%	-12,00%
	2Dkuzmin_10M	0,20%	-0,10%	-3,20%	-3,00%	-4,10%
delaunayRefine/incrementalRe	2DinCubeDelaunay_5000000	5,00%	3,50%	-0,40%	2,00%	1,90%
	2DkuzminDelaunay_5000000	4,50%	2,80%	0,70%	3,10%	-1,40%
rangeQuery2d/parallelPlaneSv	2DinCube_10M	5,30%	9,80%	1,50%	6,40%	-0,30%
	2Dkuzmin_10M	13,00%	11,20%	5,90%	7,90%	1,90%

Table 4.4: Gains and losses of LCWS over WSteal when executed on machine *AMD 16-16*.

will call **first version** to the version that requires two tasks in the private part of the deque to do a transfer, and the other shall be called **second version**.

Different machines produced different results. While machines *Intel 12-24* and *Intel 16-16* hardly have any differences when it comes to execution times, machine *AMD 32-64* has more distinct results. Similarly to the other machines, machine *AMD 32-64* has no significant improvements or losses until it starts using 64 threads, where the **second version** of the algorithm manages to get a mean 18% improvement over the **first one**. These gains also happen to manifest in almost all tests, with very few outliers.

It is interesting that the first version of the algorithm, where steals are restricted to deques that have at least two tasks, only manages to lose in one very specific environment. Upon looking at the statistics behind the operations that were run during the benchmarks, it is also observable that it features fewer successful steals and more failed steals due to the deques not being entirely empty (that have private jobs but not public ones). The reason why this solution was not overshadowed by the rather more "correct" one is that the latter one gave too much control to the thieves.

Allowing a thief to force a worker into making jobs public without restrictions may imply that the task the worker is about to execute (it may have already finished the previous one or is just about to) is made public. This resulted in an increase in unused transfers, which are directly bound to an increase of CAS and memory fences. Another thing that did not help was the increase in time that thieves spent idle. This weird discovery does not align with the structure behind this algorithm version. The thieves were given more control over the workers. This notion is further consolidated by the increase in the number of successful steals that were observed. One plausible reason for this outcome comes from the fact that thieves may often **trade roles** with workers or simply make their lives harder. A thief may force a worker to transfer work right before the worker is about to remove it. This means that instead of the worker performing a private operation, he performs a *update_bottom* and a *pop_bottom*. In case a thief happens to steal the task before the worker manages to remove it himself, it means they simply "traded roles". The thief becomes the worker and vice versa. Both scenarios are not optimal due to the extra number of steps to reach the same result, thus delaying the computation, which implies more idle times for the thieves.

4.4.2 Comparison with LCWS

We were expecting the signal-based version to always perform better than the original LCWS. While machine *AMD 32/64* struggles between gains and losses, the other two machines produced more neutral results.

Tables 4.5, 4.7 and 4.9 show the comparison between the **first version** and LCWS, while Tables 4.6, 4.8 and 4.10 do so for the **second version**.

For *AMD 32-64*, using a low number of threads, the results bounce a lot between good and bad. When we look at the results for a high number of threads it is easily

					AMD 32-64			
Benchmark	**Data Set**	**1**	**2**	**4**	**8**	**16**	**32**	**64**
integerSort/parallelRadixSort	randomSeq_100M_int	-1,50%	18,20%	-2,80%	6,90%	-4,50%	0,00%	50,70%
	exptSeq_100M_int	3,80%	0,30%	-3,00%	-8,30%	-3,50%	40,80%	53,30%
	randomSeq_100M_int_pair_int	-3,70%	-0,30%	-6,70%	-8,70%	10,70%	30,00%	30,40%
	randomSeq_100M_256_int_pair_int	1,20%	2,30%	-0,90%	-22,50%	9,40%	0,80%	22,00%
comparisonSort/sampleSort	randomSeq_100M_double	1,40%	1,10%	1,20%	-1,20%	1,00%	2,50%	1,70%
	exptSeq_100M_double	0,40%	0,90%	0,10%	-1,90%	-2,90%	4,30%	0,80%
	almostSortedSeq_100M_double	-1,90%	0,50%	0,80%	-0,50%	-2,00%	2,80%	-11,20%
	randomSeq_100M_double_pair_double	-0,40%	0,60%	1,80%	-3,20%	-3,20%	-4,90%	58,10%
	trigramSeq_100M	-2,30%	-1,50%	-3,90%	-5,10%	-1,90%	1,40%	50,10%
removeDuplicates/parlayhash	randomSeq_100M_int	-1,20%	-3,30%	-4,90%	5,40%	14,90%	22,30%	26,30%
	exptSeq_100M_int	-2,10%	-3,90%	-6,00%	0,80%	2,60%	4,40%	29,90%
	trigramSeq_100M	0,20%	1,70%	1,00%	-0,40%	3,10%	9,00%	61,90%
histogram/parallel	randomSeq_100M_256_int	0,80%	0,00%	9,10%	27,70%	23,30%	28,00%	12,50%
	randomSeq_100M_100K_int	1,00%	-3,40%	0,40%	9,40%	12,10%	46,00%	22,20%
	randomSeq_100M_int	-0,50%	1,00%	-8,60%	1,80%	3,90%	29,30%	28,40%
	exptSeq_100M_int	0,50%	3,80%	-4,30%	9,50%	13,50%	22,10%	47,80%
	almostEqualSeq_100000000	0,80%	3,90%	-0,90%	-1,60%	8,60%	18,80%	30,90%
wordCounts/histogram	trigramString_250000000	1,40%	0,70%	0,80%	-0,20%	3,80%	7,90%	45,70%
	etext99	1,80%	2,80%	-0,30%	5,70%	-3,00%	21,60%	49,80%
	wikipedia250M,txt	1,00%	4,10%	3,80%	2,40%	5,30%	10,60%	32,00%
invertedIndex/parallel	wikisamp,xml	0,20%	-0,90%	-2,20%	-1,10%	-4,60%	-0,80%	29,30%
	wikipedia250M,txt	0,10%	-1,30%	2,40%	-0,20%	-1,50%	0,40%	21,30%
classify/decisionTree	covtype,data	0,00%	-1,30%	-0,40%	3,00%	-14,00%	-27,20%	6,40%
	kddcup,data	-0,50%	1,40%	-1,80%	1,10%	-0,40%	-9,40%	7,80%
spanningForest/ndST	randLocalGraph_E_10_20000000	1,10%	0,70%	-10,30%	-1,30%	-0,60%	-2,10%	-3,80%
	rMatGraph_E_12_16000000	-0,40%	-1,70%	12,00%	-6,20%	0,10%	-1,80%	1,20%
	2Dgrid_E_64000000	0,70%	1,90%	-2,70%	2,70%	0,40%	-1,00%	-1,90%
breadthFirstSearch/backForwa	randLocalGraph_J_10_20000000	-1,30%	-3,60%	-2,00%	3,20%	-1,80%	-3,60%	-3,10%
	rMatGraph_J_12_16000000	0,60%	3,50%	-12,00%	2,40%	-11,80%	-1,40%	-1,00%
	3Dgrid_J_64000000	-4,10%	-5,30%	-1,90%	0,30%	-5,30%	-10,00%	0,90%
maximalMatching/incrementalI	randLocalGraph_E_10_20000000	3,10%	-1,60%	-12,10%	0,40%	0,20%	2,20%	-9,10%
	rMatGraph_E_10_20000000	2,10%	-1,60%	9,00%	-0,50%	1,90%	-0,10%	-7,50%
	2Dgrid_E_64000000	1,70%	-2,90%	-1,10%	3,40%	1,50%	1,10%	-6,90%
maximalIndependentSet/incre	randLocalGraph_JR_10_20000000	1,20%	-2,40%	-18,40%	8,70%	-2,90%	-7,80%	20,70%
	rMatGraph_JR_12_16000000	0,20%	-1,20%	7,50%	-13,30%	-9,70%	-2,60%	43,40%
	3Dgrid_JR_64000000	1,20%	-2,80%	-7,30%	5,90%	-0,20%	-7,20%	23,40%
nearestNeighbors/octTree	2DinCube_10000000	1,60%	1,40%	-7,90%	-0,90%	2,20%	-5,40%	-10,90%
	2Dkuzmin_10000000	1,20%	-0,30%	-0,10%	-3,00%	-3,30%	-3,90%	-3,50%
	3DinCube_10000000	0,40%	0,30%	0,70%	-2,00%	-0,60%	-0,40%	-5,00%
	3DonSphere_10000000	0,20%	0,40%	-5,20%	4,20%	-3,30%	-0,70%	-4,10%
	3DinCube_10000000	-0,50%	0,90%	-2,70%	1,20%	0,60%	0,90%	8,70%
	3Dplummer_10000000	-0,40%	-1,50%	-3,60%	0,50%	0,40%	0,10%	-7,30%
convexHull/quickHull	2DinSphere_100000000	-3,10%	-1,50%	13,90%	13,40%	-4,40%	1,80%	-3,40%
	2Dkuzmin_100000000	-4,90%	-5,50%	10,40%	-13,40%	-1,10%	10,40%	-6,10%
	2DonSphere_100000000	-1,20%	2,90%	-7,90%	9,40%	3,20%	3,40%	-3,30%
delaunayTriangulation/increme	2DinCube_10M	-0,20%	-3,40%	2,30%	-7,60%	-6,50%	-8,10%	-2,60%
	2Dkuzmin_10M	-1,70%	-3,60%	11,60%	2,90%	3,70%	-7,40%	-3,90%
delaunayRefine/incrementalRe	2DinCubeDelaunay_5000000	0,80%	1,30%	-0,50%	0,30%	30,40%	-14,80%	-15,30%
	2DkuzminDelaunay_5000000	1,10%	1,00%	-4,90%	4,40%	28,80%	-14,50%	0,60%
rangeQuery2d/parallelPlaneSv	2DinCube_10M	1,50%	-7,30%	-2,80%	4,10%	-0,30%	-0,90%	19,90%
	2Dkuzmin_10M	-1,40%	1,10%	-4,30%	-1,00%	0,70%	1,40%	25,20%

Table 4.5: Gains and losses of the **first version** of SBLCWS over LCWS when executed on machine *AMD 32-64*.

					AMD 32-64			
Benchmark	**Data Set**	**1**	**2**	**4**	**8**	**16**	**32**	**64**
integerSort/parallelRadixSort	randomSeq_100M_int	0,30%	15,90%	-2,70%	-3,40%	-0,40%	1,20%	50,10%
	exptSeq_100M_int	-2,80%	0,50%	-2,30%	-3,10%	0,40%	38,70%	52,20%
	randomSeq_100M_int_pair_int	-0,30%	0,20%	4,40%	-16,20%	11,20%	28,20%	32,80%
	randomSeq_100M_256_int_pair_int	1,70%	-0,20%	-3,90%	-13,90%	8,00%	3,40%	24,70%
comparisonSort/sampleSort	randomSeq_100M_double	1,50%	-0,40%	2,10%	-2,40%	0,90%	0,90%	5,30%
	exptSeq_100M_double	-0,40%	0,10%	0,00%	-2,20%	-6,00%	3,90%	6,70%
	almostSortedSeq_100M_double	2,00%	-0,90%	1,90%	-0,90%	-5,40%	5,50%	13,10%
	randomSeq_100M_double_pair_double	3,80%	1,50%	1,40%	0,30%	0,80%	-1,20%	75,20%
	trigramSeq_100M	-2,30%	-1,70%	-1,20%	-2,20%	-0,50%	0,60%	70,00%
removeDuplicates/parlayhash	randomSeq_100M_int	-0,90%	2,70%	-3,20%	-5,90%	16,20%	26,30%	66,30%
	exptSeq_100M_int	-1,90%	-0,10%	-4,80%	1,00%	3,30%	3,30%	70,70%
	trigramSeq_100M	1,50%	8,00%	-0,90%	-0,30%	-0,20%	14,90%	79,50%
histogram/parallel	randomSeq_100M_256_int	0,00%	0,00%	12,70%	29,80%	26,70%	28,00%	33,30%
	randomSeq_100M_100K_int	0,80%	-0,30%	-0,90%	11,50%	19,10%	51,20%	32,60%
	randomSeq_100M_int	0,00%	4,00%	-17,20%	2,50%	6,90%	29,30%	45,90%
	exptSeq_100M_int	0,40%	3,50%	-1,40%	12,80%	13,50%	26,00%	60,90%
	almostEqualSeq_100000000	-0,10%	0,00%	0,30%	-4,20%	6,80%	19,50%	42,80%
wordCounts/histogram	trigramString_250000000	0,20%	0,00%	-2,90%	-0,90%	5,60%	7,30%	56,90%
	etext99	0,60%	0,30%	0,20%	4,20%	3,30%	19,10%	66,70%
	wikipedia250M,txt	0,10%	2,10%	-2,10%	-1,00%	4,90%	9,00%	48,40%
invertedIndex/parallel	wikisamp,xml	-0,90%	-1,90%	-4,50%	-3,40%	-4,60%	0,40%	44,70%
	wikipedia250M,txt	-2,00%	-1,40%	-0,20%	-0,70%	-1,80%	-2,50%	31,40%
classify/decisionTree	covtype,data	-1,10%	-0,30%	-2,50%	0,40%	-39,50%	-63,50%	-8,30%
	kddcup,data	-0,20%	1,20%	-1,50%	0,50%	-0,90%	-9,40%	12,80%
spanningForest/ndST	randLocalGraph_E_10_20000000	1,30%	-0,10%	3,20%	-6,60%	0,30%	-3,40%	0,50%
	rMatGraph_E_12_16000000	-0,60%	-0,20%	6,50%	-6,10%	0,10%	1,90%	-6,20%
	2Dgrid_E_64000000	1,00%	1,80%	-0,80%	3,00%	1,40%	-1,60%	1,10%
breadthFirstSearch/backForwa	randLocalGraph_J_10_20000000	-0,90%	-2,90%	-5,40%	3,80%	-1,20%	-1,20%	9,50%
	rMatGraph_J_12_16000000	-1,30%	0,10%	-17,70%	0,60%	-3,10%	1,40%	16,40%
	3Dgrid_J_64000000	1,30%	-1,30%	-2,90%	-4,40%	-2,50%	-3,10%	49,60%
maximalMatching/incrementalI	randLocalGraph_E_10_20000000	1,90%	1,10%	-8,60%	3,30%	-1,80%	1,30%	-4,70%
	rMatGraph_E_10_20000000	2,20%	-27,90%	1,40%	-0,70%	1,70%	0,60%	1,90%
	2Dgrid_E_64000000	1,20%	-4,70%	-12,00%	2,30%	2,20%	2,10%	-0,40%
maximalIndependentSet/incre	randLocalGraph_JR_10_20000000	-1,20%	-0,80%	-14,10%	0,00%	-6,50%	-13,70%	54,00%
	rMatGraph_JR_12_16000000	0,40%	-5,10%	5,50%	-14,00%	-10,10%	0,90%	66,40%
	3Dgrid_JR_64000000	0,90%	-1,00%	-3,50%	-6,00%	-0,60%	-2,30%	44,40%
nearestNeighbors/octTree	2DinCube_10000000	0,20%	0,30%	1,20%	3,70%	-3,10%	-3,60%	7,10%
	2Dkuzmin_10000000	0,60%	0,10%	5,30%	0,40%	-3,20%	-2,50%	7,40%
	3DinCube_10000000	-0,20%	-0,30%	-2,30%	-1,10%	-0,60%	0,60%	8,30%
	3DonSphere_10000000	-0,70%	-0,40%	0,70%	-2,50%	-7,40%	0,30%	9,30%
	3DinCube_10000000	0,00%	0,50%	-3,80%	0,50%	0,50%	1,50%	14,40%
	3Dplummer_10000000	-0,70%	-0,90%	2,60%	0,90%	2,10%	1,30%	1,90%
convexHull/quickHull	2DinSphere_100000000	-3,20%	-11,10%	9,50%	20,90%	0,70%	-0,70%	11,60%
	2Dkuzmin_100000000	-4,70%	-2,90%	-7,80%	-3,80%	-2,80%	8,90%	19,40%
	2DonSphere_100000000	-1,20%	2,00%	-9,70%	5,90%	-5,30%	-10,20%	-10,80%
delaunayTriangulation/increme	2DinCube_10M	0,30%	0,10%	0,60%	0,80%	-7,90%	-0,40%	21,60%
	2Dkuzmin_10M	0,40%	0,70%	2,40%	6,60%	1,60%	0,00%	23,00%
delaunayRefine/incrementalRe	2DinCubeDelaunay_5000000	0,30%	1,40%	1,30%	-2,40%	-2,60%	-8,40%	20,60%
	2DkuzminDelaunay_5000000	1,20%	-2,60%	-5,60%	-4,10%	-3,40%	-10,30%	26,10%
rangeQuery2d/parallelPlaneSv	2DinCube_10M	5,00%	1,10%	-0,60%	1,90%	-1,10%	1,20%	35,90%
	2Dkuzmin_10M	-1,80%	3,80%	-6,10%	0,10%	1,00%	2,80%	39,70%

Table 4.6: Gains and losses of the **second version** of SBLCWS over LCWS when executed on machine *AMD 32-64*.

				Intel 12-24			
Benchmark	**Data Set**	**1**	**2**	**4**	**8**	**16**	**24**
integerSort/parallelRadixSort	randomSeq_100M_int	0,70%	-0,90%	1,30%	-2,30%	3,60%	4,50%
	exptSeq_100M_int	1,20%	-1,00%	-1,40%	0,30%	3,70%	1,40%
	randomSeq_100M_int_pair_int	0,50%	-0,20%	-1,50%	1,70%	4,50%	4,60%
	randomSeq_100M_256_int_pair_int	-0,10%	-0,20%	0,00%	-2,10%	-4,10%	0,90%
comparisonSort/sampleSort	randomSeq_100M_double	-0,10%	-0,60%	-0,20%	-0,20%	11,00%	0,70%
	exptSeq_100M_double	-0,20%	-0,90%	0,30%	-0,80%	9,30%	0,00%
	almostSortedSeq_100M_double	0,70%	1,90%	1,00%	1,50%	8,60%	1,40%
	randomSeq_100M_double_pair_double	-0,30%	-0,10%	0,10%	0,00%	12,10%	4,30%
	trigramSeq_100M	-9,70%	-8,50%	-7,90%	-8,00%	6,60%	1,40%
removeDuplicates/parlayhash	randomSeq_100M_int	-0,90%	-0,80%	-0,60%	-0,70%	9,80%	0,30%
	exptSeq_100M_int	0,20%	1,70%	0,60%	2,30%	11,30%	3,20%
	trigramSeq_100M	0,10%	1,40%	6,90%	16,40%	32,40%	32,40%
histogram/parallel	randomSeq_100M_256_int	0,00%	0,00%	16,70%	39,10%	31,30%	35,70%
	randomSeq_100M_100K_int	0,10%	0,80%	2,50%	4,20%	3,40%	11,50%
	randomSeq_100M_int	0,60%	1,20%	3,90%	7,50%	3,30%	16,80%
	exptSeq_100M_int	-0,70%	3,10%	3,70%	8,70%	15,90%	18,90%
	almostEqualSeq_100000000	0,10%	0,90%	2,20%	4,40%	15,60%	13,40%
wordCounts/histogram	trigramString_250000000	1,70%	1,20%	1,50%	1,30%	12,70%	5,60%
	etext99	1,60%	1,80%	0,50%	1,90%	13,00%	6,50%
	wikipedia250M,txt	1,80%	1,40%	0,40%	1,60%	11,90%	5,90%
invertedIndex/parallel	wikisamp,xml	1,80%	1,30%	1,70%	2,00%	8,50%	-0,40%
	wikipedia250M,txt	0,70%	0,60%	-0,20%	3,00%	8,90%	2,10%
classify/decisionTree	covtype,data	0,30%	-0,50%	0,00%	1,40%	-19,60%	-12,40%
	kddcup,data	-1,70%	-0,90%	-0,40%	-1,20%	6,10%	-1,80%
spanningForest/ndST	randLocalGraph_E_10_20000000	0,40%	0,30%	1,10%	0,20%	3,80%	0,70%
	rMatGraph_E_12_16000000	1,00%	0,90%	1,70%	0,20%	3,30%	2,70%
	2Dgrid_E_64000000	1,10%	0,40%	-0,50%	-0,50%	8,30%	1,10%
breadthFirstSearch/backForwa	randLocalGraph_J_10_20000000	0,40%	0,20%	2,00%	-23,30%	8,20%	-4,80%
	rMatGraph_J_12_16000000	0,00%	0,40%	-0,30%	-1,40%	15,60%	-0,90%
	3Dgrid_J_64000000	1,00%	1,40%	-0,10%	-0,80%	2,70%	-12,10%
maximalMatching/incrementalI	randLocalGraph_E_10_20000000	1,70%	1,20%	1,60%	0,70%	10,00%	-4,20%
	rMatGraph_E_10_20000000	1,10%	1,70%	1,40%	1,10%	9,10%	-1,30%
	2Dgrid_E_64000000	1,20%	1,00%	1,30%	0,60%	10,00%	-1,30%
maximalIndependentSet/increr	randLocalGraph_JR_10_20000000	1,50%	0,60%	-1,30%	-2,70%	-3,90%	-7,20%
	rMatGraph_JR_12_16000000	-0,20%	0,00%	0,70%	-3,20%	-3,60%	-7,40%
	3Dgrid_JR_64000000	0,40%	0,20%	-0,30%	-1,30%	2,90%	-4,70%
nearestNeighbors/octTree	2DinCube_10000000	2,10%	3,00%	-0,40%	3,00%	9,50%	1,50%
	2Dkuzmin_10000000	2,00%	1,60%	1,20%	1,60%	9,30%	-0,50%
	3DinCube_10000000	-0,20%	-0,20%	0,60%	-1,30%	6,90%	-0,10%
	3DonSphere_10000000	0,10%	0,30%	0,60%	-0,80%	6,90%	0,00%
	3DinCube_10000000	0,50%	0,00%	0,40%	-0,20%	9,20%	0,20%
	3Dplummer_10000000	0,50%	0,20%	-0,90%	-0,30%	7,30%	0,30%
convexHull/quickHull	2DinSphere_100000000	2,10%	9,60%	1,70%	0,60%	9,60%	1,30%
	2Dkuzmin_100000000	0,10%	2,10%	0,20%	0,00%	5,30%	1,00%
	2DonSphere_100000000	0,70%	1,30%	0,10%	-0,70%	4,40%	1,70%
delaunayTriangulation/increme	2DinCube_10M	0,90%	0,60%	0,60%	-0,20%	10,40%	0,40%
	2Dkuzmin_10M	0,00%	0,70%	1,20%	0,70%	8,50%	2,50%
delaunayRefine/incrementalRe	2DinCubeDelaunay_5000000	0,00%	0,30%	0,10%	-1,70%	-3,10%	-1,40%
	2DkuzminDelaunay_5000000	-0,30%	0,20%	0,10%	-0,30%	-4,50%	-0,40%
rangeQuery2d/parallelPlaneSv	2DinCube_10M	2,50%	2,20%	0,30%	1,50%	13,00%	-0,30%
	2Dkuzmin_10M	0,80%	1,90%	-3,50%	3,50%	13,80%	-0,80%

Table 4.7: Gains and losses of the **first version** of SBLCWS over LCWS when executed on machine *Intel 12-24*.

				Intel 12-24			
Benchmark	**Data Set**	**1**	**2**	**4**	**8**	**16**	**24**
integerSort/parallelRadixSort	randomSeq_100M_int	0,10%	-0,30%	0,40%	1,00%	3,60%	4,90%
	exptSeq_100M_int	0,10%	-3,90%	-0,20%	0,00%	3,20%	1,40%
	randomSeq_100M_int_pair_int	0,30%	-4,20%	-1,10%	-0,20%	4,90%	4,60%
	randomSeq_100M_256_int_pair_int	-11,10%	-1,00%	-0,40%	0,00%	-4,10%	0,90%
comparisonSort/sampleSort	randomSeq_100M_double	-3,40%	-2,70%	-3,90%	-3,80%	9,30%	-1,60%
	exptSeq_100M_double	-2,80%	-2,80%	-1,10%	-3,40%	8,20%	-0,10%
	almostSortedSeq_100M_double	0,40%	1,30%	1,00%	0,00%	8,50%	0,60%
	randomSeq_100M_double_pair_double	-0,10%	0,00%	0,10%	0,70%	12,00%	3,20%
	trigramSeq_100M	-9,40%	-7,90%	-7,80%	-8,80%	6,70%	1,10%
removeDuplicates/parlayhash	randomSeq_100M_int	-0,30%	0,80%	-1,30%	1,10%	10,40%	2,30%
	exptSeq_100M_int	-1,60%	-0,20%	-1,30%	0,70%	7,00%	2,00%
	trigramSeq_100M	0,20%	4,00%	6,80%	18,80%	33,10%	34,00%
histogram/parallel	randomSeq_100M_256_int	0,00%	0,00%	16,70%	39,10%	43,80%	42,90%
	randomSeq_100M_100K_int	-0,90%	0,30%	3,80%	4,80%	4,20%	13,70%
	randomSeq_100M_int	-0,10%	2,00%	4,30%	8,80%	12,20%	20,70%
	exptSeq_100M_int	-0,40%	2,80%	4,60%	10,60%	18,10%	19,30%
	almostEqualSeq_100000000	0,00%	-0,20%	3,10%	5,50%	18,30%	17,80%
wordCounts/histogram	trigramString_250000000	1,50%	2,10%	1,90%	2,40%	13,20%	6,20%
	etext99	-0,40%	0,70%	-0,20%	0,20%	11,60%	5,80%
	wikipedia250M,txt	-0,20%	0,00%	0,40%	-0,20%	10,90%	5,20%
invertedIndex/parallel	wikisamp,xml	-0,60%	-1,80%	-0,90%	-0,40%	6,60%	-0,80%
	wikipedia250M,txt	0,60%	0,10%	-0,10%	2,30%	8,90%	2,50%
classify/decisionTree	covtype,data	1,00%	0,50%	1,10%	-0,10%	-36,60%	-37,40%
	kddcup,data	-0,10%	1,80%	0,60%	0,00%	7,00%	0,30%
spanningForest/ndST	randLocalGraph_E_10_20000000	-0,60%	0,00%	-0,40%	-0,30%	3,10%	-0,30%
	rMatGraph_E_12_16000000	0,60%	0,60%	1,30%	0,80%	3,40%	2,60%
	2Dgrid_E_64000000	0,30%	0,50%	-0,30%	0,00%	8,40%	0,70%
breadthFirstSearch/backForwa	randLocalGraph_J_10_20000000	-0,50%	-0,10%	0,70%	-1,20%	6,40%	-8,60%
	rMatGraph_J_12_16000000	-0,90%	0,20%	-1,00%	-1,40%	14,30%	-5,40%
	3Dgrid_J_64000000	0,50%	0,50%	0,50%	-1,40%	2,90%	-13,30%
maximalMatching/incrementalI	randLocalGraph_E_10_20000000	1,10%	-11,60%	1,00%	0,20%	9,60%	-3,00%
	rMatGraph_E_10_20000000	0,80%	0,60%	0,60%	0,80%	8,60%	-2,50%
	2Dgrid_E_64000000	0,40%	0,80%	0,50%	0,00%	9,90%	-2,00%
maximalIndependentSet/increr	randLocalGraph_JR_10_20000000	0,90%	-0,50%	-2,80%	-1,50%	-3,60%	-3,80%
	rMatGraph_JR_12_16000000	0,10%	-0,20%	0,90%	-5,70%	-5,10%	-12,50%
	3Dgrid_JR_64000000	0,10%	0,10%	-1,20%	-1,30%	1,30%	-5,30%
nearestNeighbors/octTree	2DinCube_10000000	0,80%	1,30%	0,30%	0,40%	9,70%	0,00%
	2Dkuzmin_10000000	0,60%	0,30%	-0,50%	1,10%	9,20%	-1,70%
	3DinCube_10000000	-0,70%	-0,70%	-1,20%	-0,30%	6,80%	-0,90%
	3DonSphere_10000000	0,00%	-0,10%	-0,30%	-0,60%	8,40%	-1,40%
	3DinCube_10000000	0,80%	0,50%	0,30%	0,20%	8,70%	-0,20%
	3Dplummer_10000000	0,60%	0,30%	-0,40%	-0,60%	6,90%	0,00%
convexHull/quickHull	2DinSphere_100000000	2,10%	9,50%	0,80%	-1,30%	8,70%	3,60%
	2Dkuzmin_100000000	0,40%	0,10%	-0,80%	0,60%	4,40%	1,00%
	2DonSphere_100000000	0,60%	2,30%	-0,20%	-1,50%	1,60%	-5,80%
delaunayTriangulation/increme	2DinCube_10M	1,40%	1,40%	1,00%	-0,10%	9,00%	0,60%
	2Dkuzmin_10M	-0,30%	1,30%	0,70%	0,70%	7,90%	0,90%
delaunayRefine/incrementalRe	2DinCubeDelaunay_5000000	-0,10%	0,20%	0,00%	-1,70%	-2,10%	-2,30%
	2DkuzminDelaunay_5000000	-0,20%	0,10%	0,20%	0,20%	-2,10%	-2,90%
rangeQuery2d/parallelPlaneSv	2DinCube_10M	3,90%	1,70%	-0,30%	-1,00%	13,20%	-1,40%
	2Dkuzmin_10M	2,40%	1,20%	-4,30%	1,50%	13,90%	0,00%

Table 4.8: Gains and losses of the **second version** of SBLCWS over LCWS when executed on machine *Intel 12-24*.

				Intel 16-16		
Benchmark	**Data Set**	**1**	**2**	**4**	**8**	**16**
integerSort/parallelRadixSort	randomSeq_100M_int	0,10%	0,10%	0,00%	14,40%	21,20%
	exptSeq_100M_int	0,30%	-0,20%	-1,20%	8,50%	17,50%
	randomSeq_100M_int_pair_int	0,20%	0,70%	13,40%	19,20%	21,10%
	randomSeq_100M_256_int_pair_int	0,30%	17,20%	38,20%	43,10%	43,60%
comparisonSort/sampleSort	randomSeq_100M_double	-1,50%	-1,80%	-0,70%	-0,60%	-0,80%
	exptSeq_100M_double	-0,60%	-0,60%	-0,60%	-0,90%	0,00%
	almostSortedSeq_100M_double	0,30%	0,50%	0,50%	-0,50%	1,70%
	randomSeq_100M_double_pair_double	-1,90%	-3,60%	0,70%	-9,80%	1,10%
	trigramSeq_100M	-4,80%	-4,80%	-4,50%	-4,50%	-3,40%
removeDuplicates/parlayhash	randomSeq_100M_int	-0,80%	-0,60%	-2,70%	1,90%	8,20%
	exptSeq_100M_int	0,10%	0,80%	-1,10%	1,40%	12,50%
	trigramSeq_100M	0,20%	1,90%	8,50%	17,80%	32,60%
histogram/parallel	randomSeq_100M_256_int	0,00%	26,30%	28,60%	26,30%	11,10%
	randomSeq_100M_100K_int	0,40%	-0,10%	4,90%	5,30%	31,40%
	randomSeq_100M_int	0,00%	1,50%	9,80%	7,20%	11,50%
	exptSeq_100M_int	-1,10%	0,50%	3,40%	12,10%	14,60%
	almostEqualSeq_100000000	-1,80%	1,50%	1,70%	3,80%	6,30%
wordCounts/histogram	trigramString_250000000	0,40%	2,20%	6,20%	11,50%	22,60%
	etext99	0,50%	0,30%	0,00%	2,20%	8,60%
	wikipedia250M,txt	0,70%	2,20%	4,90%	11,10%	20,40%
invertedIndex/parallel	wikisamp,xml	12,10%	6,80%	8,00%	9,30%	9,60%
	wikipedia250M,txt	2,20%	2,30%	0,80%	1,60%	2,80%
classify/decisionTree	covtype,data	0,20%	0,60%	-0,70%	-1,90%	-4,10%
	kddcup,data	0,20%	-1,20%	0,30%	-0,80%	-2,20%
spanningForest/ndST	randLocalGraph_E_10_20000000	-0,40%	-7,70%	-1,30%	13,80%	6,60%
	rMatGraph_E_12_16000000	0,70%	24,80%	0,90%	-2,50%	6,90%
	2Dgrid_E_64000000	-0,90%	-4,50%	-12,30%	-18,60%	1,40%
breadthFirstSearch/backForwa	randLocalGraph_J_10_20000000	-1,90%	-2,10%	-2,20%	17,20%	4,80%
	rMatGraph_J_12_16000000	-0,60%	-1,50%	-0,10%	1,40%	1,80%
	3Dgrid_J_64000000	1,90%	1,40%	-0,40%	-0,70%	-10,80%
maximalMatching/incrementalI	randLocalGraph_E_10_20000000	0,50%	1,40%	-0,80%	0,90%	11,40%
	rMatGraph_E_10_20000000	0,80%	1,00%	0,00%	1,20%	-12,10%
	2Dgrid_E_64000000	0,70%	-0,80%	-1,70%	4,80%	0,40%
maximalIndependentSet/increi	randLocalGraph_JR_10_20000000	-0,20%	5,30%	-28,00%	-9,10%	1,80%
	rMatGraph_JR_12_16000000	-20,00%	0,20%	-24,50%	0,80%	-1,70%
	3Dgrid_JR_64000000	-0,10%	11,80%	-12,70%	-3,30%	-2,80%
nearestNeighbors/octTree	2DinCube_10000000	1,70%	1,90%	-4,40%	3,50%	1,10%
	2Dkuzmin_10000000	1,90%	1,90%	2,60%	-1,10%	-0,60%
	3DinCube_10000000	0,80%	0,30%	-2,00%	0,60%	-0,30%
	3DonSphere_10000000	1,30%	0,30%	-1,10%	4,00%	9,10%
	3DinCube_10000000	0,10%	-1,90%	0,00%	0,70%	0,70%
	3Dplummer_10000000	0,00%	0,20%	0,30%	-2,10%	-1,00%
convexHull/quickHull	2DinSphere_100000000	-0,30%	-1,30%	-10,00%	2,40%	-8,60%
	2Dkuzmin_100000000	-0,10%	-0,20%	-0,60%	-1,70%	9,70%
	2DonSphere_100000000	-0,30%	-0,50%	-1,00%	0,10%	2,70%
delaunayTriangulation/increme	2DinCube_10M	-0,70%	0,90%	-2,90%	-6,30%	6,30%
	2Dkuzmin_10M	0,70%	0,40%	-0,10%	-2,30%	-0,80%
delaunayRefine/incrementalRe	2DinCubeDelaunay_5000000	0,40%	0,50%	0,80%	-1,10%	-2,70%
	2DkuzminDelaunay_5000000	1,50%	0,00%	0,70%	-2,80%	-2,60%
rangeQuery2d/parallelPlaneSv	2DinCube_10M	0,50%	0,30%	6,10%	-0,30%	2,40%
	2Dkuzmin_10M	-3,50%	-1,80%	4,00%	-0,80%	0,30%

Table 4.9: Gains and losses of the **first version** of SBLCWS over LCWS when executed on machine *AMD 16-16*.

				Intel 16-16		
Benchmark	**Data Set**	**1**	**2**	**4**	**8**	**16**
integerSort/parallelRadixSort	randomSeq_100M_int	0,50%	0,30%	-1,00%	14,10%	21,20%
	exptSeq_100M_int	0,10%	-0,60%	-2,60%	-2,50%	17,50%
	randomSeq_100M_int_pair_int	-1,90%	3,50%	14,20%	15,00%	20,30%
	randomSeq_100M_256_int_pair_int	-0,30%	17,70%	36,70%	32,20%	43,60%
comparisonSort/sampleSort	randomSeq_100M_double	-0,30%	-0,70%	-0,40%	-0,40%	-1,70%
	exptSeq_100M_double	-0,50%	-1,00%	-0,20%	-1,10%	-0,20%
	almostSortedSeq_100M_double	0,10%	0,10%	-0,30%	-1,60%	0,60%
	randomSeq_100M_double_pair_double	0,50%	-3,90%	0,70%	0,10%	0,10%
	trigramSeq_100M	-4,80%	-5,20%	-4,40%	-4,70%	-3,60%
removeDuplicates/parlayhash	randomSeq_100M_int	0,30%	1,50%	0,80%	4,80%	10,30%
	exptSeq_100M_int	-0,90%	-2,00%	-3,00%	-0,20%	12,80%
	trigramSeq_100M	0,30%	3,20%	7,20%	20,30%	35,10%
histogram/parallel	randomSeq_100M_256_int	0,00%	27,50%	30,60%	28,90%	14,80%
	randomSeq_100M_100K_int	0,60%	-4,70%	4,40%	7,00%	30,90%
	randomSeq_100M_int	-0,60%	-0,20%	10,90%	13,30%	11,90%
	exptSeq_100M_int	-2,60%	-0,30%	5,30%	8,70%	14,60%
	almostEqualSeq_100000000	-1,80%	1,60%	1,70%	5,40%	6,30%
wordCounts/histogram	trigramString_250000000	0,00%	2,40%	5,40%	12,40%	22,00%
	etext99	1,80%	0,40%	0,80%	2,80%	8,90%
	wikipedia250M,txt	1,90%	3,40%	6,20%	13,20%	20,90%
invertedIndex/parallel	wikisamp,xml	0,10%	-2,60%	-0,40%	-0,50%	1,70%
	wikipedia250M,txt	0,50%	1,80%	0,70%	1,80%	2,30%
classify/decisionTree	covtype,data	0,70%	1,10%	-1,20%	-4,40%	-21,70%
	kddcup,data	0,70%	1,60%	1,30%	0,70%	1,00%
spanningForest/ndST	randLocalGraph_E_10_20000000	-0,80%	-0,70%	-0,40%	-3,50%	6,80%
	rMatGraph_E_12_16000000	0,60%	24,50%	0,50%	-3,70%	6,80%
	2Dgrid_E_64000000	-0,80%	-1,20%	-0,50%	4,30%	3,80%
breadthFirstSearch/backForwa	randLocalGraph_J_10_20000000	1,20%	0,90%	0,40%	19,80%	3,70%
	rMatGraph_J_12_16000000	-1,60%	0,50%	2,10%	3,70%	3,20%
	3Dgrid_J_64000000	1,30%	2,70%	1,20%	-4,90%	-11,20%
maximalMatching/incrementalI	randLocalGraph_E_10_20000000	0,50%	0,90%	-0,50%	-0,20%	13,70%
	rMatGraph_E_10_20000000	1,00%	0,80%	-0,20%	1,80%	-14,60%
	2Dgrid_E_64000000	0,50%	0,70%	1,40%	6,10%	-1,70%
maximalIndependentSet/increi	randLocalGraph_JR_10_20000000	-0,30%	5,70%	-1,30%	-4,20%	3,40%
	rMatGraph_JR_12_16000000	0,80%	0,60%	-3,40%	-1,50%	-3,10%
	3Dgrid_JR_64000000	-0,60%	12,30%	-2,00%	-6,10%	-3,20%
nearestNeighbors/octTree	2DinCube_10000000	0,20%	0,10%	-0,20%	2,30%	1,30%
	2Dkuzmin_10000000	-0,20%	-4,60%	1,30%	2,10%	-1,50%
	3DinCube_10000000	3,10%	2,70%	0,30%	2,30%	1,60%
	3DonSphere_10000000	4,80%	4,00%	1,50%	6,30%	11,60%
	3DinCube_10000000	-4,50%	-1,10%	-4,70%	-3,50%	-3,40%
	3Dplummer_10000000	-4,20%	-3,20%	-3,80%	-4,80%	-5,30%
convexHull/quickHull	2DinSphere_100000000	-0,60%	-0,30%	-2,60%	-6,10%	-8,70%
	2Dkuzmin_100000000	0,00%	-0,40%	-3,70%	-1,20%	8,10%
	2DonSphere_100000000	-0,10%	-1,00%	-1,30%	-1,10%	-0,60%
delaunayTriangulation/increme	2DinCube_10M	-1,20%	1,30%	-2,90%	-1,80%	7,30%
	2Dkuzmin_10M	0,00%	-0,20%	-0,40%	-2,20%	-0,60%
delaunayRefine/incrementalRe	2DinCubeDelaunay_5000000	0,50%	0,60%	0,90%	-0,90%	-2,30%
	2DkuzminDelaunay_5000000	0,80%	-2,80%	-1,70%	-0,70%	-0,80%
rangeQuery2d/parallelPlaneSv	2DinCube_10M	0,00%	-1,20%	4,90%	-0,20%	0,50%
	2Dkuzmin_10M	-4,40%	-1,80%	0,20%	-3,20%	-0,20%

Table 4.10: Gains and losses of the **second version** of SBLCWS over LCWS when executed on machine *AMD 16-16*.

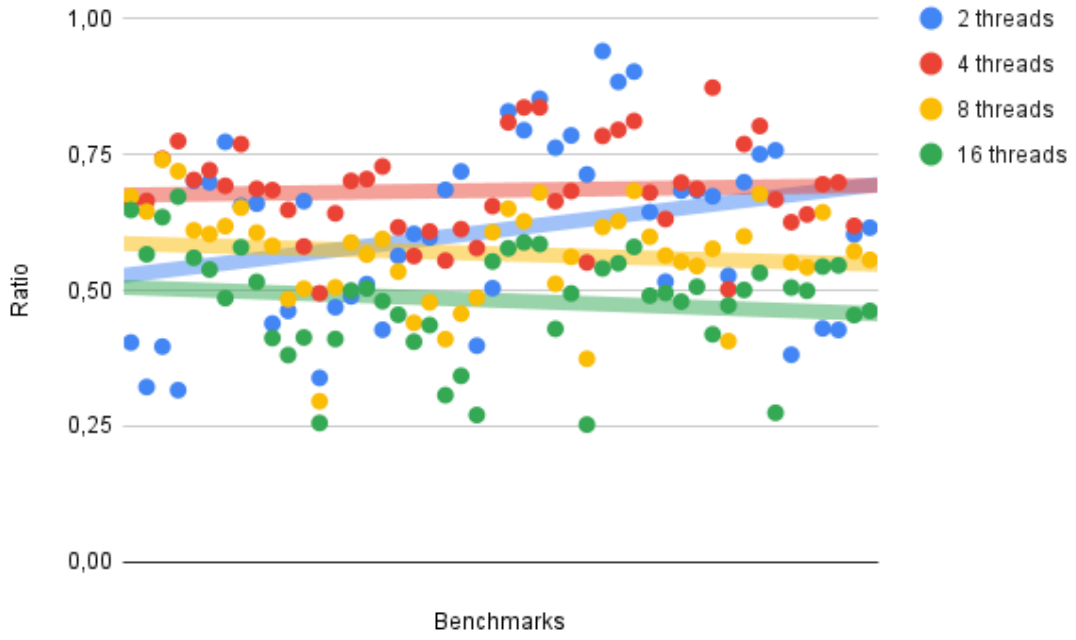


Figure 4.5: Ratio of unused updates between SBLCWS (**first version**) and LCWS on machine *Intel 16-16*.

observable that for certain benchmarks there are massive boosts in performance. Two of these benchmarks are ones that were previously mentioned in Section 4.3 for being very poorly executed using LCWS. This boost is then spread across a couple more benchmarks by using the **second version** of SBLCWS.

Intel 12-24 shows very mild results, not really going up and down like the previous machine. It features a slight boost at 16 threads across most tests.

Intel 16-16 features results more in tune with *AMD 32-64* but with lighter discrepancies.

One thing both versions of SBLCWS share is the reduced number of unused deque updates. We can see in Figure 4.5 that the algorithm starts having substantially less unused deque transfers as the number of threads grows, while keeping a relatively close number of steals. Having fewer unused *updateBottom* calls means there are less memory fences and CAS operations. The downside is that they both have more idle times than LCWS.

However, even though the threads are idle longer and the results are not as promising as we would like, this algorithm opens up more possibilities despite the fluctuations in performance. It still managed to improve on top of LCWS on a high number of threads. Another advantage is that signals make it so that the size of tasks does not matter because workers can always be interrupted to satisfy thieves. This removes the steps needed to make tasks thinner to better accommodate algorithms tailored to fine-grained jobs.

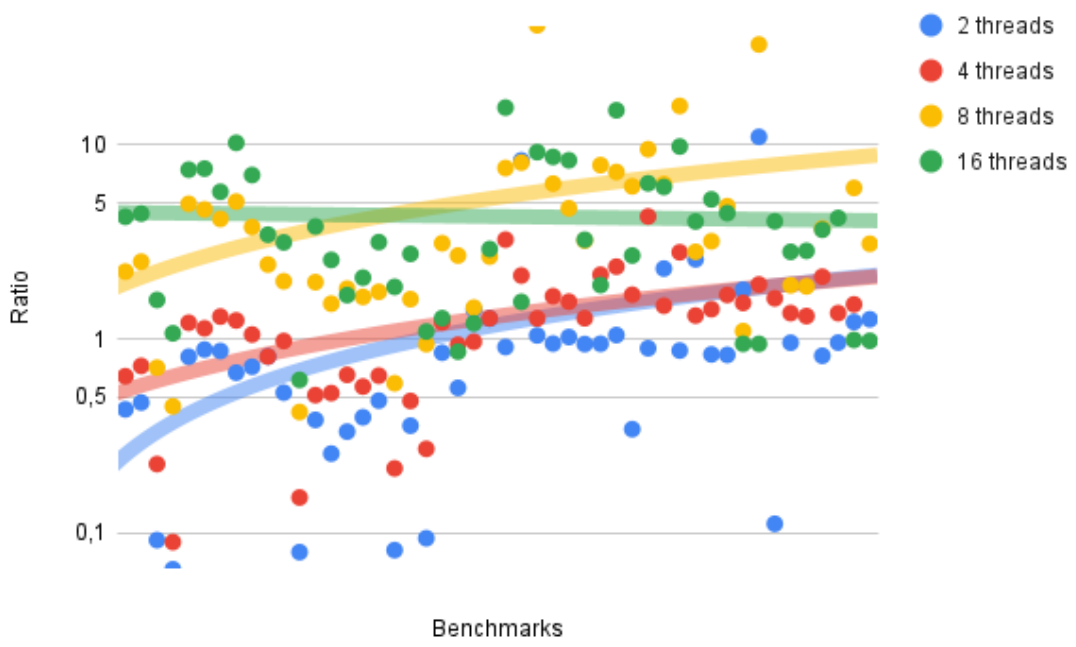


Figure 4.6: Ratio of missed steals between SBLCWS (**first version**) and LCWS on machine *Intel 16-16*.

4.5 Hybrid WShare vs LCWS

Introducing WShare in WSteal was a way to help balance the load across multiple processors. While results show that this was not the case when only using two threads, there is a decent increase in the number of successful steals when using more than this amount. This increase is directly related to the number of threads as we can see in Figure 4.7. When compared with LCWS, as we increase the number of threads, the ratio of successful steal attempts also increases.

Another positive factor shown in the results is that there is a high percentage of steals happening that were sender-initiated (Figure 4.8). This is a great result since it shows how the algorithm's basis affects its performance. Furthermore, it also helps reduce the overall number of unused updates. Figure 4.9 displays that for two threads, the percentage of unused updates revolves around 10%, while for any other number of threads it falls below 5%.

Unfortunately, accompanied by the increased number of steals, there is also an even wider difference in the ratio of missed steal attempts. Figure 4.10 shows that the ratio is incredibly high, with some outliers even reaching ten times the amount of unsuccessful steals when compared to LCWS. This alone was enough to make it so that there are mostly losses apart from using 64 threads on *AMD 32-64* and 16 on *Intel 12-24*. The results were similar to SBLCWS but less favorable. They can be seen on Tables 4.11, 4.12 and 4.13.

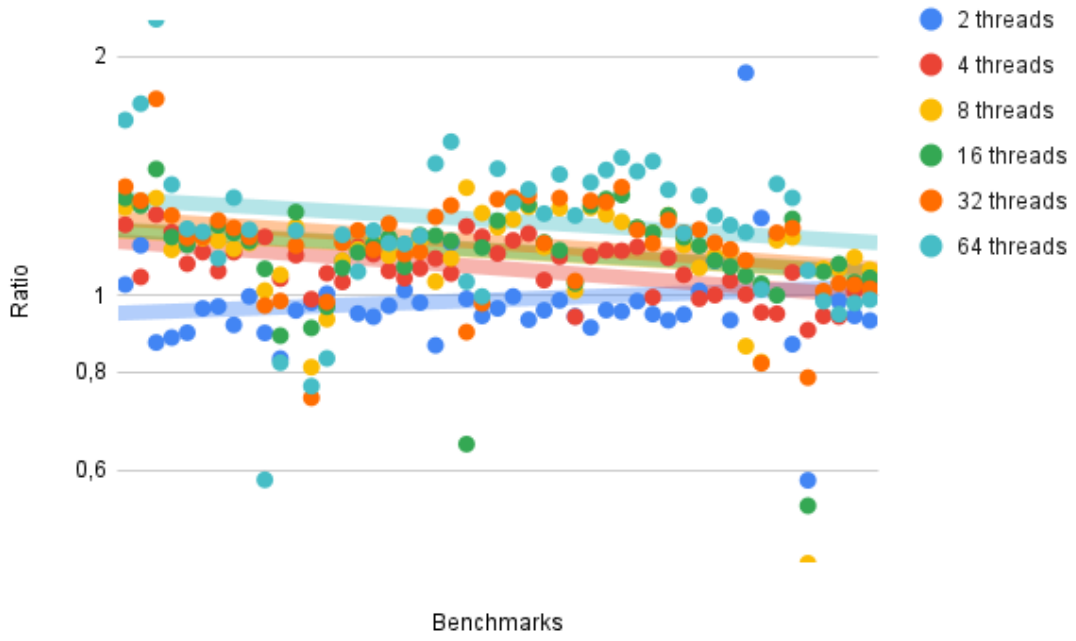


Figure 4.7: Ratio of successful steals of WShare using tit-for-tat heuristic over LCWS on machine *AMD 32-64*.

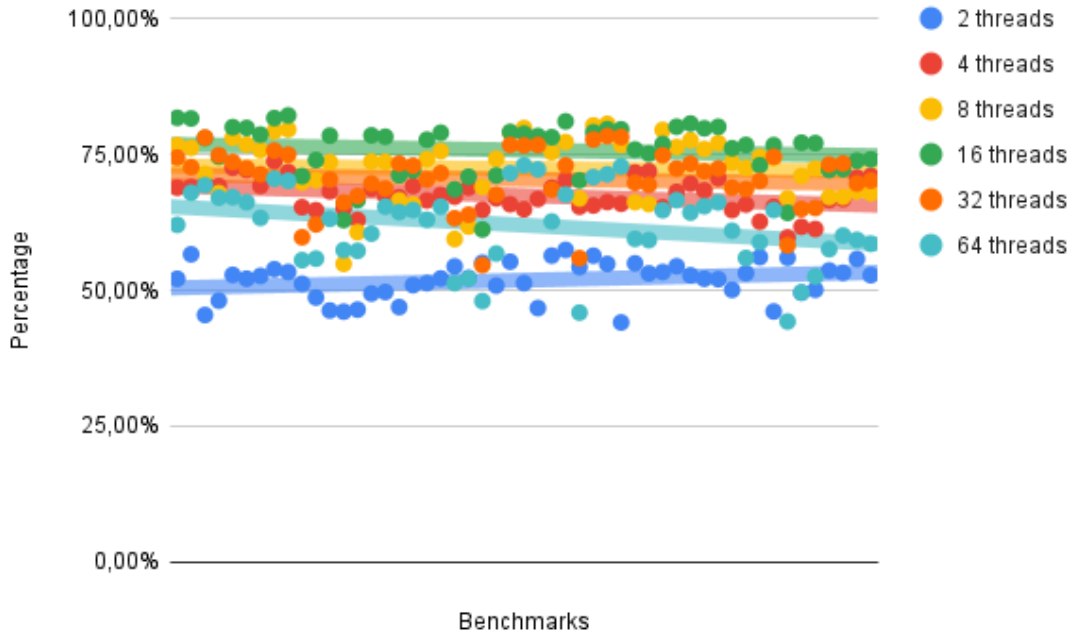


Figure 4.8: Percentage of sender-initiated steals of WShare using tit-for-tat heuristic on machine *AMD 32-64*.

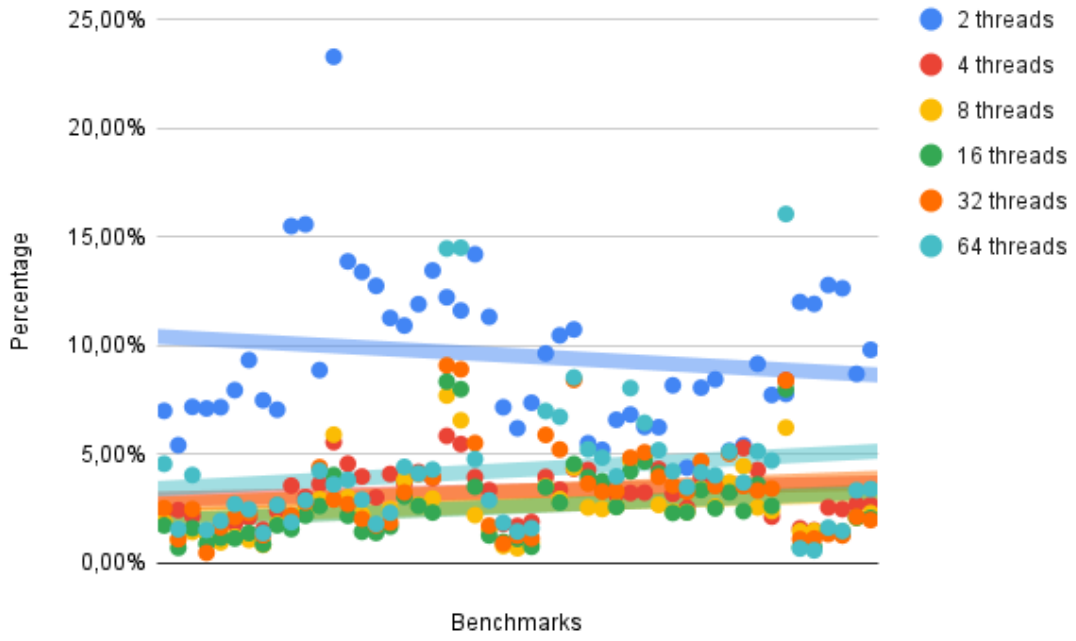


Figure 4.9: Percentage of unused updates of WShare using tit-for-tat heuristic on machine *AMD 32-64*.

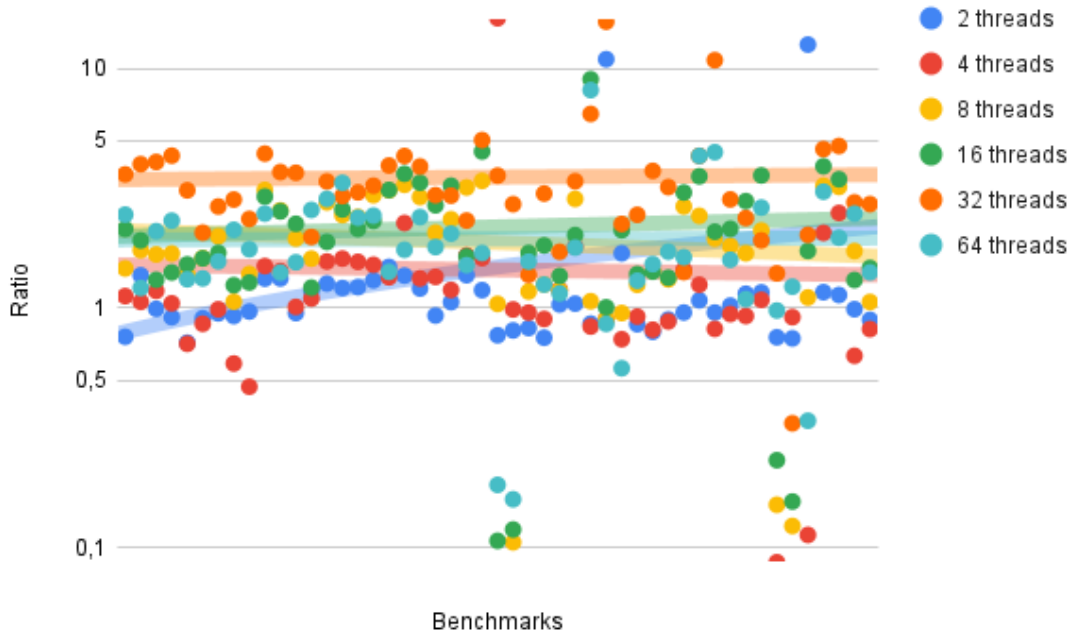


Figure 4.10: Ratio of unsuccessful steals of WShare using tit-for-tat heuristic over LCWS on machine *AMD 32-64*.

					AMD 32-64			
Benchmark	**Data Set**	**1**	**2**	**4**	**8**	**16**	**32**	**64**
integerSort/parallelRadixSort	randomSeq_100M_int	-1,10%	16,30%	-2,80%	4,90%	-2,50%	-7,20%	-2,40%
	exptSeq_100M_int	0,20%	-0,10%	-5,20%	-8,80%	-17,20%	-3,50%	46,70%
	randomSeq_100M_int_pair_int	-3,90%	-1,30%	-0,40%	-22,00%	5,90%	26,10%	27,20%
	randomSeq_100M_256_int_pair_int	1,80%	1,10%	-3,90%	-50,90%	-22,50%	-5,00%	-44,70%
comparisonSort/sampleSort	randomSeq_100M_double	-0,60%	0,10%	-0,20%	-2,10%	0,00%	-1,20%	-1,50%
	exptSeq_100M_double	-0,40%	-0,30%	-1,80%	-3,00%	-5,50%	-0,80%	4,90%
	almostSortedSeq_100M_double	-0,80%	0,10%	0,40%	-2,10%	-7,80%	3,10%	-6,80%
	randomSeq_100M_double_pair_double	-1,10%	-3,00%	-1,30%	-0,50%	-0,20%	0,30%	59,10%
	trigramSeq_100M	-2,40%	-2,60%	-2,10%	-2,60%	-0,90%	0,80%	56,70%
removeDuplicates/parlayhash	randomSeq_100M_int	1,60%	0,30%	-8,60%	3,00%	11,10%	17,00%	16,90%
	exptSeq_100M_int	-0,80%	0,60%	-1,00%	-1,90%	-5,80%	-20,80%	13,60%
	trigramSeq_100M	0,70%	3,00%	1,20%	-3,40%	-8,90%	-5,10%	50,20%
histogram/parallel	randomSeq_100M_256_int	0,80%	-10,10%	-27,30%	-19,10%	-13,30%	-12,00%	-79,20%
	randomSeq_100M_100K_int	1,10%	-1,30%	-6,10%	-4,80%	-5,00%	4,20%	-170,40%
	randomSeq_100M_int	-0,40%	-12,50%	-5,00%	-25,10%	-17,00%	-25,70%	-106,30%
	exptSeq_100M_int	0,70%	-2,00%	-4,20%	-3,20%	-16,90%	-20,50%	-69,90%
	almostEqualSeq_100000000	0,80%	2,20%	-4,10%	-10,80%	-18,30%	-16,50%	-120,40%
wordCounts/histogram	trigramString_250000000	0,30%	-2,20%	-1,40%	-3,80%	-5,10%	-27,30%	12,80%
	etext99	0,70%	0,00%	-3,60%	-2,40%	-7,40%	-2,80%	29,20%
	wikipedia250M,txt	-0,20%	2,40%	-1,60%	-7,20%	-9,70%	-12,50%	-5,20%
invertedIndex/parallel	wikisamp,xml	-2,70%	-5,30%	-5,50%	-5,40%	-10,00%	-7,80%	28,50%
	wikipedia250M,txt	-1,70%	-1,60%	-0,60%	-2,10%	-5,10%	-8,60%	20,10%
classify/decisionTree	covtype,data	-0,40%	-2,00%	0,10%	-0,30%	0,30%	7,70%	32,10%
	kddcup,data	0,00%	-1,00%	-1,20%	-1,70%	-4,90%	-9,60%	13,30%
spanningForest/ndST	randLocalGraph_E_10_20000000	0,40%	-2,70%	3,80%	-4,70%	0,90%	-2,80%	0,30%
	rMatGraph_E_12_16000000	-0,10%	0,10%	-2,00%	-4,50%	0,30%	2,80%	-6,10%
	2Dgrid_E_64000000	0,30%	2,00%	-0,30%	1,40%	0,10%	-0,20%	0,60%
breadthFirstSearch/backForwa	randLocalGraph_J_10_20000000	-2,40%	-4,60%	-4,80%	5,70%	-7,50%	-7,70%	2,70%
	rMatGraph_J_12_16000000	-2,80%	1,40%	-4,60%	-0,40%	-13,00%	-1,00%	7,80%
	3Dgrid_J_64000000	0,10%	-4,50%	-3,80%	-6,10%	-17,60%	-6,80%	41,00%
maximalMatching/incrementalI	randLocalGraph_E_10_20000000	-1,00%	0,40%	-6,10%	-2,40%	0,90%	-7,00%	4,90%
	rMatGraph_E_10_20000000	-0,40%	-4,40%	9,70%	-4,30%	0,40%	-6,10%	-0,60%
	2Dgrid_E_64000000	-0,10%	-0,60%	-1,50%	-2,70%	-0,90%	-2,70%	-1,40%
maximalIndependentSet/incre	randLocalGraph_JR_10_20000000	-2,20%	0,30%	-6,90%	-3,20%	-8,30%	-10,60%	45,30%
	rMatGraph_JR_12_16000000	0,00%	-1,20%	2,50%	-2,60%	-11,50%	2,60%	57,50%
	3Dgrid_JR_64000000	-0,40%	-3,70%	1,20%	-0,20%	-3,70%	-4,30%	42,90%
nearestNeighbors/octTree	2DinCube_10000000	-0,50%	-0,20%	-3,30%	1,90%	-4,00%	-14,20%	0,20%
	2Dkuzmin_10000000	-0,60%	-2,40%	-0,70%	1,70%	-4,90%	-21,90%	-10,70%
	3DinCube_10000000	0,30%	-0,90%	-0,70%	-1,40%	-0,40%	-3,80%	1,00%
	3DonSphere_10000000	-0,20%	-0,30%	-6,70%	-0,70%	-5,40%	-3,80%	0,00%
	3DinCube_10000000	0,40%	0,10%	-4,40%	0,30%	0,70%	-1,60%	9,60%
	3Dplummer_10000000	0,80%	2,20%	-1,40%	2,80%	2,00%	-0,90%	-2,90%
convexHull/quickHull	2DinSphere_100000000	-3,60%	-9,90%	7,50%	8,20%	1,00%	1,80%	5,10%
	2Dkuzmin_100000000	-5,10%	-6,00%	12,30%	-10,90%	-3,30%	-1,20%	11,70%
	2DonSphere_100000000	-2,70%	1,90%	-12,20%	-10,50%	0,40%	6,20%	-0,20%
delaunayTriangulation/increme	2DinCube_10M	0,80%	-4,20%	-11,10%	-4,50%	-14,10%	-16,90%	-5,60%
	2Dkuzmin_10M	0,80%	0,20%	-7,40%	-5,50%	-8,90%	-18,00%	-12,80%
delaunayRefine/incrementalRe	2DinCubeDelaunay_5000000	-0,30%	0,80%	5,90%	-2,80%	-7,30%	-5,50%	20,30%
	2DkuzminDelaunay_5000000	1,20%	1,60%	-4,20%	-3,10%	-6,80%	-4,90%	29,10%
rangeQuery2d/parallelPlaneSv	2DinCube_10M	1,50%	-4,70%	-3,70%	-0,20%	-2,40%	-3,40%	29,80%
	2Dkuzmin_10M	1,80%	1,10%	-3,10%	-2,30%	-2,30%	-2,30%	32,40%

Table 4.11: Gains and losses of LCWS with WShare using tit-for-tat over LCWS when executed on machine *AMD 32-64*.

				Intel 12-24			
Benchmark	**Data Set**	**1**	**2**	**4**	**8**	**16**	**24**
integerSort/parallelRadixSort	randomSeq_100M_int	0,70%	-0,60%	0,70%	-1,70%	-2,70%	-1,30%
	exptSeq_100M_int	-2,10%	-0,30%	-1,20%	-2,60%	-1,40%	-2,80%
	randomSeq_100M_int_pair_int	0,10%	1,20%	-0,90%	-0,70%	-0,70%	1,80%
	randomSeq_100M_256_int_pair_int	0,20%	-0,20%	-1,50%	-2,10%	-8,20%	-0,90%
comparisonSort/sampleSort	randomSeq_100M_double	0,20%	0,60%	0,50%	-0,50%	10,90%	1,00%
	exptSeq_100M_double	-0,10%	-0,60%	1,60%	-0,60%	9,10%	0,80%
	almostSortedSeq_100M_double	0,10%	-0,20%	-0,20%	-0,20%	8,60%	0,80%
	randomSeq_100M_double_pair_double	-0,20%	-0,30%	-0,50%	0,00%	10,70%	1,80%
	trigramSeq_100M	-0,80%	-0,30%	0,70%	-1,10%	9,90%	0,90%
removeDuplicates/parlayhash	randomSeq_100M_int	-1,30%	-0,40%	-3,10%	-3,90%	-0,80%	-4,20%
	exptSeq_100M_int	0,00%	0,10%	-1,40%	-2,90%	-1,30%	-6,90%
	trigramSeq_100M	0,20%	-2,10%	-4,10%	-5,10%	-1,80%	-6,80%
histogram/parallel	randomSeq_100M_256_int	0,00%	2,10%	-23,30%	-8,70%	-43,80%	-35,70%
	randomSeq_100M_100K_int	0,10%	-1,00%	-1,00%	-7,10%	-30,30%	-11,50%
	randomSeq_100M_int	0,20%	-0,40%	-3,60%	-16,30%	-32,90%	-18,20%
	exptSeq_100M_int	0,10%	-0,10%	-4,00%	-12,00%	-17,40%	-20,40%
	almostEqualSeq_100000000	1,00%	-0,90%	-4,60%	-9,90%	-14,10%	-18,10%
wordCounts/histogram	trigramString_250000000	1,40%	1,60%	-0,20%	-1,80%	4,30%	-6,80%
	etext99	1,30%	1,70%	-0,40%	-1,90%	4,00%	-7,60%
	wikipedia250M,txt	1,30%	1,30%	0,10%	-1,80%	3,50%	-4,60%
invertedIndex/parallel	wikisamp,xml	1,70%	1,00%	1,20%	0,70%	4,90%	-3,20%
	wikipedia250M,txt	0,90%	0,80%	0,20%	2,30%	6,80%	-1,70%
classify/decisionTree	covtype,data	0,20%	-0,90%	0,60%	-0,30%	-14,60%	-0,10%
	kddcup,data	0,30%	-0,10%	-0,20%	-2,10%	0,90%	-6,70%
spanningForest/ndST	randLocalGraph_E_10_20000000	-0,10%	-0,20%	0,10%	0,10%	2,20%	-5,10%
	rMatGraph_E_12_16000000	0,50%	0,80%	2,20%	0,30%	3,30%	2,40%
	2Dgrid_E_64000000	0,60%	0,30%	0,10%	0,00%	8,10%	0,80%
breadthFirstSearch/backForwa	randLocalGraph_J_10_20000000	1,40%	1,40%	3,20%	0,20%	9,30%	-4,80%
	rMatGraph_J_12_16000000	-0,10%	-1,50%	-1,10%	-1,90%	13,40%	-3,10%
	3Dgrid_J_64000000	0,10%	-0,20%	-3,40%	-9,60%	-7,60%	-19,40%
maximalMatching/incrementalI	randLocalGraph_E_10_20000000	0,40%	0,30%	1,40%	0,10%	9,50%	-3,30%
	rMatGraph_E_10_20000000	0,20%	1,00%	0,10%	0,40%	8,20%	-1,40%
	2Dgrid_E_64000000	0,70%	1,10%	0,40%	-0,20%	9,50%	-1,40%
maximalIndependentSet/increr	randLocalGraph_JR_10_20000000	1,20%	0,20%	-0,80%	-0,50%	-6,50%	-4,40%
	rMatGraph_JR_12_16000000	-0,10%	-0,30%	2,10%	-2,50%	-3,20%	-4,70%
	3Dgrid_JR_64000000	-1,00%	-1,10%	-0,40%	-1,20%	0,60%	-3,90%
nearestNeighbors/octTree	2DinCube_10000000	0,40%	0,90%	0,30%	0,50%	7,80%	-1,00%
	2Dkuzmin_10000000	0,90%	0,20%	-0,20%	0,80%	5,90%	-4,40%
	3DinCube_10000000	0,00%	0,00%	0,60%	-0,80%	6,50%	-0,80%
	3DonSphere_10000000	0,00%	-0,10%	-0,60%	-1,40%	7,90%	-1,20%
	3DinCube_10000000	0,50%	-0,20%	0,90%	-1,20%	8,10%	-1,20%
	3Dplummer_10000000	-0,10%	0,70%	-1,00%	-0,20%	8,10%	0,90%
convexHull/quickHull	2DinSphere_100000000	2,50%	9,60%	1,80%	-2,10%	3,00%	-2,80%
	2Dkuzmin_100000000	0,20%	0,30%	-0,50%	-4,40%	-2,70%	-5,00%
	2DonSphere_100000000	0,60%	1,20%	-0,40%	-1,50%	-0,40%	-1,10%
delaunayTriangulation/increme	2DinCube_10M	0,60%	0,80%	-2,30%	-5,50%	-0,60%	-17,00%
	2Dkuzmin_10M	0,40%	0,50%	-2,00%	-5,00%	-3,80%	-16,30%
delaunayRefine/incrementalRe	2DinCubeDelaunay_5000000	0,20%	0,50%	-0,40%	-2,50%	-6,80%	-8,90%
	2DkuzminDelaunay_5000000	-0,10%	-0,10%	-0,20%	-0,70%	-6,00%	-7,50%
rangeQuery2d/parallelPlaneSv	2DinCube_10M	0,50%	0,80%	-1,50%	1,10%	13,60%	1,80%
	2Dkuzmin_10M	0,40%	0,90%	-3,30%	1,70%	15,20%	0,40%

Table 4.12: Gains and losses of LCWS with WShare using tit-for-tat over LCWS when executed on machine *Intel 12-24*.

				Intel 16-16		
Benchmark	**Data Set**	**1**	**2**	**4**	**8**	**16**
integerSort/parallelRadixSort	randomSeq_100M_int	-0,40%	0,10%	1,70%	-2,80%	-0,60%
	exptSeq_100M_int	0,60%	-1,90%	-2,50%	-4,00%	0,00%
	randomSeq_100M_int_pair_int	0,30%	-0,40%	-0,80%	1,20%	-2,60%
	randomSeq_100M_256_int_pair_int	0,00%	0,70%	-1,30%	0,00%	-1,70%
comparisonSort/sampleSort	randomSeq_100M_double	0,40%	0,50%	0,00%	-2,10%	-0,20%
	exptSeq_100M_double	0,00%	0,00%	0,00%	-0,30%	-1,60%
	almostSortedSeq_100M_double	0,70%	0,50%	0,30%	0,00%	1,40%
	randomSeq_100M_double_pair_double	-1,00%	-6,50%	0,20%	-0,40%	-0,80%
	trigramSeq_100M	0,30%	0,20%	0,30%	0,40%	1,10%
removeDuplicates/parlayhash	randomSeq_100M_int	0,70%	-1,60%	-4,50%	1,40%	-2,20%
	exptSeq_100M_int	-1,40%	-0,80%	-4,40%	-5,40%	-3,80%
	trigramSeq_100M	-0,40%	-0,80%	-1,30%	-4,10%	-5,10%
histogram/parallel	randomSeq_100M_256_int	-17,30%	-2,50%	-18,40%	-13,20%	-11,10%
	randomSeq_100M_100K_int	0,40%	-0,90%	-1,00%	-5,70%	-5,20%
	randomSeq_100M_int	-0,10%	-0,40%	-0,60%	-11,40%	-16,90%
	exptSeq_100M_int	0,10%	-0,80%	-5,50%	-4,70%	-6,50%
	almostEqualSeq_100000000	-0,10%	0,00%	-4,70%	-11,40%	-12,10%
wordCounts/histogram	trigramString_250000000	-0,10%	0,30%	0,50%	-3,00%	-1,70%
	etext99	-1,00%	-1,00%	-2,30%	-4,10%	-8,10%
	wikipedia250M,txt	-1,20%	-1,20%	-2,00%	-4,40%	-5,70%
invertedIndex/parallel	wikisamp,xml	0,90%	-1,10%	-0,50%	0,20%	-0,90%
	wikipedia250M,txt	0,20%	1,00%	-0,70%	0,80%	-2,50%
classify/decisionTree	covtype,data	0,50%	0,70%	-2,30%	-3,10%	-5,30%
	kddcup,data	1,20%	-0,10%	-0,70%	-2,10%	-4,90%
spanningForest/ndST	randLocalGraph_E_10_20000000	0,20%	-0,40%	0,30%	-5,30%	-11,00%
	rMatGraph_E_12_16000000	0,60%	25,00%	0,60%	-3,00%	-2,40%
	2Dgrid_E_64000000	-0,20%	-1,10%	-18,60%	2,50%	4,10%
breadthFirstSearch/backForwa	randLocalGraph_J_10_20000000	0,70%	0,40%	-0,80%	17,60%	4,20%
	rMatGraph_J_12_16000000	-0,60%	-0,60%	-0,20%	2,70%	1,40%
	3Dgrid_J_64000000	0,10%	2,30%	-5,50%	-7,80%	-11,40%
maximalMatching/incrementalI	randLocalGraph_E_10_20000000	0,50%	-21,20%	1,20%	-1,00%	4,90%
	rMatGraph_E_10_20000000	0,50%	0,60%	-0,30%	1,00%	-17,60%
	2Dgrid_E_64000000	0,60%	0,50%	1,80%	-0,30%	1,10%
maximalIndependentSet/increi	randLocalGraph_JR_10_20000000	-0,70%	0,30%	-1,60%	-15,10%	2,90%
	rMatGraph_JR_12_16000000	1,50%	0,80%	-3,50%	-21,90%	-3,10%
	3Dgrid_JR_64000000	-1,00%	11,90%	-1,00%	-10,40%	-1,50%
nearestNeighbors/octTree	2DinCube_10000000	0,30%	0,20%	-2,20%	4,40%	0,00%
	2Dkuzmin_10000000	0,00%	-0,10%	2,60%	-0,10%	3,10%
	3DinCube_10000000	2,90%	2,50%	2,30%	-1,00%	-0,10%
	3DonSphere_10000000	5,10%	3,90%	-0,60%	6,00%	10,30%
	3DinCube_10000000	-5,50%	-8,00%	-4,30%	-4,20%	-3,70%
	3Dplummer_10000000	-4,40%	-3,60%	-3,30%	-3,50%	6,50%
convexHull/quickHull	2DinSphere_100000000	-0,30%	-0,20%	-4,00%	-1,10%	-10,90%
	2Dkuzmin_100000000	-0,20%	-0,50%	-2,10%	-8,20%	6,80%
	2DonSphere_100000000	0,30%	-0,30%	0,00%	0,90%	-2,80%
delaunayTriangulation/increme	2DinCube_10M	-1,10%	0,70%	-4,20%	-6,00%	-0,20%
	2Dkuzmin_10M	0,80%	-0,70%	-3,40%	-5,70%	-10,30%
delaunayRefine/incrementalRe	2DinCubeDelaunay_5000000	-0,20%	0,10%	1,00%	-2,90%	-3,60%
	2DkuzminDelaunay_5000000	0,90%	0,20%	-1,40%	-2,80%	-4,30%
rangeQuery2d/parallelPlaneSv	2DinCube_10M	2,00%	-0,70%	5,50%	-1,40%	1,60%
	2Dkuzmin_10M	-0,10%	-3,80%	-0,20%	-1,00%	-1,20%

Table 4.13: Gains and losses of LCWS with WShare using tit-for-tat over LCWS when executed on machine *Intel 16-16*.

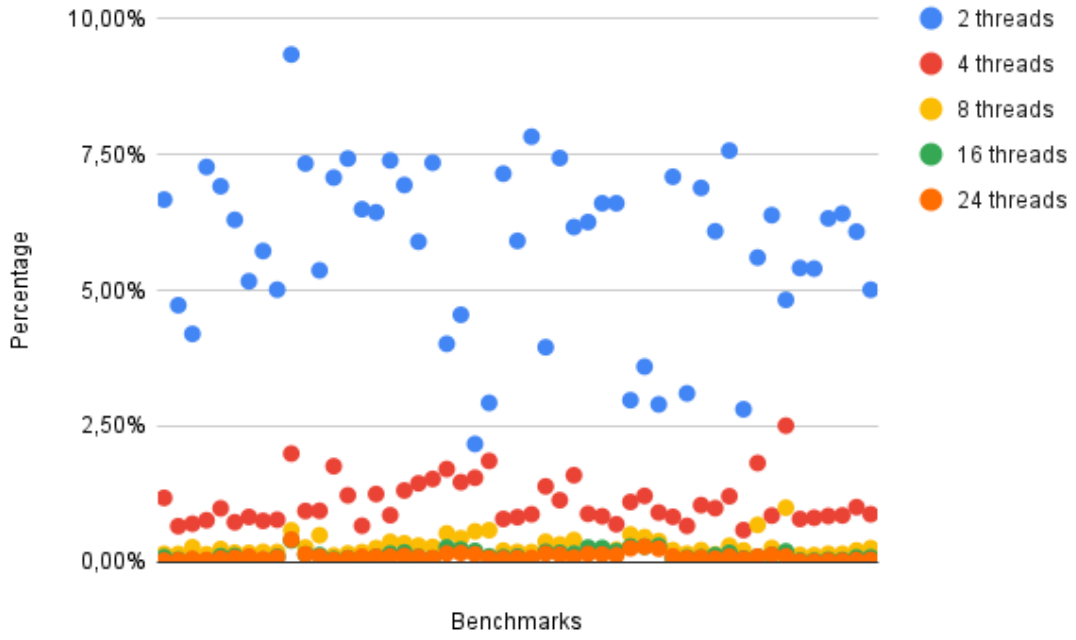


Figure 4.11: Percentage of unused updates using WShare with Signals on machine *Intel 12-24*.

4.5.1 Continuous tit-for-tat

When using the *continuous tit-for-tat* heuristic, the results did not vary much. It's noticeable that it features less successful steals, accompanied by slightly more unused updates. This is due to the continuous nature of the heuristic that keeps the worker spreading tasks until it fails to do so. The gains compared to LCWS are also almost equal to the previous heuristic, even going up and down in the same tests. Tables 4.14, 4.15 and 4.16 show the results.

4.6 SBLCWS with WShare

Just like with LCWS, we inserted signals into the WShare solution resulting in two different solutions. The results also tell the same story, where the **second** version features more steals, more unused updates and more idle time. The execution times compared to SBLCWS are also pretty similar. Using *AMD 32-64*, there are lots of gains and losses and both versions provide a significant boost at 64 threads, with the **second version** being the better one. The other machines are mostly neutral, except for a few outliers being better (again the *Integer Sort* and *Histogram* benchmarks) or worse.

The comparisons with LCWS can be seen in Tables 4.17, 4.19 and 4.21 for the **first version** and 4.18, 4.20 and 4.22 for the **second**.

This algorithm functions like SBLCWS, but with more steps involved. With that said, overall it does not perform well. The tendency is to have performances slightly below

					AMD 32-64			
Benchmark	**Data Set**	**1**	**2**	**4**	**8**	**16**	**32**	**64**
integerSort/parallelRadixSort	randomSeq_100M_int	2,50%	15,30%	7,80%	11,80%	-3,30%	-81,90%	44,80%
	exptSeq_100M_int	3,60%	-0,30%	-1,10%	-10,60%	-20,70%	37,90%	14,20%
	randomSeq_100M_int_pair_int	-0,30%	0,20%	-1,60%	-22,50%	8,50%	4,30%	-3,50%
	randomSeq_100M_256_int_pair_int	1,80%	1,70%	7,80%	-16,20%	-18,10%	-51,30%	-32,00%
comparisonSort/sampleSort	randomSeq_100M_double	2,30%	2,20%	2,70%	-1,00%	4,00%	0,20%	-0,30%
	exptSeq_100M_double	3,10%	2,00%	1,90%	-0,90%	-3,00%	5,40%	7,20%
	almostSortedSeq_100M_double	4,60%	3,90%	3,20%	-0,40%	-3,70%	-3,00%	-1,30%
	randomSeq_100M_double_pair_double	1,10%	-2,00%	2,00%	1,40%	0,00%	-1,00%	67,20%
	trigramSeq_100M	-2,70%	-1,70%	-2,20%	-1,60%	-1,90%	-1,80%	63,30%
removeDuplicates/parlayhash	randomSeq_100M_int	0,10%	2,80%	-12,30%	3,60%	12,30%	0,70%	30,50%
	exptSeq_100M_int	0,10%	-0,20%	0,60%	-7,20%	-8,40%	-35,00%	25,70%
	trigramSeq_100M	0,50%	5,20%	-1,90%	0,60%	-12,50%	-1,50%	60,40%
histogram/parallel	randomSeq_100M_256_int	0,00%	-5,80%	-32,70%	-36,20%	-30,00%	-8,00%	-25,00%
	randomSeq_100M_100K_int	1,10%	-0,70%	-6,60%	2,40%	-4,00%	2,30%	-94,10%
	randomSeq_100M_int	0,00%	2,70%	-23,30%	-18,50%	-35,90%	-25,30%	-33,40%
	exptSeq_100M_int	0,50%	3,40%	-4,40%	-7,40%	-23,40%	-23,60%	-18,80%
	almostEqualSeq_100000000	-0,10%	2,90%	-5,40%	-17,30%	-17,20%	-22,20%	-52,00%
wordCounts/histogram	trigramString_250000000	0,50%	0,00%	-3,70%	-3,90%	-3,80%	-22,00%	27,40%
	etext99	0,40%	2,20%	-3,60%	-0,40%	-11,30%	-9,60%	39,80%
	wikipedia250M,txt	0,10%	2,00%	0,10%	-6,30%	-7,00%	-19,60%	2,10%
invertedIndex/parallel	wikisamp,xml	-0,80%	-2,20%	-3,30%	-3,10%	-7,40%	-9,00%	35,30%
	wikipedia250M,txt	-0,70%	-1,40%	0,10%	-3,00%	-6,50%	-13,20%	10,50%
classify/decisionTree	covtype,data	-0,80%	-0,60%	-2,20%	4,30%	0,40%	12,30%	35,00%
	kddcup,data	-0,70%	1,40%	-0,90%	-0,20%	-6,10%	-11,10%	18,80%
spanningForest/ndST	randLocalGraph_E_10_20000000	0,70%	0,10%	-7,80%	-14,50%	3,40%	-3,40%	0,20%
	rMatGraph_E_12_16000000	-0,50%	0,00%	7,90%	-5,40%	-2,40%	-3,10%	0,10%
	2Dgrid_E_64000000	0,10%	2,20%	8,10%	-0,70%	1,60%	-0,90%	0,40%
breadthFirstSearch/backForwa	randLocalGraph_J_10_20000000	-1,60%	-0,60%	-5,00%	-1,90%	-6,90%	-8,90%	6,90%
	rMatGraph_J_12_16000000	-2,00%	2,20%	-18,40%	-0,30%	-7,00%	-12,60%	11,30%
	3Dgrid_J_64000000	-1,30%	-2,00%	-13,10%	-11,30%	-18,30%	-7,90%	42,20%
maximalMatching/incrementalI	randLocalGraph_E_10_20000000	-0,90%	-1,50%	-14,30%	-1,00%	2,20%	-2,30%	-0,40%
	rMatGraph_E_10_20000000	-0,70%	-0,20%	8,10%	1,00%	0,60%	-6,30%	4,00%
	2Dgrid_E_64000000	0,10%	-0,60%	-0,90%	-3,40%	1,30%	-4,60%	2,50%
maximalIndependentSet/incre	randLocalGraph_JR_10_20000000	-1,50%	-0,30%	-17,00%	8,00%	-11,70%	-8,50%	47,40%
	rMatGraph_JR_12_16000000	0,10%	0,60%	0,90%	-4,10%	-7,60%	-4,00%	65,60%
	3Dgrid_JR_64000000	-0,20%	0,10%	-6,50%	-1,70%	-5,10%	-4,20%	43,60%
nearestNeighbors/octTree	2DinCube_10000000	0,00%	-0,10%	1,40%	0,10%	-2,10%	-5,20%	1,40%
	2Dkuzmin_10000000	0,00%	-0,30%	6,50%	0,40%	-8,50%	-16,10%	-2,30%
	3DinCube_10000000	1,00%	0,10%	-2,30%	-0,60%	-1,20%	-1,00%	3,20%
	3DonSphere_10000000	0,60%	-0,30%	0,30%	2,70%	-3,50%	-0,70%	2,50%
	3DinCube_10000000	0,20%	0,70%	-2,90%	0,90%	0,30%	-2,40%	9,70%
	3Dplummer_10000000	0,40%	2,10%	-3,80%	1,20%	-0,30%	-1,50%	-2,70%
convexHull/quickHull	2DinSphere_100000000	-3,40%	1,40%	14,20%	4,40%	-2,00%	-7,90%	5,10%
	2Dkuzmin_100000000	-5,30%	-3,30%	3,40%	-20,30%	-1,70%	0,00%	11,40%
	2DonSphere_100000000	-2,70%	5,00%	-34,80%	-7,70%	3,90%	4,00%	-1,40%
delaunayTriangulation/increme	2DinCube_10M	0,50%	-4,10%	-11,30%	-5,30%	-13,30%	-12,60%	1,30%
	2Dkuzmin_10M	0,50%	-3,80%	9,00%	4,00%	-14,00%	-11,40%	4,00%
delaunayRefine/incrementalRe	2DinCubeDelaunay_5000000	0,70%	-2,10%	10,20%	-3,00%	-6,10%	-3,70%	24,00%
	2DkuzminDelaunay_5000000	1,30%	1,60%	-4,00%	-1,90%	-5,90%	-4,90%	31,10%
rangeQuery2d/parallelPlaneSv	2DinCube_10M	0,30%	-5,60%	3,00%	-1,00%	-0,50%	2,40%	30,60%
	2Dkuzmin_10M	4,80%	3,00%	-3,20%	0,70%	0,90%	0,50%	35,10%

Table 4.14: Gains and losses of LCWS with WShare using continuous tit-for-tat over LCWS when executed on machine *AMD 32-64*.

				Intel 12-24			
Benchmark	**Data Set**	**1**	**2**	**4**	**8**	**16**	**24**
integerSort/parallelRadixSort	randomSeq_100M_int	0,30%	-0,20%	0,40%	-1,30%	-2,30%	-0,40%
	exptSeq_100M_int	0,90%	0,00%	-1,60%	-2,30%	-0,90%	-2,40%
	randomSeq_100M_int_pair_int	0,70%	0,70%	-1,50%	0,20%	-2,10%	-0,70%
	randomSeq_100M_256_int_pair_int	-0,10%	0,40%	-1,50%	-1,40%	-8,20%	-0,90%
comparisonSort/sampleSort	randomSeq_100M_double	-0,10%	0,70%	-0,20%	-0,60%	11,70%	-0,40%
	exptSeq_100M_double	0,00%	-1,10%	0,90%	-0,60%	9,00%	-0,30%
	almostSortedSeq_100M_double	-0,10%	0,80%	-0,50%	-0,70%	7,70%	0,80%
	randomSeq_100M_double_pair_double	-0,20%	-0,10%	-0,10%	-0,20%	11,30%	0,90%
	trigramSeq_100M	-1,20%	-0,40%	0,80%	-0,20%	10,30%	0,60%
removeDuplicates/parlayhash	randomSeq_100M_int	-0,30%	0,50%	-1,00%	-4,20%	-2,80%	-3,90%
	exptSeq_100M_int	-1,10%	-1,60%	-4,60%	-6,80%	-1,00%	-7,30%
	trigramSeq_100M	-0,90%	-0,40%	-5,10%	-3,30%	-3,40%	-6,10%
histogram/parallel	randomSeq_100M_256_int	0,00%	-8,50%	-16,70%	-4,30%	-56,30%	-35,70%
	randomSeq_100M_100K_int	0,20%	0,30%	-1,30%	-7,70%	-26,90%	-12,90%
	randomSeq_100M_int	0,20%	0,10%	-6,40%	-17,30%	-35,40%	-19,60%
	exptSeq_100M_int	-0,40%	2,00%	-2,10%	-8,70%	-18,80%	-19,60%
	almostEqualSeq_100000000	-0,10%	0,30%	-2,10%	-11,00%	-12,20%	-20,70%
wordCounts/histogram	trigramString_250000000	0,60%	0,90%	-0,20%	-1,50%	5,30%	-6,70%
	etext99	-0,70%	-0,80%	-2,70%	-4,40%	1,70%	-9,50%
	wikipedia250M,txt	-0,70%	3,10%	-2,50%	-3,20%	2,70%	-6,40%
invertedIndex/parallel	wikisamp,xml	1,50%	1,90%	1,80%	0,90%	5,60%	-4,90%
	wikipedia250M,txt	0,90%	0,70%	0,00%	1,60%	6,10%	-2,70%
classify/decisionTree	covtype,data	0,40%	0,30%	0,00%	-0,10%	-13,90%	2,70%
	kddcup,data	0,10%	-0,10%	-2,00%	-1,90%	1,20%	-6,60%
spanningForest/ndST	randLocalGraph_E_10_20000000	0,50%	0,00%	-0,80%	0,90%	4,20%	0,60%
	rMatGraph_E_12_16000000	1,20%	-33,30%	0,80%	0,90%	3,10%	2,60%
	2Dgrid_E_64000000	0,70%	0,40%	-0,50%	0,00%	8,50%	0,60%
breadthFirstSearch/backForwa	randLocalGraph_J_10_20000000	1,40%	1,70%	2,00%	1,00%	9,30%	-5,30%
	rMatGraph_J_12_16000000	0,10%	-0,70%	0,40%	-0,50%	15,00%	-2,70%
	3Dgrid_J_64000000	1,20%	0,50%	-3,70%	-8,80%	-6,80%	-20,90%
maximalMatching/incrementalI	randLocalGraph_E_10_20000000	1,00%	-12,30%	1,00%	-0,30%	9,10%	-3,70%
	rMatGraph_E_10_20000000	1,00%	1,40%	0,50%	0,60%	7,30%	-1,30%
	2Dgrid_E_64000000	0,80%	1,30%	0,30%	0,00%	9,80%	-1,60%
maximalIndependentSet/increr	randLocalGraph_JR_10_20000000	0,60%	0,40%	-2,80%	-2,00%	-4,20%	-5,00%
	rMatGraph_JR_12_16000000	-0,40%	-1,00%	-0,70%	-1,90%	-4,70%	-5,40%
	3Dgrid_JR_64000000	-0,10%	-0,80%	-1,30%	-1,00%	0,30%	-5,30%
nearestNeighbors/octTree	2DinCube_10000000	1,70%	1,60%	1,30%	2,00%	9,30%	-1,30%
	2Dkuzmin_10000000	1,80%	1,60%	1,10%	1,00%	5,40%	-3,20%
	3DinCube_10000000	-0,50%	-0,50%	0,40%	-1,00%	6,40%	-2,10%
	3DonSphere_10000000	0,10%	0,20%	-0,40%	-1,00%	6,20%	-1,20%
	3DinCube_10000000	0,40%	-0,10%	0,90%	-0,50%	8,50%	-0,80%
	3Dplummer_10000000	0,50%	-3,30%	-0,50%	-0,30%	7,80%	1,20%
convexHull/quickHull	2DinSphere_100000000	0,20%	-0,70%	-1,30%	-2,30%	-0,30%	-4,70%
	2Dkuzmin_100000000	0,00%	-0,90%	-1,50%	-3,10%	-3,60%	-5,00%
	2DonSphere_100000000	0,20%	1,00%	-0,10%	-2,10%	-0,20%	-1,40%
delaunayTriangulation/increme	2DinCube_10M	1,00%	-0,20%	-2,90%	-5,90%	-0,10%	-15,40%
	2Dkuzmin_10M	-0,10%	-0,40%	-2,10%	-5,10%	-3,90%	-15,00%
delaunayRefine/incrementalRe	2DinCubeDelaunay_5000000	0,00%	1,10%	-0,10%	-2,40%	-7,40%	-7,50%
	2DkuzminDelaunay_5000000	0,30%	1,00%	0,20%	-0,50%	-5,20%	-6,00%
rangeQuery2d/parallelPlaneSv	2DinCube_10M	1,60%	3,40%	-0,20%	1,30%	13,00%	-0,40%
	2Dkuzmin_10M	4,10%	1,60%	0,80%	1,40%	15,40%	-0,20%

Table 4.15: Gains and losses of LCWS with WShare using continuous tit-for-tat over LCWS when executed on machine *Intel 12-24*.

				Intel 16-16		
Benchmark	**Data Set**	**1**	**2**	**4**	**8**	**16**
integerSort/parallelRadixSort	randomSeq_100M_int	0,20%	-1,80%	0,20%	3,30%	-1,80%
	exptSeq_100M_int	0,20%	-1,80%	-2,60%	-0,30%	-2,40%
	randomSeq_100M_int_pair_int	-0,40%	-1,30%	1,20%	1,60%	-2,30%
	randomSeq_100M_256_int_pair_int	0,20%	-2,20%	-0,90%	-3,10%	3,10%
comparisonSort/sampleSort	randomSeq_100M_double	-0,20%	-0,50%	-0,80%	-1,40%	-2,60%
	exptSeq_100M_double	-0,40%	-0,30%	-1,20%	-1,40%	-0,60%
	almostSortedSeq_100M_double	0,40%	0,00%	-0,10%	-0,80%	-0,10%
	randomSeq_100M_double_pair_double	-1,30%	-6,20%	0,30%	-0,60%	-1,60%
	trigramSeq_100M	0,20%	0,20%	0,40%	-0,20%	-0,60%
removeDuplicates/parlayhash	randomSeq_100M_int	0,80%	1,10%	-3,10%	0,50%	-2,40%
	exptSeq_100M_int	-0,70%	-0,10%	-3,80%	-4,30%	-3,50%
	trigramSeq_100M	-0,90%	-2,90%	-5,70%	-4,30%	-3,00%
histogram/parallel	randomSeq_100M_256_int	0,00%	-2,50%	-14,30%	2,60%	-11,10%
	randomSeq_100M_100K_int	0,10%	-0,30%	-0,20%	-6,60%	-5,80%
	randomSeq_100M_int	-0,60%	-0,60%	2,50%	-5,60%	-13,10%
	exptSeq_100M_int	-2,60%	-1,90%	-5,20%	-5,90%	-5,50%
	almostEqualSeq_100000000	-1,90%	-0,70%	-4,70%	-5,20%	-14,80%
wordCounts/histogram	trigramString_250000000	-0,10%	0,00%	0,40%	0,30%	-2,20%
	etext99	1,50%	1,10%	-0,30%	-1,60%	-4,20%
	wikipedia250M,txt	2,70%	2,50%	0,50%	1,70%	-4,00%
invertedIndex/parallel	wikisamp,xml	0,80%	-1,00%	0,60%	-1,10%	-2,00%
	wikipedia250M,txt	-0,20%	1,10%	-0,60%	0,50%	-2,50%
classify/decisionTree	covtype,data	0,90%	0,70%	-1,40%	-5,90%	-4,80%
	kddcup,data	0,20%	0,20%	-1,20%	-1,60%	-5,80%
spanningForest/ndST	randLocalGraph_E_10_20000000	-0,70%	-0,70%	-1,50%	68,00%	4,70%
	rMatGraph_E_12_16000000	0,00%	24,50%	1,10%	-2,80%	2,90%
	2Dgrid_E_64000000	-0,30%	-0,80%	0,30%	4,40%	3,70%
breadthFirstSearch/backForwa	randLocalGraph_J_10_20000000	-0,70%	-4,20%	-0,70%	18,80%	2,00%
	rMatGraph_J_12_16000000	-1,30%	-1,20%	-2,40%	1,90%	-1,40%
	3Dgrid_J_64000000	-0,70%	1,90%	-1,20%	-8,30%	-18,20%
maximalMatching/incrementalI	randLocalGraph_E_10_20000000	-0,10%	0,20%	-0,50%	-3,10%	0,10%
	rMatGraph_E_10_20000000	0,00%	-24,50%	-0,90%	-1,70%	-16,50%
	2Dgrid_E_64000000	-0,20%	0,10%	1,30%	4,80%	-3,80%
maximalIndependentSet/increi	randLocalGraph_JR_10_20000000	-1,10%	5,70%	-2,60%	-8,60%	-1,80%
	rMatGraph_JR_12_16000000	1,20%	0,60%	-2,00%	0,60%	-4,20%
	3Dgrid_JR_64000000	-0,60%	11,40%	-1,40%	-25,10%	-4,00%
nearestNeighbors/octTree	2DinCube_10000000	-0,50%	-5,00%	-4,20%	0,40%	-7,20%
	2Dkuzmin_10000000	-0,70%	-0,70%	-2,60%	1,40%	-4,40%
	3DinCube_10000000	-3,50%	-1,20%	-3,20%	-1,10%	-1,10%
	3DonSphere_10000000	-0,10%	-0,70%	-4,60%	-2,30%	-1,80%
	3DinCube_10000000	-0,10%	0,30%	-2,70%	-1,60%	-0,50%
	3Dplummer_10000000	0,00%	0,20%	0,70%	-0,60%	8,40%
convexHull/quickHull	2DinSphere_100000000	-0,20%	-0,80%	-3,70%	-1,80%	-9,30%
	2Dkuzmin_100000000	-0,20%	-0,20%	-0,30%	1,40%	4,10%
	2DonSphere_100000000	0,30%	-0,50%	0,10%	0,70%	-0,20%
delaunayTriangulation/increme	2DinCube_10M	-0,90%	0,80%	-3,70%	-8,50%	-0,50%
	2Dkuzmin_10M	-0,10%	-0,10%	-1,00%	-4,70%	-9,40%
delaunayRefine/incrementalRe	2DinCubeDelaunay_5000000	-0,50%	0,00%	1,80%	-3,40%	-5,50%
	2DkuzminDelaunay_5000000	0,60%	0,20%	0,90%	-1,00%	-6,10%
rangeQuery2d/parallelPlaneSv	2DinCube_10M	1,40%	-1,00%	5,70%	-3,70%	-2,70%
	2Dkuzmin_10M	-3,40%	-2,10%	1,40%	-1,70%	-0,20%

Table 4.16: Gains and losses of LCWS with WShare using continuous tit-for-tat over LCWS when executed on machine *Intel 16-16*.

					AMD 32-64			
Benchmark	**Data Set**	**1**	**2**	**4**	**8**	**16**	**32**	**64**
integerSort/parallelRadixSort	randomSeq_100M_int	2,40%	17,70%	-15,90%	1,30%	-2,00%	-6,60%	44,80%
	exptSeq_100M_int	3,70%	-1,00%	-5,20%	-11,90%	-6,60%	35,10%	44,40%
	randomSeq_100M_int_pair_int	-1,40%	-0,10%	-19,20%	-12,50%	9,00%	25,90%	27,60%
	randomSeq_100M_256_int_pair_int	1,90%	1,50%	-3,00%	-2,90%	6,50%	-16,00%	17,30%
comparisonSort/sampleSort	randomSeq_100M_double	2,30%	-3,20%	2,10%	-1,90%	1,30%	-3,50%	-5,60%
	exptSeq_100M_double	1,60%	0,40%	-1,50%	-1,50%	-3,80%	-3,90%	-7,00%
	almostSortedSeq_100M_double	3,20%	1,40%	2,60%	0,00%	-6,90%	-5,00%	-16,90%
	randomSeq_100M_double_pair_double	3,50%	-2,00%	-2,40%	-1,70%	1,00%	-2,10%	69,10%
	trigramSeq_100M	-2,40%	-2,40%	0,00%	-2,80%	-2,30%	-1,70%	63,20%
removeDuplicates/parlayhash	randomSeq_100M_int	0,10%	-2,10%	-2,30%	-6,20%	14,90%	23,30%	46,70%
	exptSeq_100M_int	-0,40%	-2,90%	-3,30%	-1,40%	-13,10%	1,60%	61,90%
	trigramSeq_100M	3,00%	4,80%	-0,20%	2,30%	2,50%	8,20%	68,80%
histogram/parallel	randomSeq_100M_256_int	0,80%	-1,40%	10,90%	31,90%	23,30%	24,00%	16,70%
	randomSeq_100M_100K_int	0,90%	-2,90%	-6,10%	5,20%	10,10%	46,50%	-20,00%
	randomSeq_100M_int	-0,90%	1,00%	-17,30%	2,30%	6,20%	20,10%	37,20%
	exptSeq_100M_int	0,80%	0,10%	-2,10%	5,40%	7,10%	16,70%	17,00%
	almostEqualSeq_100000000	1,10%	0,00%	0,70%	0,20%	2,10%	14,30%	20,10%
wordCounts/histogram	trigramString_250000000	0,10%	-1,30%	-5,00%	-2,60%	4,00%	0,80%	47,50%
	etext99	-0,10%	3,50%	-2,80%	1,90%	-0,80%	15,20%	50,60%
	wikipedia250M,txt	0,10%	1,70%	-3,30%	-0,60%	0,20%	4,90%	28,70%
invertedIndex/parallel	wikisamp,xml	-0,60%	-2,10%	-4,40%	-4,00%	-11,40%	-19,70%	19,50%
	wikipedia250M,txt	-0,80%	2,20%	-1,30%	-3,20%	-4,20%	-14,20%	14,80%
classify/decisionTree	covtype,data	0,20%	-1,40%	-0,80%	-6,30%	-39,40%	-83,50%	-37,60%
	kddcup,data	0,10%	-1,80%	-1,70%	-2,70%	-15,50%	-53,30%	-47,90%
spanningForest/ndST	randLocalGraph_E_10_20000000	1,60%	-2,10%	-1,70%	-7,60%	-1,90%	-9,90%	4,60%
	rMatGraph_E_12_16000000	1,00%	-2,40%	12,70%	-7,30%	-1,20%	7,90%	-1,10%
	2Dgrid_E_64000000	0,50%	2,50%	10,00%	-0,90%	1,10%	-1,50%	0,20%
breadthFirstSearch/backForwa	randLocalGraph_J_10_20000000	-1,20%	-1,10%	0,20%	-0,20%	-8,40%	-16,60%	-25,60%
	rMatGraph_J_12_16000000	1,60%	3,40%	-20,50%	-5,20%	-8,90%	-22,40%	-12,60%
	3Dgrid_J_64000000	0,10%	-1,30%	-3,90%	-10,50%	-39,80%	-57,70%	-12,00%
maximalMatching/incrementalI	randLocalGraph_E_10_20000000	2,70%	-3,10%	-1,90%	-3,00%	-7,40%	-15,00%	-48,00%
	rMatGraph_E_10_20000000	1,60%	1,30%	1,60%	1,80%	-4,50%	-13,80%	-32,00%
	2Dgrid_E_64000000	1,60%	-3,20%	-1,80%	-1,50%	-0,20%	-12,80%	-26,60%
maximalIndependentSet/incre	randLocalGraph_JR_10_20000000	3,00%	0,70%	-14,60%	-4,50%	-26,30%	-45,50%	23,00%
	rMatGraph_JR_12_16000000	1,40%	-2,70%	4,00%	-18,70%	-22,10%	-34,00%	43,10%
	3Dgrid_JR_64000000	2,60%	0,90%	1,40%	-3,30%	-11,90%	-26,00%	21,90%
nearestNeighbors/octTree	2DinCube_10000000	0,30%	0,90%	-2,40%	3,60%	-0,90%	-11,40%	-44,90%
	2Dkuzmin_10000000	0,60%	0,70%	5,20%	-2,00%	-3,20%	-15,20%	-18,90%
	3DinCube_10000000	0,60%	-1,40%	-3,10%	-0,80%	-0,50%	-4,20%	-10,10%
	3DonSphere_10000000	0,00%	-0,60%	3,20%	1,10%	-9,10%	-6,80%	-14,20%
	3DinCube_10000000	-0,70%	-2,50%	-2,80%	0,30%	0,60%	-0,70%	8,90%
	3Dplummer_10000000	-2,90%	-3,10%	-5,00%	0,20%	-0,80%	-2,20%	-7,10%
convexHull/quickHull	2DinSphere_100000000	-1,40%	2,90%	0,10%	18,70%	-2,30%	-5,90%	-9,20%
	2Dkuzmin_100000000	-2,60%	-0,80%	4,80%	-1,60%	-1,90%	-5,70%	-5,30%
	2DonSphere_100000000	-1,00%	-1,20%	-31,50%	-4,10%	-9,50%	-14,70%	-16,80%
delaunayTriangulation/increme	2DinCube_10M	0,60%	-3,80%	-11,50%	-4,20%	-26,20%	-58,60%	-72,30%
	2Dkuzmin_10M	0,70%	0,20%	0,70%	2,00%	-16,80%	-47,30%	-55,20%
delaunayRefine/incrementalRe	2DinCubeDelaunay_5000000	1,60%	-0,30%	1,70%	-5,80%	-28,20%	-58,40%	-39,60%
	2DkuzminDelaunay_5000000	1,70%	-0,10%	-2,10%	-7,10%	-27,60%	-62,70%	-37,10%
rangeQuery2d/parallelPlaneSv	2DinCube_10M	0,80%	-6,70%	-6,80%	2,90%	-4,00%	-1,80%	22,00%
	2Dkuzmin_10M	-1,90%	1,30%	-8,50%	3,30%	-1,30%	-2,90%	28,60%

Table 4.17: Gains and losses of the **first version** of WShare with signals over LCWS when executed on machine *AMD 32-64*.

					AMD 32-64			
Benchmark	**Data Set**	**1**	**2**	**4**	**8**	**16**	**32**	**64**
integerSort/parallelRadixSort	randomSeq_100M_int	1,10%	17,30%	12,60%	5,60%	-1,60%	-9,60%	42,10%
	exptSeq_100M_int	4,00%	0,00%	0,40%	-13,20%	-4,00%	32,60%	43,00%
	randomSeq_100M_int_pair_int	-6,90%	0,20%	-9,10%	-16,60%	6,30%	25,20%	24,30%
	randomSeq_100M_256_int_pair_int	1,70%	1,70%	-3,30%	-7,50%	6,50%	-3,40%	18,00%
comparisonSort/sampleSort	randomSeq_100M_double	-0,40%	-0,30%	0,30%	-2,80%	-1,20%	-3,60%	-5,80%
	exptSeq_100M_double	1,50%	-3,40%	-1,90%	-1,40%	-5,50%	-4,90%	-1,80%
	almostSortedSeq_100M_double	-1,50%	0,50%	0,80%	-0,70%	-5,80%	0,60%	-3,60%
	randomSeq_100M_double_pair_double	3,10%	0,70%	-1,30%	1,90%	-1,50%	-2,80%	74,00%
	trigramSeq_100M	-2,70%	-5,50%	-2,60%	-3,90%	-0,70%	0,40%	67,80%
removeDuplicates/parlayhash	randomSeq_100M_int	-0,90%	2,50%	-8,30%	3,00%	9,00%	21,30%	61,80%
	exptSeq_100M_int	-1,80%	0,40%	-3,00%	1,90%	0,70%	-1,10%	65,10%
	trigramSeq_100M	0,30%	2,30%	1,00%	4,10%	0,10%	10,60%	79,50%
histogram/parallel	randomSeq_100M_256_int	0,80%	-1,40%	7,30%	40,40%	33,30%	28,00%	25,00%
	randomSeq_100M_100K_int	1,00%	-3,10%	-3,90%	11,50%	12,10%	44,20%	22,20%
	randomSeq_100M_int	-0,10%	1,20%	-10,90%	-6,20%	2,70%	28,10%	49,70%
	exptSeq_100M_int	1,50%	1,00%	-8,00%	12,50%	11,40%	20,50%	57,60%
	almostEqualSeq_100000000	0,10%	0,20%	0,00%	-3,50%	9,20%	17,70%	32,00%
wordCounts/histogram	trigramString_250000000	1,70%	0,30%	-4,00%	0,20%	2,30%	3,10%	52,10%
	etext99	1,90%	2,40%	-2,80%	3,00%	-3,60%	11,30%	54,00%
	wikipedia250M,txt	0,90%	0,10%	1,30%	0,30%	1,30%	7,40%	38,10%
invertedIndex/parallel	wikisamp,xml	1,20%	0,60%	-2,60%	-1,80%	-9,40%	-18,00%	22,50%
	wikipedia250M,txt	0,90%	0,50%	0,90%	-1,80%	-1,90%	-11,20%	16,30%
classify/decisionTree	covtype,data	-1,70%	-1,10%	-2,40%	-11,80%	-90,40%	-172,10%	-94,50%
	kddcup,data	-0,40%	1,60%	0,30%	-4,50%	-20,20%	-69,80%	-79,80%
spanningForest/ndST	randLocalGraph_E_10_20000000	1,70%	0,30%	-3,40%	-4,80%	-0,70%	-3,50%	3,60%
	rMatGraph_E_12_16000000	-0,20%	-2,50%	9,40%	-1,90%	-0,60%	2,30%	-7,20%
	2Dgrid_E_64000000	0,70%	1,80%	2,30%	3,40%	2,20%	-1,10%	-1,90%
breadthFirstSearch/backForwa	randLocalGraph_J_10_20000000	-0,80%	-2,80%	-4,50%	4,60%	-10,80%	-25,90%	-28,60%
	rMatGraph_J_12_16000000	0,00%	-2,30%	-17,30%	0,40%	-15,10%	-16,10%	-5,10%
	3Dgrid_J_64000000	1,30%	1,90%	-9,90%	-15,30%	-35,80%	-48,70%	27,80%
maximalMatching/incrementalI	randLocalGraph_E_10_20000000	2,70%	-1,90%	-7,80%	-4,30%	-6,70%	-16,10%	-38,80%
	rMatGraph_E_10_20000000	3,20%	2,10%	5,70%	-0,60%	-2,10%	-16,40%	-32,20%
	2Dgrid_E_64000000	2,10%	1,60%	-4,80%	2,20%	-2,60%	-11,60%	-27,00%
maximalIndependentSet/incre	randLocalGraph_JR_10_20000000	1,60%	-1,80%	-17,40%	-2,00%	-22,80%	-41,40%	36,30%
	rMatGraph_JR_12_16000000	1,30%	-1,20%	1,50%	-11,80%	-22,80%	-36,90%	53,50%
	3Dgrid_JR_64000000	1,40%	-3,20%	-1,00%	-6,60%	-12,40%	-22,50%	25,70%
nearestNeighbors/octTree	2DinCube_10000000	0,30%	-2,00%	-3,30%	0,60%	-0,60%	-17,60%	-36,70%
	2Dkuzmin_10000000	-0,30%	-1,00%	4,00%	-2,70%	-7,00%	-20,20%	-10,70%
	3DinCube_10000000	0,90%	0,50%	-0,70%	0,20%	-1,10%	-4,90%	-12,20%
	3DonSphere_10000000	0,50%	-2,00%	0,80%	-3,40%	-5,50%	-8,60%	-18,70%
	3DinCube_10000000	-0,90%	-0,60%	-2,30%	1,20%	0,40%	-0,80%	11,50%
	3Dplummer_10000000	-4,50%	0,30%	1,90%	1,70%	1,80%	-2,60%	-7,70%
convexHull/quickHull	2DinSphere_100000000	-3,40%	1,80%	2,20%	19,80%	-3,00%	-7,60%	-1,60%
	2Dkuzmin_100000000	-5,20%	-3,40%	5,30%	-16,10%	-0,80%	-7,70%	-6,10%
	2DonSphere_100000000	-2,20%	7,80%	-16,90%	-14,20%	-26,30%	-49,50%	-24,10%
delaunayTriangulation/increme	2DinCube_10M	1,00%	-1,70%	-11,10%	-8,30%	-35,00%	-81,30%	-0,10%
	2Dkuzmin_10M	1,00%	0,30%	-0,70%	-0,90%	-25,60%	-67,50%	10,60%
delaunayRefine/incrementalRe	2DinCubeDelaunay_5000000	1,00%	0,70%	0,90%	-8,10%	-33,30%	-74,00%	21,60%
	2DkuzminDelaunay_5000000	0,90%	-1,70%	0,60%	-9,40%	-32,70%	-75,90%	24,50%
rangeQuery2d/parallelPlaneSv	2DinCube_10M	1,70%	-2,00%	-6,50%	5,00%	-1,20%	-4,00%	27,80%
	2Dkuzmin_10M	-1,50%	-3,80%	-0,30%	-2,70%	-0,10%	-5,50%	34,40%

Table 4.18: Gains and losses of the **second version** of WShare with signals over LCWS when executed on machine *AMD 32-64*.

				Intel 12-24			
Benchmark	**Data Set**	**1**	**2**	**4**	**8**	**16**	**24**
integerSort/parallelRadixSort	randomSeq_100M_int	0,30%	-4,70%	0,70%	-1,00%	0,50%	2,70%
	exptSeq_100M_int	0,70%	-0,90%	-0,30%	-0,60%	1,40%	-2,40%
	randomSeq_100M_int_pair_int	0,60%	-6,30%	-1,20%	-0,50%	1,70%	1,80%
	randomSeq_100M_256_int_pair_int	0,30%	-0,40%	0,00%	-0,70%	-6,10%	-0,90%
comparisonSort/sampleSort	randomSeq_100M_double	-3,30%	-3,10%	-4,00%	-4,10%	7,90%	-3,10%
	exptSeq_100M_double	-2,90%	-3,20%	-1,70%	-3,30%	7,30%	-3,40%
	almostSortedSeq_100M_double	0,40%	1,10%	-0,20%	-0,20%	8,80%	0,20%
	randomSeq_100M_double_pair_double	-0,20%	-0,30%	-1,20%	-0,70%	11,50%	1,10%
	trigramSeq_100M	-9,10%	-7,70%	-7,90%	-8,10%	5,10%	0,60%
removeDuplicates/parlayhash	randomSeq_100M_int	0,20%	1,00%	0,70%	0,40%	8,80%	0,00%
	exptSeq_100M_int	0,10%	2,50%	3,30%	1,80%	10,00%	2,40%
	trigramSeq_100M	-1,50%	3,80%	7,90%	19,00%	31,80%	32,80%
histogram/parallel	randomSeq_100M_256_int	0,00%	0,00%	16,70%	39,10%	31,30%	35,70%
	randomSeq_100M_100K_int	0,30%	0,30%	2,90%	3,60%	0,00%	9,40%
	randomSeq_100M_int	0,40%	2,90%	3,70%	6,50%	7,70%	13,60%
	exptSeq_100M_int	0,50%	3,20%	3,70%	7,00%	14,50%	15,20%
	almostEqualSeq_100000000	1,50%	1,80%	1,70%	3,70%	14,40%	12,30%
wordCounts/histogram	trigramString_250000000	1,30%	1,20%	1,80%	2,00%	11,30%	3,40%
	etext99	1,10%	1,10%	1,00%	1,00%	11,30%	2,50%
	wikipedia250M,txt	1,50%	1,80%	1,00%	1,20%	10,90%	3,60%
invertedIndex/parallel	wikisamp,xml	0,40%	-0,40%	0,80%	-0,70%	2,30%	-10,50%
	wikipedia250M,txt	0,90%	1,10%	1,00%	2,40%	7,50%	-2,20%
classify/decisionTree	covtype,data	0,20%	0,20%	0,40%	-1,40%	-48,20%	-60,40%
	kddcup,data	0,20%	0,70%	-1,30%	-2,30%	-2,20%	-16,90%
spanningForest/ndST	randLocalGraph_E_10_20000000	0,80%	0,40%	-0,30%	-0,80%	2,20%	-0,50%
	rMatGraph_E_12_16000000	1,00%	1,10%	1,70%	1,40%	-1,40%	1,50%
	2Dgrid_E_64000000	0,90%	0,40%	-0,60%	-0,50%	7,60%	0,40%
breadthFirstSearch/backForwa	randLocalGraph_J_10_20000000	-2,90%	2,10%	-3,70%	-1,00%	4,30%	-16,70%
	rMatGraph_J_12_16000000	0,90%	0,00%	-1,00%	-3,10%	11,50%	-11,20%
	3Dgrid_J_64000000	0,80%	1,40%	-0,80%	-11,60%	-24,30%	-58,90%
maximalMatching/incrementalI	randLocalGraph_E_10_20000000	1,30%	2,20%	1,30%	-1,30%	4,10%	-15,00%
	rMatGraph_E_10_20000000	1,30%	2,10%	0,40%	-0,30%	5,20%	-9,80%
	2Dgrid_E_64000000	1,40%	1,80%	1,30%	-0,70%	6,40%	-9,30%
maximalIndependentSet/increr	randLocalGraph_JR_10_20000000	1,30%	-0,80%	-3,10%	-11,20%	-21,70%	-29,20%
	rMatGraph_JR_12_16000000	1,30%	0,20%	0,30%	-10,10%	-27,70%	-36,60%
	3Dgrid_JR_64000000	1,20%	0,30%	-1,30%	-4,80%	-8,80%	-21,50%
nearestNeighbors/octTree	2DinCube_10000000	-0,30%	-0,10%	-0,70%	2,40%	6,20%	-2,30%
	2Dkuzmin_10000000	0,10%	-0,70%	0,60%	-0,20%	6,00%	-6,70%
	3DinCube_10000000	-0,40%	-0,50%	-0,90%	-1,90%	5,40%	-1,90%
	3DonSphere_10000000	-0,20%	-0,10%	0,10%	-1,60%	6,60%	-3,20%
	3DinCube_10000000	0,20%	0,60%	0,10%	-0,50%	8,40%	-0,70%
	3Dplummer_10000000	-0,80%	-0,80%	-0,10%	-0,70%	6,10%	-1,60%
convexHull/quickHull	2DinSphere_100000000	0,10%	6,90%	-0,10%	-5,40%	1,40%	-9,40%
	2Dkuzmin_100000000	0,10%	-0,60%	-1,70%	-2,20%	-0,40%	-6,40%
	2DonSphere_100000000	0,30%	1,10%	0,00%	-3,70%	-5,60%	-16,30%
delaunayTriangulation/increme	2DinCube_10M	1,60%	1,70%	0,80%	-4,20%	-4,10%	-32,80%
	2Dkuzmin_10M	0,60%	1,70%	1,20%	-3,10%	-4,60%	-27,40%
delaunayRefine/incrementalRe	2DinCubeDelaunay_5000000	0,70%	1,60%	0,40%	-6,10%	-21,20%	-38,00%
	2DkuzminDelaunay_5000000	0,20%	1,70%	0,20%	-4,20%	-22,10%	-36,50%
rangeQuery2d/parallelPlaneSv	2DinCube_10M	0,80%	1,00%	-0,60%	1,70%	12,60%	-4,20%
	2Dkuzmin_10M	1,10%	2,70%	-1,90%	1,60%	12,30%	-4,50%

Table 4.19: Gains and losses of the **first version** of WShare with signals over LCWS when executed on machine *Intel 12-24*.

				Intel 12-24			
Benchmark	**Data Set**	**1**	**2**	**4**	**8**	**16**	**24**
integerSort/parallelRadixSort	randomSeq_100M_int	-0,60%	0,40%	0,50%	0,00%	1,40%	2,20%
	exptSeq_100M_int	-0,10%	-1,00%	-0,20%	-1,00%	0,90%	-3,30%
	randomSeq_100M_int_pair_int	0,30%	-0,30%	-0,60%	-0,20%	2,40%	2,50%
	randomSeq_100M_256_int_pair_int	-0,10%	-3,30%	-1,10%	-0,70%	-5,10%	-0,90%
comparisonSort/sampleSort	randomSeq_100M_double	0,20%	0,00%	0,50%	-0,80%	10,40%	-1,60%
	exptSeq_100M_double	0,10%	-1,00%	0,60%	-0,90%	8,30%	-1,80%
	almostSortedSeq_100M_double	-0,10%	0,60%	-0,50%	-0,30%	7,30%	-0,20%
	randomSeq_100M_double_pair_double	-0,10%	0,40%	-0,50%	0,20%	11,70%	4,00%
	trigramSeq_100M	-9,80%	-8,70%	-8,10%	-9,40%	6,30%	0,60%
removeDuplicates/parlayhash	randomSeq_100M_int	-0,20%	0,40%	0,30%	0,20%	9,60%	0,30%
	exptSeq_100M_int	-1,60%	0,90%	1,40%	1,40%	10,00%	1,20%
	trigramSeq_100M	0,00%	1,20%	7,40%	18,10%	32,30%	33,40%
histogram/parallel	randomSeq_100M_256_int	0,00%	0,00%	16,70%	39,10%	43,80%	35,70%
	randomSeq_100M_100K_int	0,10%	0,80%	2,90%	4,20%	2,50%	11,50%
	randomSeq_100M_int	0,30%	2,40%	4,30%	8,10%	13,00%	19,30%
	exptSeq_100M_int	-0,60%	3,70%	5,10%	9,80%	18,50%	18,10%
	almostEqualSeq_100000000	-0,10%	2,20%	3,30%	4,80%	17,70%	16,30%
wordCounts/histogram	trigramString_250000000	1,50%	1,60%	1,70%	1,70%	11,00%	4,00%
	etext99	-0,20%	0,40%	-0,40%	0,20%	10,20%	2,90%
	wikipedia250M,txt	-0,50%	0,50%	-1,10%	-1,00%	10,20%	3,90%
invertedIndex/parallel	wikisamp,xml	1,90%	1,30%	2,30%	0,20%	3,90%	-8,50%
	wikipedia250M,txt	0,80%	1,00%	0,70%	1,40%	8,30%	-0,50%
classify/decisionTree	covtype,data	0,10%	-0,30%	0,00%	-21,00%	-100,60%	-120,40%
	kddcup,data	0,60%	-0,70%	-0,20%	-3,50%	-6,50%	-22,60%
spanningForest/ndST	randLocalGraph_E_10_20000000	-0,30%	0,60%	-0,20%	0,30%	3,80%	-0,20%
	rMatGraph_E_12_16000000	0,50%	1,50%	1,10%	1,30%	2,10%	1,50%
	2Dgrid_E_64000000	0,50%	0,80%	-0,80%	-0,70%	8,10%	0,50%
breadthFirstSearch/backForwa	randLocalGraph_J_10_20000000	-0,80%	-0,20%	0,40%	-2,00%	3,20%	-14,80%
	rMatGraph_J_12_16000000	-0,40%	-0,50%	0,10%	-2,40%	10,80%	-10,70%
	3Dgrid_J_64000000	0,30%	1,00%	0,50%	-11,80%	-20,60%	-48,10%
maximalMatching/incrementalI	randLocalGraph_E_10_20000000	1,70%	2,20%	1,70%	-1,00%	4,10%	-14,40%
	rMatGraph_E_10_20000000	1,50%	1,50%	2,00%	0,30%	4,00%	-10,10%
	2Dgrid_E_64000000	1,70%	2,00%	1,40%	-0,20%	6,70%	-10,50%
maximalIndependentSet/increr	randLocalGraph_JR_10_20000000	0,30%	0,60%	-3,50%	-7,70%	-23,00%	-29,50%
	rMatGraph_JR_12_16000000	-0,80%	-1,30%	0,20%	-13,60%	-27,30%	-35,00%
	3Dgrid_JR_64000000	-0,20%	-0,70%	-1,60%	-6,20%	-9,40%	-21,50%
nearestNeighbors/octTree	2DinCube_10000000	0,30%	0,20%	0,30%	0,30%	6,10%	-2,00%
	2Dkuzmin_10000000	0,50%	0,20%	-0,50%	-0,80%	5,00%	-4,70%
	3DinCube_10000000	-0,90%	-1,00%	0,00%	-1,30%	5,60%	-3,00%
	3DonSphere_10000000	-0,40%	-0,50%	-2,60%	-1,00%	5,40%	-4,80%
	3DinCube_10000000	0,20%	-0,40%	0,50%	-0,70%	8,20%	-1,50%
	3Dplummer_10000000	0,40%	-0,20%	-0,40%	-0,80%	5,20%	-2,80%
convexHull/quickHull	2DinSphere_100000000	0,20%	7,80%	-0,60%	-4,70%	-1,90%	-10,00%
	2Dkuzmin_100000000	0,00%	-0,20%	-0,70%	-1,60%	-0,40%	-7,40%
	2DonSphere_100000000	-0,10%	0,70%	-0,90%	-5,90%	-21,80%	-40,00%
delaunayTriangulation/increme	2DinCube_10M	0,30%	1,00%	-0,90%	-7,40%	-11,70%	-43,60%
	2Dkuzmin_10M	0,40%	0,00%	0,00%	-5,90%	-12,20%	-35,10%
delaunayRefine/incrementalRe	2DinCubeDelaunay_5000000	0,00%	1,30%	0,10%	-7,50%	-26,90%	-41,10%
	2DkuzminDelaunay_5000000	-0,10%	0,30%	0,20%	-6,50%	-27,70%	-41,60%
rangeQuery2d/parallelPlaneSv	2DinCube_10M	2,00%	0,00%	-0,50%	1,20%	13,50%	-2,60%
	2Dkuzmin_10M	1,10%	0,30%	-2,30%	0,90%	12,60%	-4,10%

Table 4.20: Gains and losses of the **second version** of WShare with signals over LCWS when executed on machine *Intel 12-24*.

				Intel 16-16		
Benchmark	**Data Set**	**1**	**2**	**4**	**8**	**16**
integerSort/parallelRadixSort	randomSeq_100M_int	-0,20%	0,40%	0,00%	12,30%	20,60%
	exptSeq_100M_int	0,10%	-0,10%	0,20%	10,50%	16,00%
	randomSeq_100M_int_pair_int	0,00%	4,50%	11,30%	16,60%	21,50%
	randomSeq_100M_256_int_pair_int	0,30%	20,60%	37,10%	37,00%	43,80%
comparisonSort/sampleSort	randomSeq_100M_double	0,20%	-0,10%	-0,40%	-1,10%	-0,60%
	exptSeq_100M_double	-0,60%	-1,30%	-0,50%	-1,00%	-2,40%
	almostSortedSeq_100M_double	0,00%	-0,50%	-0,40%	-0,20%	-0,90%
	randomSeq_100M_double_pair_double	-3,50%	-8,00%	0,30%	-1,40%	0,30%
	trigramSeq_100M	-4,60%	-4,60%	-4,50%	-4,90%	-2,90%
removeDuplicates/parlayhash	randomSeq_100M_int	-0,50%	0,20%	-2,50%	3,60%	7,80%
	exptSeq_100M_int	0,00%	0,80%	0,50%	-0,90%	11,30%
	trigramSeq_100M	-0,50%	2,80%	6,50%	17,70%	29,10%
histogram/parallel	randomSeq_100M_256_int	-2,90%	27,50%	18,40%	34,20%	14,80%
	randomSeq_100M_100K_int	0,50%	-0,70%	4,10%	8,80%	31,40%
	randomSeq_100M_int	-0,20%	1,40%	9,80%	11,40%	8,50%
	exptSeq_100M_int	-2,50%	1,10%	5,20%	14,80%	13,30%
	almostEqualSeq_100000000	-1,70%	1,80%	1,50%	3,90%	6,30%
wordCounts/histogram	trigramString_250000000	0,60%	2,40%	6,50%	12,40%	22,10%
	etext99	2,90%	2,40%	2,30%	3,40%	8,60%
	wikipedia250M,txt	3,50%	5,30%	7,50%	13,60%	20,40%
invertedIndex/parallel	wikisamp,xml	9,20%	5,00%	5,30%	5,90%	3,20%
	wikipedia250M,txt	2,10%	2,50%	1,00%	2,70%	2,00%
classify/decisionTree	covtype,data	1,10%	0,60%	-0,80%	-6,10%	-30,70%
	kddcup,data	0,20%	-0,80%	1,20%	-2,80%	-9,70%
spanningForest/ndST	randLocalGraph_E_10_20000000	-0,60%	-0,80%	-1,60%	-5,60%	5,70%
	rMatGraph_E_12_16000000	-0,30%	24,60%	1,90%	-2,40%	-12,10%
	2Dgrid_E_64000000	-2,00%	-1,20%	-0,90%	3,80%	3,70%
breadthFirstSearch/backForwa	randLocalGraph_J_10_20000000	1,70%	0,50%	0,10%	18,20%	-0,80%
	rMatGraph_J_12_16000000	0,30%	-0,10%	1,50%	2,10%	-4,20%
	3Dgrid_J_64000000	1,50%	-2,30%	1,50%	-9,30%	-25,30%
maximalMatching/incrementalI	randLocalGraph_E_10_20000000	0,80%	-6,00%	1,10%	85,10%	9,10%
	rMatGraph_E_10_20000000	0,90%	1,10%	0,60%	0,10%	-3,10%
	2Dgrid_E_64000000	0,90%	0,80%	1,60%	6,10%	-1,40%
maximalIndependentSet/increi	randLocalGraph_JR_10_20000000	-1,00%	6,60%	-1,30%	-9,10%	-5,40%
	rMatGraph_JR_12_16000000	1,20%	1,60%	-3,60%	-3,10%	-14,00%
	3Dgrid_JR_64000000	-0,20%	12,00%	-1,50%	-3,70%	-10,60%
nearestNeighbors/octTree	2DinCube_10000000	0,20%	0,00%	-2,10%	1,80%	-1,10%
	2Dkuzmin_10000000	0,00%	0,00%	1,30%	0,40%	-10,30%
	3DinCube_10000000	0,60%	0,20%	-2,70%	0,20%	-1,70%
	3DonSphere_10000000	1,10%	0,30%	-1,30%	0,00%	8,50%
	3DinCube_10000000	-4,10%	-3,80%	-4,20%	-3,10%	-2,20%
	3Dplummer_10000000	-4,10%	-3,10%	-2,70%	-5,40%	-4,20%
convexHull/quickHull	2DinSphere_100000000	-0,10%	-1,30%	-4,30%	-5,80%	-18,20%
	2Dkuzmin_100000000	0,20%	-1,70%	-2,80%	0,60%	-0,20%
	2DonSphere_100000000	-0,40%	-0,10%	-0,10%	-2,40%	-8,70%
delaunayTriangulation/increme	2DinCube_10M	-0,60%	-0,20%	-2,10%	-5,50%	-1,30%
	2Dkuzmin_10M	0,20%	0,30%	0,00%	-6,00%	-10,50%
delaunayRefine/incrementalRe	2DinCubeDelaunay_5000000	0,10%	1,00%	1,70%	-8,20%	-10,10%
	2DkuzminDelaunay_5000000	0,70%	0,90%	-0,10%	-6,70%	-10,70%
rangeQuery2d/parallelPlaneSv	2DinCube_10M	0,40%	-10,30%	4,90%	0,60%	-0,80%
	2Dkuzmin_10M	-3,50%	-1,30%	1,20%	0,60%	0,40%

Table 4.21: Gains and losses of the **first version** of WShare with signals over LCWS when executed on machine *AMD 16-16*.

				Intel 16-16		
Benchmark	**Data Set**	**1**	**2**	**4**	**8**	**16**
integerSort/parallelRadixSort	randomSeq_100M_int	-0,20%	0,40%	1,50%	14,40%	20,40%
	exptSeq_100M_int	0,20%	-0,20%	-1,70%	9,80%	16,00%
	randomSeq_100M_int_pair_int	0,40%	4,60%	7,20%	19,30%	20,00%
	randomSeq_100M_256_int_pair_int	-0,10%	14,80%	37,80%	41,20%	41,70%
comparisonSort/sampleSort	randomSeq_100M_double	-0,50%	-0,70%	-0,70%	-0,70%	-1,50%
	exptSeq_100M_double	-0,70%	-0,50%	-0,70%	-0,80%	-0,90%
	almostSortedSeq_100M_double	0,70%	0,60%	0,20%	-0,40%	0,30%
	randomSeq_100M_double_pair_double	-0,70%	-5,30%	1,20%	0,30%	1,00%
	trigramSeq_100M	-4,80%	-4,80%	-4,20%	-4,10%	-4,30%
removeDuplicates/parlayhash	randomSeq_100M_int	-0,20%	0,90%	0,20%	3,20%	9,30%
	exptSeq_100M_int	-0,90%	0,90%	-1,60%	-3,70%	11,60%
	trigramSeq_100M	-0,10%	1,60%	9,50%	18,70%	33,00%
histogram/parallel	randomSeq_100M_256_int	0,00%	27,50%	32,70%	34,20%	18,50%
	randomSeq_100M_100K_int	0,60%	0,10%	7,00%	7,90%	30,90%
	randomSeq_100M_int	0,10%	0,80%	9,60%	11,10%	11,50%
	exptSeq_100M_int	-2,50%	0,50%	4,10%	14,60%	14,90%
	almostEqualSeq_100000000	-1,70%	1,70%	2,00%	5,50%	6,30%
wordCounts/histogram	trigramString_250000000	0,40%	2,60%	5,90%	12,90%	22,50%
	etext99	1,60%	1,50%	1,00%	2,30%	7,80%
	wikipedia250M,txt	1,90%	3,50%	6,50%	12,70%	20,50%
invertedIndex/parallel	wikisamp,xml	12,60%	7,60%	8,90%	8,90%	5,80%
	wikipedia250M,txt	3,10%	3,50%	2,40%	3,90%	2,80%
classify/decisionTree	covtype,data	1,00%	0,30%	-1,40%	-6,60%	-75,70%
	kddcup,data	1,20%	-0,80%	1,20%	-2,30%	-14,10%
spanningForest/ndST	randLocalGraph_E_10_20000000	-0,40%	-1,20%	-0,50%	-7,60%	9,60%
	rMatGraph_E_12_16000000	0,10%	24,70%	0,00%	-9,20%	-0,20%
	2Dgrid_E_64000000	-0,70%	-1,10%	0,00%	4,00%	3,50%
breadthFirstSearch/backForwa	randLocalGraph_J_10_20000000	0,50%	0,90%	0,50%	18,80%	1,40%
	rMatGraph_J_12_16000000	0,00%	0,20%	-2,70%	1,20%	-0,70%
	3Dgrid_J_64000000	1,30%	2,90%	0,20%	-8,40%	-23,30%
maximalMatching/incrementalI	randLocalGraph_E_10_20000000	1,90%	1,80%	0,50%	-1,60%	-3,10%
	rMatGraph_E_10_20000000	1,80%	1,40%	0,10%	0,20%	-8,50%
	2Dgrid_E_64000000	1,70%	-2,50%	2,50%	3,30%	-1,00%
maximalIndependentSet/increi	randLocalGraph_JR_10_20000000	-1,30%	5,30%	-6,10%	-12,30%	-6,40%
	rMatGraph_JR_12_16000000	1,40%	0,70%	-7,60%	-6,50%	-7,80%
	3Dgrid_JR_64000000	-0,80%	11,80%	-1,30%	-8,30%	-6,30%
nearestNeighbors/octTree	2DinCube_10000000	0,40%	0,30%	-7,10%	1,30%	-0,90%
	2Dkuzmin_10000000	0,10%	0,00%	3,90%	-0,80%	-3,50%
	3DinCube_10000000	0,00%	-0,20%	-0,30%	-4,00%	-0,70%
	3DonSphere_10000000	1,00%	0,20%	-0,60%	-0,30%	7,90%
	3DinCube_10000000	-4,40%	-3,40%	0,00%	-3,00%	0,50%
	3Dplummer_10000000	-3,70%	-2,60%	-0,60%	-6,00%	-6,70%
convexHull/quickHull	2DinSphere_100000000	-0,30%	0,00%	-5,20%	1,90%	-31,30%
	2Dkuzmin_100000000	0,20%	-0,30%	-0,10%	1,00%	-16,60%
	2DonSphere_100000000	0,20%	0,00%	-0,80%	-6,40%	-26,40%
delaunayTriangulation/increme	2DinCube_10M	-0,90%	0,80%	-3,10%	-4,60%	-6,70%
	2Dkuzmin_10M	0,50%	0,40%	-0,40%	-4,60%	-13,20%
delaunayRefine/incrementalRe	2DinCubeDelaunay_5000000	-0,90%	0,80%	2,00%	-4,40%	-11,80%
	2DkuzminDelaunay_5000000	1,10%	0,60%	0,00%	-6,90%	-11,10%
rangeQuery2d/parallelPlaneSv	2DinCube_10M	0,80%	-1,20%	6,10%	-0,10%	1,40%
	2Dkuzmin_10M	-2,90%	-1,60%	2,30%	0,30%	0,30%

Table 4.22: Gains and losses of the **second version** of WShare with signals over LCWS when executed on machine *AMD 16-16*.

that of LCWS and then there are one or two benchmarks where it actually performs better. On the positive side, it actually manages to have even less unused updates than it would without signals. Figure 4.11 shows that, for a high number of threads, there are almost no unused updates. This results in a lower number of memory fences but the execution times do not really make it worth.

When compared with the SBLCWS, this version manages to have more successful steals, less time spent idle, less memory fences and less CAS operations. The problem here is due to the excessive number of interruptions and costs associated with signals. In SBLCWS we are very strict as to when signals can be sent and part of these restrictions were lifted so that this algorithm would allow processors to spread work without too many obstacles. This path however, may not have been very viable. It also removes the purpose of some heuristics. After evaluating our results, we can say that a more imposing heuristic would be better suited to this approach. We allowed too many signals and experimenting with more restrictive heuristics for this version of the algorithm could be a path for future work.

4.7 Summary of the results

In order to summarize what algorithm should be used for each benchmark, regardless of the number of threads, we present three tables (one for each machine), where WSteal is considered as the comparison factor. The Tables 4.23, 4.24 and 4.25 show that WSteal is a suitable choice in a few cases. It shows just how flexible the canonical algorithm can be against multiple variants. The algorithms presented throughout this paper had very bad results for some benchmarks **when using a high number of threads** and still managed to **remain competitive** with the WSteal algorithm. The remainder of the graphs that show execution times can be found in Appendix A.

		WS	LCWS	LCWSS1	LCWSS2	Work Giving TFT	Work Giving C-TFT	Work Giving Signals1	Work Giving Signals2
integerSort/parallelRadixSort	randomSeq_100M_int	1	1,188	1,006	1,017	1,177	1,119	1,051	1,022
	exptSeq_100M_int	1	1,273	1,022	1,025	1,179	1,178	1,078	1,077
	randomSeq_100M_int_pair_int	1	1,132	1,016	1,003	1,05	1,151	1,053	1,064
	randomSeq_100M_256_int_pair_int	1	1,044	1,007	0,998	1,232	1,213	1,027	1,013
comparisonSort/sampleSort	randomSeq_100M_double	1	1,034	1,023	1,022	1,042	1,019	1,047	1,055
	exptSeq_100M_double	1	1,025	1,022	1,02	1,034	1	1,048	1,05
	almostSortedSeq_100M_double	1	1,046	1,064	1,022	1,067	1,041	1,079	1,061
	randomSeq_100M_double_pair_double	1	1,406	1,104	0,988	1,094	1,039	1,036	1,004
	trigramSeq_100M	1	1,296	1,078	0,976	1,042	1,014	1,015	0,997
removeDuplicates/parlayhash	randomSeq_100M_int	1	1,357	1,18	1,003	1,237	1,21	1,101	1,038
	exptSeq_100M_int	1	1,372	1,225	1,011	1,347	1,316	1,086	1,045
	trigramSeq_100M	1	1,583	1,127	0,99	1,251	1,174	1,07	0,998
histogram/parallel	randomSeq_100M_256_int	1	1,306	1,101	1,039	1,631	1,579	1,091	1,035
	randomSeq_100M_100K_int	1	1,266	1,051	0,996	1,653	1,478	1,161	1,059
	randomSeq_100M_int	1	1,231	1,084	1,031	1,699	1,492	1,084	1,038
	exptSeq_100M_int	1	1,302	1,045	0,981	1,641	1,467	1,189	1,019
	almostEqualSeq_100000000	1	1,169	1,037	1,017	1,555	1,398	1,084	1,043
wordCounts/histogram	trigramString_250000000	1	1,111	0,942	0,912	1,122	1,065	0,96	0,935
	etext99	1	1,204	0,963	0,898	1,11	1,079	0,977	0,971
	wikipedia250M,txt	1	1,049	0,93	0,907	1,102	1,087	0,971	0,936
invertedIndex/parallel	wikisamp,xml	1	1,086	1,013	0,975	1,051	1,017	1,087	1,062
	wikipedia250M,txt	1	0,978	0,935	0,926	0,963	0,988	0,975	0,961
classify/decisionTree	covtype,data	1	1,32	1,374	1,556	1,186	1,16	1,716	2,215
	kddcup,data	1	1,066	1,063	1,054	1,064	1,051	1,275	1,368
spanningForest/ndST	randLocalGraph_E_10_20000000	1	0,996	1,019	1,002	1,002	1,025	1,019	1,005
	rMatGraph_E_12_16000000	1	0,978	0,973	0,983	0,99	0,982	0,963	0,978
	2Dgrid_E_64000000	1	1,024	1,025	1,016	1,019	1,008	1,006	1,013
breadthFirstSearch/backForwa	randLocalGraph_J_10_20000000	1	0,865	0,881	0,859	0,887	0,885	0,94	0,961
	rMatGraph_J_12_16000000	1	0,898	0,921	0,894	0,909	0,925	0,983	0,967
	3Dgrid_J_64000000	1	1,606	1,645	1,299	1,4	1,414	1,905	1,624
maximalMatching/incrementalI	randLocalGraph_E_10_20000000	1	0,897	0,919	0,907	0,909	0,919	1,004	0,998
	rMatGraph_E_10_20000000	1	0,93	0,927	0,955	0,937	0,921	0,996	0,992
	2Dgrid_E_64000000	1	0,929	0,934	0,94	0,942	0,936	0,992	0,986
maximalIndependentSet/incre	randLocalGraph_JR_10_20000000	1	1,175	1,123	1,017	1,044	1,035	1,211	1,154
	rMatGraph_JR_12_16000000	1	1,32	1,117	0,999	1,03	0,993	1,194	1,137
	3Dgrid_JR_64000000	1	1,122	1,064	0,999	1,002	1,009	1,106	1,104
nearestNeighbors/octTree	2DinCube_10000000	1	0,941	0,969	0,932	0,968	0,947	1,021	1,025
	2Dkuzmin_10000000	1	0,949	0,967	0,936	1,004	0,977	0,997	1,002
	3DinCube_10000000	1	0,977	0,987	0,969	0,984	0,977	1,005	1,002
	3DonSphere_10000000	1	0,973	0,986	0,972	0,997	0,97	1,012	1,027
	3DinCube_10000000	1	1,005	0,99	0,982	0,995	0,993	0,998	0,99
	3Dplummer_10000000	1	0,978	0,994	0,968	0,974	0,984	1,007	0,991
convexHull/quickHull	2DinSphere_100000000	1	1,026	1	0,98	1,008	1,008	1,023	1,014
	2Dkuzmin_100000000	1	1,064	1,074	1,044	1,057	1,075	1,085	1,113
	2DonSphere_100000000	1	0,977	0,966	1,019	0,995	1,011	1,082	1,158
delaunayTriangulation/increme	2DinCube_10M	1	1,115	1,158	1,073	1,204	1,181	1,452	1,329
	2Dkuzmin_10M	1	1,124	1,126	1,053	1,214	1,14	1,349	1,249
delaunayRefine/incrementalRe	2DinCubeDelaunay_5000000	1	1,075	1,085	1,045	1,044	1,028	1,294	1,193
	2DkuzminDelaunay_5000000	1	1,096	1,073	1,068	1,048	1,039	1,332	1,211
rangeQuery2d/parallelPlaneSv	2DinCube_10M	1	1,028	0,989	0,937	0,978	0,96	0,998	0,974
	2Dkuzmin_10M	1	1,068	1,011	0,972	0,999	0,974	1,012	1,004

Table 4.23: General picture of what algorithm to use for each benchmark, disregarding number of threads. It is based on the execution time of the WSteal algorithm, which is always placed as **1**, meaning that being below this number is better. For machine *AMD 32-64*.

		WS	LCWS	LCWSS1	LCWSS2	Work Giving TFT	Work Giving C-TFT	Work Giving Signals1	Work Giving Signals2
integerSort/parallelRadixSort	randomSeq_100M_int	1	1,019	1,004	0,999	1,021	1,026	1,019	1,009
	exptSeq_100M_int	1	1,005	0,996	1,001	1,02	1,016	1,009	1,012
	randomSeq_100M_int_pair_int	1	1,009	0,994	1,001	1,006	1,022	1,016	1,005
	randomSeq_100M_256_int_pair_int	1	1,007	1,004	1,019	1,016	1,022	1,012	1,016
comparisonSort/sampleSort	randomSeq_100M_double	1	1,017	1	1,029	0,997	0,999	1,035	1,003
	exptSeq_100M_double	1	1,014	1,001	1,019	0,997	1,001	1,028	1,005
	almostSortedSeq_100M_double	1	1,013	0,99	0,996	1,001	1	0,998	1,005
	randomSeq_100M_double_pair_double	1	1,019	0,993	0,992	0,999	1,001	1,003	0,994
	trigramSeq_100M	1	0,943	0,985	0,985	0,939	0,928	0,987	0,99
removeDuplicates/parlayhasq	randomSeq_100M_int	1	1,02	1,005	0,994	1,037	1,047	0,999	1
	exptSeq_100M_int	1	1,023	0,99	1,01	1,039	1,063	0,989	1
	trigramSeq_100M	1	1,242	1,014	0,999	1,234	1,284	1,008	1,008
histogram/parallel	randomSeq_100M_256_int	1	1,424	1,074	1,011	1,594	1,793	1,059	1,027
	randomSeq_100M_100K_int	1	1,05	1,004	0,998	1,118	1,144	1,016	1,007
	randomSeq_100M_int	1	1,104	1,032	1	1,203	1,268	1,027	1,002
	exptSeq_100M_int	1	1,104	1,002	0,988	1,176	1,213	1,014	0,99
	almostEqualSeq_100000000	1	1,081	1,009	0,992	1,144	1,181	1,009	0,996
wordCounts/histogram	trigramString_250000000	1	0,947	0,913	0,906	0,947	0,955	0,915	0,916
	etext99	1	0,933	0,892	0,903	0,932	0,961	0,907	0,911
	wikipedia250M.txt	1	0,921	0,886	0,896	0,918	0,936	0,892	0,903
invertedIndex/parallel	wikisamp.xml	1	0,968	0,945	0,966	0,957	0,961	0,982	0,97
	wikipedia250M.txt	1	0,955	0,931	0,934	0,939	0,948	5191,127	0,938
classify/decisionTree	covtype.data	1	1,07	1,131	1,217	1,104	1,09	1,292	1,552
	kddcup.data	1	1,012	1,012	0,997	1,024	1,03	1,051	1,07
spanningForest/ndST	randLocalGraph_E_10_20000000	1	1,001	0,993	1	1,007	0,994	1	0,995
	rMatGraph_E_12_16000000	1	1,005	0,991	0,991	0,991	1,038	1,003	0,991
	2Dgrid_E_64000000	1	1,012	0,997	0,997	0,997	0,997	1,002	0,999
breadthFirstSearch/backForwa	randLocalGraph_J_10_20000000	1	0,915	0,937	0,922	0,903	0,901	0,946	0,938
	rMatGraph_J_12_16000000	1	0,919	0,897	0,908	0,91	0,9	0,926	0,925
	3Dgrid_J_64000000	1	1,078	1,097	1,101	1,15	1,164	1,267	1,25
maximalMatching/incrementalI	randLocalGraph_E_10_20000000	1	0,88	0,865	0,882	0,868	0,885	0,894	0,891
	rMatGraph_E_10_20000000	1	0,9	0,881	0,887	0,887	0,887	0,903	0,903
	2Dgrid_E_64000000	1	0,916	0,897	0,902	0,901	0,901	0,917	0,919
maximalIndependentSet/incren	randLocalGraph_JR_10_20000000	1	0,951	1,001	0,968	0,972	0,971	1,064	1,057
	rMatGraph_JR_12_16000000	1	0,953	0,98	0,992	0,977	0,979	1,081	1,092
	3Dgrid_JR_64000000	1	0,935	0,941	0,945	0,945	0,95	0,995	1,001
nearestNeighbors/octTree	2DinCube_10000000	1	0,972	0,944	0,954	0,958	0,951	0,964	0,964
	2Dkuzmin_10000000	1	0,973	0,948	0,958	0,966	0,961	0,972	0,972
	3DinCube_10000000	1	0,966	0,957	0,962	0,958	0,963	0,968	0,968
	3DonSphere_10000000	1	0,963	0,952	0,954	0,955	0,959	0,962	0,97
	3DinCube_10000000	1	0,995	0,979	0,979	0,987	0,982	0,982	0,984
	3Dplummer_10000000	1	0,983	0,974	0,975	0,972	0,974	0,98	0,981
convexHull/quickHull	2DinSphere_100000000	1	1,003	0,968	0,966	0,985	1,023	1,017	1,025
	2Dkuzmin_100000000	1	1,011	0,997	0,999	1,027	1,036	1,033	1,031
	2DonSphere_100000000	1	1,013	0,996	1,015	1,008	1,014	1,052	1,13
delaunayTriangulation/increme	2DinCube_10M	1	1,057	1,035	1,036	1,094	1,104	1,131	1,178
	2Dkuzmin_10M	1	1,058	1,035	1,04	1,097	1,112	1,121	1,163
delaunayRefine/incrementalRe	2DinCubeDelaunay_5000000	1	1,003	1,013	1,014	1,031	1,031	1,113	1,134
	2DkuzminDelaunay_5000000	1	1	1,009	1,008	1,023	1,02	1,106	1,133
rangeQuery2d/parallelPlaneSv	2DinCube_10M	1	0,973	0,946	0,948	0,947	0,946	0,956	0,951
	2Dkuzmin_10M	1	0,969	0,943	0,94	0,944	0,934	0,951	0,956

Table 4.24: General picture of what algorithm to use for each benchmark, disregarding number of threads. It is based on the execution time of the WSteal algorithm, which is always placed as **1**, meaning that being below this number is better. For machine *Intel 12-24*.

		WS	LCWS	LCWSS1	LCWSS2	Work Giving TFT	Work Giving C-TFT	Work Giving Signals1	Work Giving Signals2
integerSort/parallelRadixSort	randomSeq_100M_int	1	1,077	0,989	0,99	1,082	1,077	0,995	0,988
	exptSeq_100M_int	1	1,044	0,987	1,014	1,06	1,059	0,984	0,989
	randomSeq_100M_int_pair_int	1	1,129	0,996	1,006	1,135	1,132	0,999	1,004
	randomSeq_100M_256_int_pair_int	1	1,463	0,997	1,039	1,472	1,471	1,012	1,018
comparisonSort/sampleSort	randomSeq_100M_double	1	0,995	1,006	1,002	0,997	1,006	0,999	1,003
	exptSeq_100M_double	1	0,996	1,001	1,002	0,999	1,003	1,007	1,003
	almostSortedSeq_100M_double	1	1,009	1,004	1,011	1,003	1,01	1,013	1,006
	randomSeq_100M_double_pair_double	1	0,984	1,01	0,989	1	1,002	1,008	0,99
	trigramSeq_100M	1	0,944	0,985	0,987	0,939	0,944	0,984	0,986
removeDuplicates/parlayhab	randomSeq_100M_int	1	1,02	1,006	0,982	1,033	1,026	1,001	0,991
	exptSeq_100M_int	1	1,027	0,996	1,01	1,06	1,053	1	1,011
	trigramSeq_100M	1	1,164	1,001	0,988	1,195	1,204	1,016	0,997
histogram/parallel	randomSeq_100M_256_int	1	1,3	1,04	1,014	1,461	1,368	1,041	0,984
	randomSeq_100M_100K_int	1	1,119	1,005	1,013	1,15	1,151	1	0,996
	randomSeq_100M_int	1	1,069	1,002	0,989	1,135	1,108	1	0,994
	exptSeq_100M_int	1	1,078	1,009	1,018	1,117	1,124	1,004	1,004
	almostEqualSeq_100000000	1	1,037	1,012	1,008	1,097	1,095	1,011	1,007
wordCounts/histogram	trigramString_250000000	1	0,998	0,904	0,905	1,007	1,002	0,902	0,901
	etext99	1	0,899	0,876	0,871	0,929	0,906	0,863	0,872
	wikipedia250M.txt	1	0,956	0,874	0,862	0,985	0,951	0,854	0,863
invertedIndex/parallel	wikisamp.xml	1	1,015	0,922	1,018	1,018	1,02	0,957	0,926
	wikipedia250M.txt	1	0,941	0,922	0,927	0,943	0,944	0,921	0,911
classify/decisionTree	covtype.data	1	1,038	1,052	1,098	1,059	1,061	1,122	1,235
	kddcup.data	1	1,003	1,01	0,992	1,016	1,019	1,026	1,032
spanningForest/ndST	randLocalGraph_E_10_20000000	1	0,999	0,977	0,996	1,032	0,861	1,005	0,999
	rMatGraph_E_12_16000000	1	1,043	0,966	0,97	0,984	0,976	1,002	0,995
	2Dgrid_E_64000000	1	1,01	1,08	0,998	1,036	0,995	1,002	0,998
breadthFirstSearch/backForwa	randLocalGraph_J_10_20000000	1	0,933	0,896	0,878	0,886	0,897	0,891	0,886
	rMatGraph_J_12_16000000	1	0,868	0,866	0,854	0,863	0,876	0,87	0,872
	3Dgrid_J_64000000	1	0,985	1,005	1,01	1,033	1,042	1,058	1,045
maximalMatching/incrementalI	randLocalGraph_E_10_20000000	1	0,832	0,806	0,804	0,854	0,838	0,685	0,835
	rMatGraph_E_10_20000000	1	0,843	0,86	0,863	0,872	0,916	0,844	0,853
	2Dgrid_E_64000000	1	0,878	0,872	0,866	0,872	0,875	0,864	0,871
maximalIndependentSet/incre	randLocalGraph_JR_10_20000000	1	0,945	0,999	0,938	0,971	0,96	0,964	0,984
	rMatGraph_JR_12_16000000	1	0,947	1,03	0,96	0,998	0,955	0,983	0,985
	3Dgrid_JR_64000000	1	0,941	0,952	0,939	0,943	0,978	0,948	0,949
nearestNeighbors/octTree	2DinCube_10000000	1	0,948	0,94	0,941	0,942	0,979	0,95	0,959
	2Dkuzmin_10000000	1	0,953	0,944	0,958	0,942	0,966	0,969	0,953
	3DinCube_10000000	1	0,96	0,961	0,94	0,947	0,979	0,966	0,97
	3DonSphere_10000000	1	1,004	0,975	0,946	0,953	1,023	0,985	0,986
	3DinCube_10000000	1	0,939	0,94	0,972	0,987	0,948	0,972	0,958
	3Dplummer_10000000	1	0,928	0,933	0,968	0,944	0,912	0,965	0,965
convexHull/quickHull	2DinSphere_100000000	1	0,91	0,94	0,942	0,938	0,937	0,96	0,965
	2Dkuzmin_100000000	1	0,983	0,968	0,977	0,99	0,974	0,991	1,017
	2DonSphere_100000000	1	0,995	0,993	1,003	0,999	0,994	1,019	1,063
delaunayTriangulation/increme	2DinCube_10M	1	1,024	1,028	1,017	1,046	1,05	1,044	1,055
	2Dkuzmin_10M	1	1,02	1,025	1,027	1,06	1,052	1,053	1,056
delaunayRefine/incrementalRe	2DinCubeDelaunay_5000000	1	0,976	0,98	0,978	0,987	0,991	1,006	1,004
	2DkuzminDelaunay_5000000	1	0,981	0,987	0,991	0,996	0,992	1,012	1,013
rangeQuery2d/parallelPlaneSv	2DinCube_10M	1	0,954	0,937	0,946	0,94	0,955	0,963	0,94
	2Dkuzmin_10M	1	0,92	0,923	0,937	0,932	0,931	0,924	0,923

Table 4.25: General picture of what algorithm to use for each benchmark, disregarding number of threads. It is based on the execution time of the WSteal algorithm, which is always placed as **1**, meaning that being below this number is better. For machine *Intel 16-16*.

5

Conclusions

We will now outline the major contributions made in this thesis.

Our first principal contribution lies in showing that the Low-Cost Work Stealing (LCWS) algorithm actually performs slightly better than the Work Stealing (WSteal) algorithm. It was already expected that it would provide better results based on the nature of the algorithm and the very reduced synchronization that it imposes. Now however, we are able to see that these are partly brought down due to the bad results when using a high number of threads. Fast computations are thus inneficient due to the rather slow start regarding load balancing. It will always take more computation steps than the canonical WSteal to balance, across all processors, the initial load from the root.

Secondly, we showed how versatile the LCWS algorithm is. It manages to get positive results in most benchmarks that were used for testing (when not using the total amount of threads of the CPU). We also showed that it manages to avoid most synchronization overheads and why. It is also easily adaptable by altering how its load balancing works. It can be tailored to be more efficient for specific types of workloads. By switching some notions around, a different algorithm is created, which might better suit the user's needs. There is still room for improvement and experimentation to be done in future works. There are always more balancing policies that can be made and tested, to alter the flow of the algorithm.

Our experimentation with signals also shows that there is more to be gained in this area. Most algorithms force users to prepare their workload when it is too coarse, as most algorithms are made with fine-grained tasks in mind. By using signals, one can completely eliminate this need, without much change to be done in the algorithm. We did not delve too deeply into the tiny intricacies revolving around this tool. Nevertheless, we can say that there is more to be gained from their proper utilization.

We showed how work giving can be impactful. It is usually associated with being the worst of the two types of algorithms. Which is usually the case. However, we, like others before us, have shown that there are some gains to be made by mixing the two. We explored few heuristics and there are more yet to be surveyed.

Lastly, we were able to show that all algorithms proposed have some inherent advantages depending on the workload and machine used. An algorithm cannot be disregarded because of some bad results. We believe we have not reached the limit where code complexity harms performance. Even by introducing signals that interrupt threads, the algorithms still produced positive results.

Bibliography

[1] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. "The Data Locality of Work Stealing". In: *Theory Comput. Syst.* 35.3 (2002), pp. 321–347. DOI: 10.1007/s00224-002-1057-3. URL: https://doi.org/10.1007/s00224-002-1057-3 (cit. on p. 12).

[2] U. A. Acar, A. Charguéraud, and M. Rainey. "Scheduling parallel programs by work stealing with private deques". In: *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*. Ed. by A. Nicolau et al. ACM, 2013, pp. 219–228. DOI: 10.1145/2442516.2442538. URL: https://doi.org/10.1145/2442516.2442538 (cit. on p. 14).

[3] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. "Thread Scheduling for Multiprogrammed Multiprocessors". In: *Theory Comput. Syst.* 34.2 (2001), pp. 115–144. DOI: 10.1007/s00224-001-0004-z. URL: https://doi.org/10.1007/s00224-001-0004-z (cit. on pp. 3, 11, 14, 24, 25).

[4] H. Attiya et al. "Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated". In: *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. Ed. by T. Ball and M. Sagiv. ACM, 2011, pp. 487–498. ISBN: 978-1-4503-0490-0. DOI: 10.1145/1926385.1926442. URL: https://doi.org/10.1145/1926385.1926442 (cit. on pp. 2, 15).

[5] R. D. Blumofe. "Scheduling Multithreaded Computations by Work Stealing". In: *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994*. IEEE Computer Society, 1994, pp. 356–368. DOI: 10.1109/SFCS.1994.365680. URL: https://doi.org/10.1109/SFCS.1994.365680 (cit. on pp. 2, 8, 14).

[6] R. D. Blumofe and C. E. Leiserson. "Space-Efficient Scheduling of Multithreaded Computations". In: *SIAM J. Comput.* 27.1 (1998), pp. 202–229. DOI: 10.1137/S0097539793259471. URL: https://doi.org/10.1137/S0097539793259471 (cit. on p. 8).

[7] F. W. Burton. "Storage Management in Virtual Tree Machines". In: *IEEE Trans. Computers* 37.3 (1988), pp. 321–328. DOI: 10.1109/12.2169. URL: https://doi.org/10.1109/12.2169 (cit. on p. 6).

[8] V. R. B. G. Caldiera and H. D. Rombach. "The goal question metric approach". In: *Encyclopedia of software engineering* (1994), pp. 528–532 (cit. on p. 43).

[9] D. Chase and Y. Lev. "Dynamic circular work-stealing deque". In: *SPAA 2005: Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures, July 18-20, 2005, Las Vegas, Nevada, USA*. Ed. by P. B. Gibbons and P. G. Spirakis. ACM, 2005, pp. 21–28. DOI: 10.1145/1073970.1073974. URL: https://doi.org/10.1145/1073970.1073974 (cit. on pp. 11, 25).

[10] D. Dechev, P. Pirkelbauer, and B. Stroustrup. "Understanding and Effectively Preventing the ABA Problem in Descriptor-Based Lock-Free Designs". In: *13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, ISORC 2010, Carmona, Sevilla, Spain, 5-6 May 2010*. IEEE Computer Society, 2010, pp. 185–192. DOI: 10.1109/ISORC.2010.10. URL: https://doi.org/10.1109/ISORC.2010.10 (cit. on p. 23).

[11] T. van Dijk and J. C. van de Pol. "Lace: Non-blocking Split Deque for Work-Stealing". In: *Euro-Par 2014: Parallel Processing Workshops - Euro-Par 2014 International Workshops, Porto, Portugal, August 25-26, 2014, Revised Selected Papers, Part II*. Ed. by L. M. B. Lopes et al. Vol. 8806. Lecture Notes in Computer Science. Springer, 2014, pp. 206–217. DOI: 10.1007/978-3-319-14313-2_18. URL: https://doi.org/10.1007/978-3-319-14313-2%5C_18 (cit. on pp. 16, 25).

[12] Y. Guo et al. "SLAW: a scalable locality-aware adaptive work-stealing scheduler for multi-core systems". In: *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2010, Bangalore, India, January 9-14, 2010*. Ed. by R. Govindarajan, D. A. Padua, and M. W. Hall. ACM, 2010, pp. 341–342. DOI: 10.1145/1693453.1693504. URL: https://doi.org/10.1145/1693453.1693504 (cit. on pp. 10, 12, 13).

[13] D. Hendler and N. Shavit. "Non-blocking steal-half work queues". In: *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Distributed Computing, PODC 2002, Monterey, California, USA, July 21-24, 2002*. Ed. by A. Ricciardi. ACM, 2002, pp. 280–289. DOI: 10.1145/571825.571876. URL: https://doi.org/10.1145/571825.571876 (cit. on pp. 3, 14, 25).

[14] N. M. Lê et al. "Correct and efficient work-stealing for weak memory models". In: *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*. Ed. by A. Nicolau et al. ACM, 2013, pp. 69–80. DOI: 10.1145/2442516.2442524. URL: https://doi.org/10.1145/2442516.2442524 (cit. on p. 25).

[15] L. M. B. Lopes and F. M. A. Silva. "Thread- and Process-Based Implementations of the pSystem Parallel Programming Environment". In: *Softw. Pract. Exp.* 27.3 (1997), pp. 329–351. DOI: 10.1002/(SICI)1097-024X(199703)27:3\<329::AID-SPE90\>3.0.CO;2-9. URL: https://doi.org/10.1002/(SICI)1097-024X(199703)27:3%5C%3C329::AID-SPE90%5C%3E3.0.CO;2-9 (cit. on p. 18).

[16] A. Morrison and Y. Afek. "Fence-free work stealing on bounded TSO processors". In: *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2014, Salt Lake City, UT, USA, March 1-5, 2014*. Ed. by R. Balasubramonian, A. Davis, and S. V. Adve. ACM, 2014, pp. 413–426. DOI: 10.1145/2541940.2541987. URL: https://doi.org/10.1145/2541940.2541987 (cit. on p. 15).

[17] S. K. Muller and U. A. Acar. "Latency-Hiding Work Stealing: Scheduling Interacting Parallel Computations with Work Stealing". In: *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, Asilomar State Beach/Pacific Grove, CA, USA, July 11-13, 2016*. Ed. by C. Scheideler and S. Gilbert. ACM, 2016, pp. 71–82. DOI: 10.1145/2935764.2935793. URL: https://doi.org/10.1145/2935764.2935793 (cit. on p. 16).

[18] J. Paudel, O. Tardieu, and J. N. Amaral. "Hybrid parallel task placement in X10". In: *Proceedings of the third ACM SIGPLAN X10 Workshop, X10 2013, Seattle, Washington, USA, June 20, 2013*. Ed. by M. Hind and D. Grove. ACM, 2013, pp. 31–38. DOI: 10.1145/2481268.2481277. URL: https://doi.org/10.1145/2481268.2481277 (cit. on p. 17).

[19] M. Raynal. *Concurrent programming: algorithms, principles, and foundations*. Springer Science & Business Media, 2012 (cit. on p. 5).

[20] G. Rito and H. Paulino. "Scheduling computations with provably low synchronization overheads". In: *Journal of Scheduling* (2021). DOI: 10.1007/s10951-021-00706-6. URL: https://doi.org/10.1007/s10951-021-00706-6 (cit. on pp. 3, 20–23, 37, 39).

[21] L. Rudolph, M. Slivkin-Allalouf, and E. Upfal. "A Simple Load Balancing Scheme for Task Allocation in Parallel Machines". In: *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '91, Hilton Head, South Carolina, USA, July 21-24, 1991*. Ed. by T. Leighton. ACM, 1991, pp. 237–245. DOI: 10.1145/113379.113401. URL: https://doi.org/10.1145/113379.113401 (cit. on p. 14).

[22] P. Sewell et al. "x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors." In: *Commun. ACM* 53.7 (2010), pp. 89–97. URL: http://dblp.uni-trier.de/db/journals/cacm/cacm53.html#SewellSONM10 (cit. on p. 29).

[23] F. M. A. Silva, H. Paulino, and L. M. B. Lopes. "Di_pSystem: A Parallel Programming System for Distributed Memory Architectures". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 6th European PVM/MPI Users' Group Meeting, Barcelona, Spain, September 26-29, 1999, Proceedings*. Ed. by J. J. Dongarra, E. Luque, and T. Margalef. Vol. 1697. Lecture Notes in Computer Science. Springer, 1999, pp. 525–532. DOI: 10.1007/3-540-48158-3_65. URL: https://doi.org/10.1007/3-540-48158-3%5C_65 (cit. on p. 18).

[24] A. Tzannes et al. "Lazy Scheduling: A Runtime Adaptive Scheduler for Declarative Parallelism". In: *ACM Trans. Program. Lang. Syst.* 36.3 (2014), 10:1–10:51. DOI: 10.1145/2629643. URL: https://doi.org/10.1145/2629643 (cit. on p. 13).

A

Benchmark Execution Times

A.1 Integer Sort

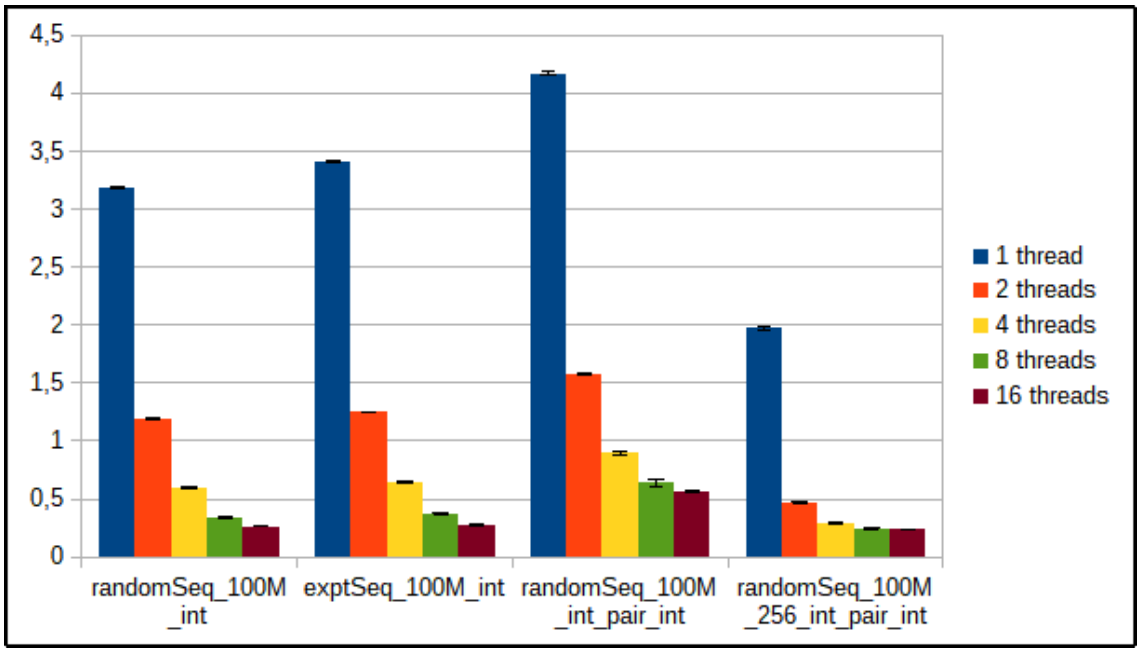


Figure A.1: Execution times (in minutes) of Integer Sort ran on Intel 12-24 using WSteal

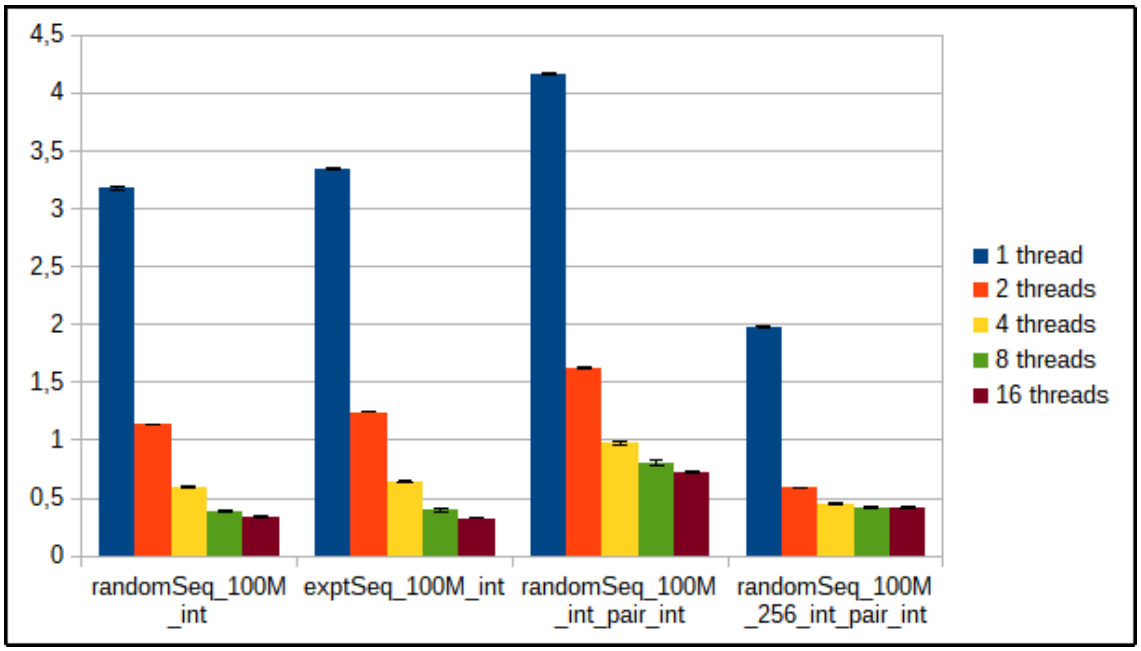


Figure A.2: Execution times (in minutes) of Integer Sort ran on Intel 12-24 using LCWS

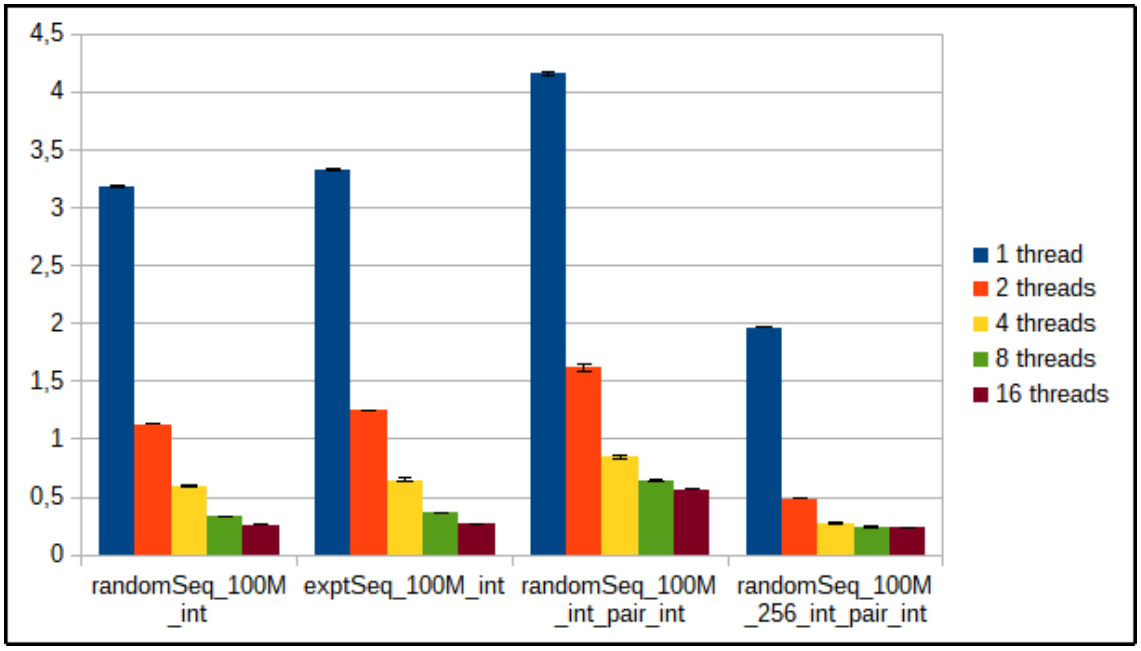


Figure A.3: Execution times (in minutes) of Integer Sort ran on Intel 12-24 using Signal-Based Low-Cost Work Stealing (SBLCWS) (**first version**)

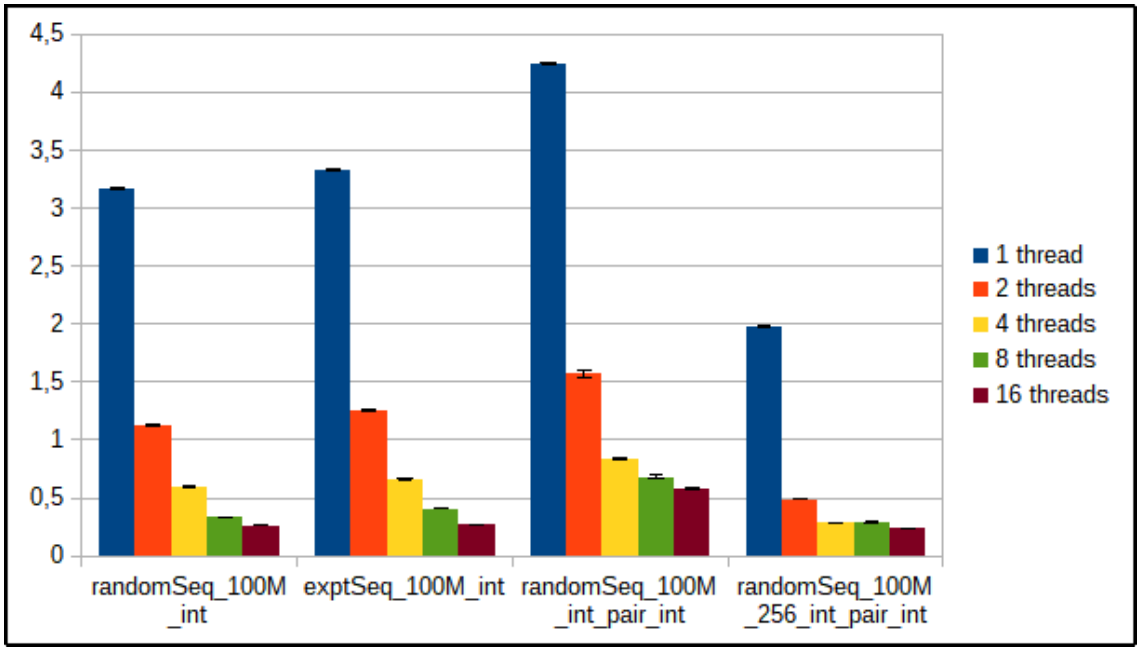


Figure A.4: Execution times (in minutes) of Integer Sort ran on Intel 12-24 using SBLCWS (**second version**)

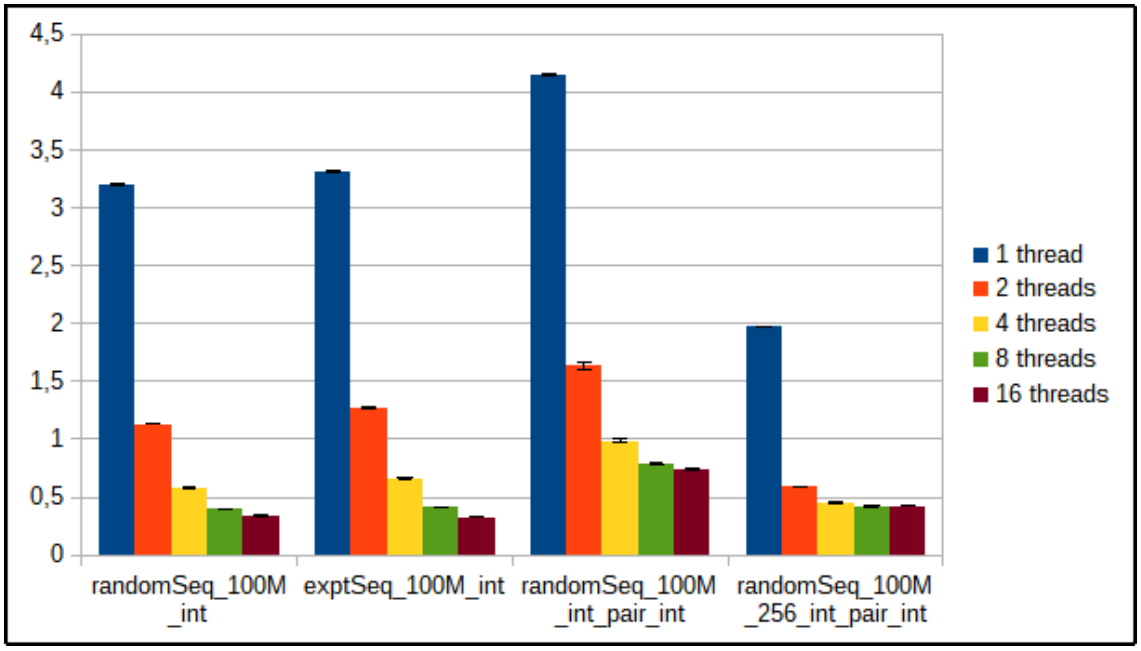


Figure A.5: Execution times (in minutes) of Integer Sort ran on Intel 12-24 using LCWS with Work Sharing (WShare) (**tit-for-tat**)

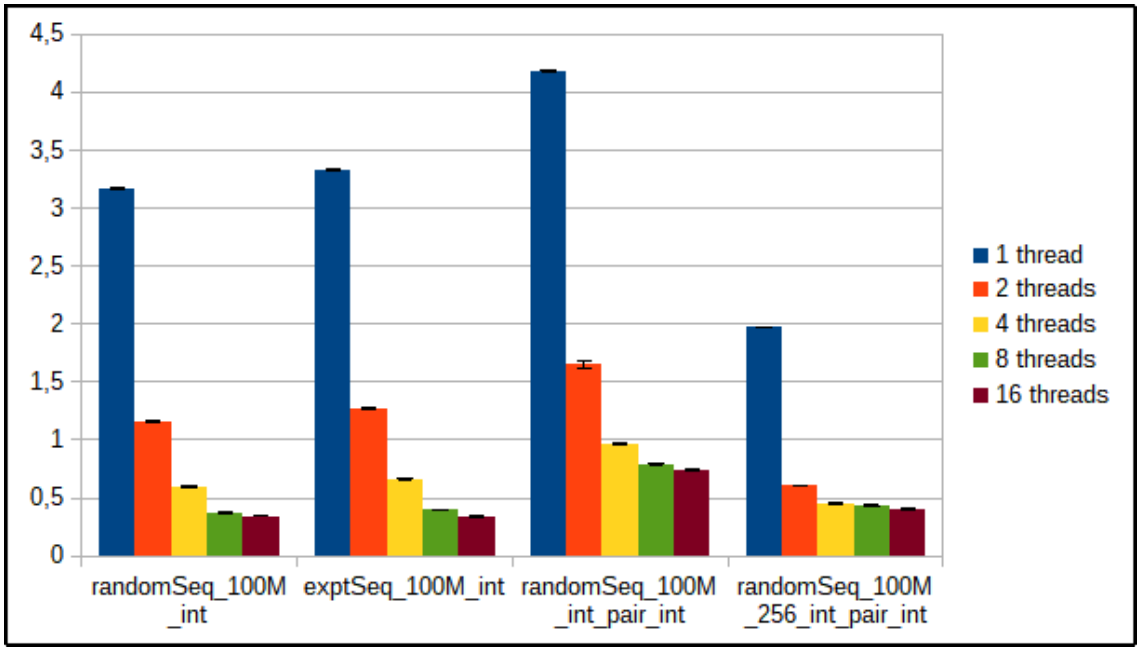


Figure A.6: Execution times (in minutes) of Integer Sort ran on Intel 12-24 using LCWS with WShare (**continuous tit-for-tat**)

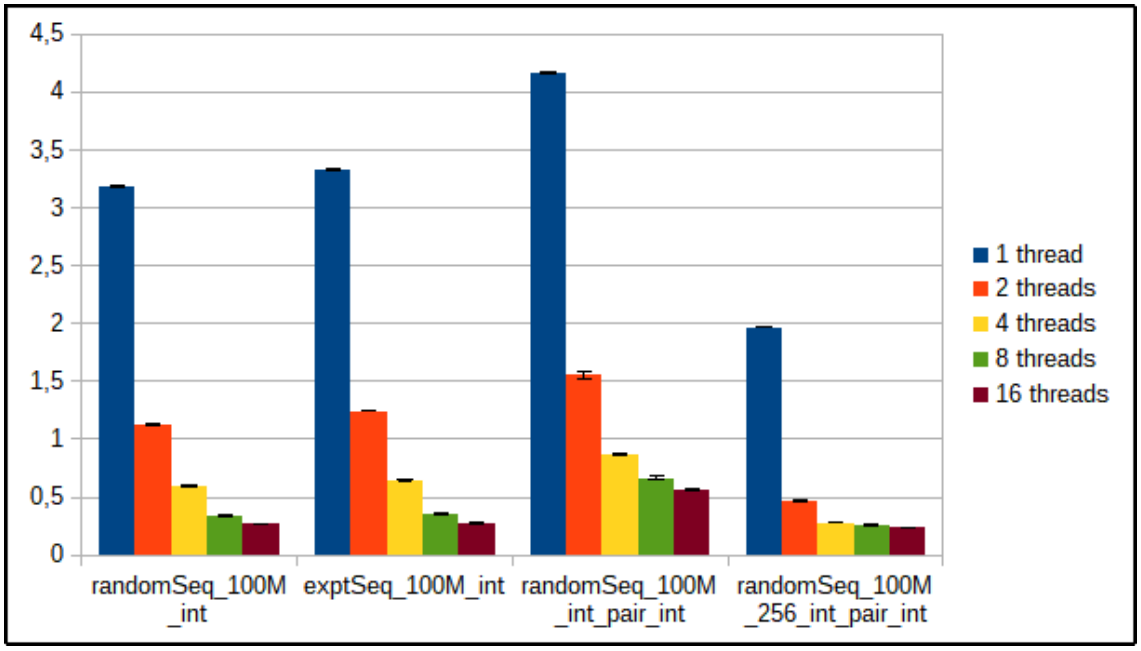


Figure A.7: Execution times (in minutes) of Integer Sort ran on Intel 12-24 using SBLCWS with WShare(**first version**)

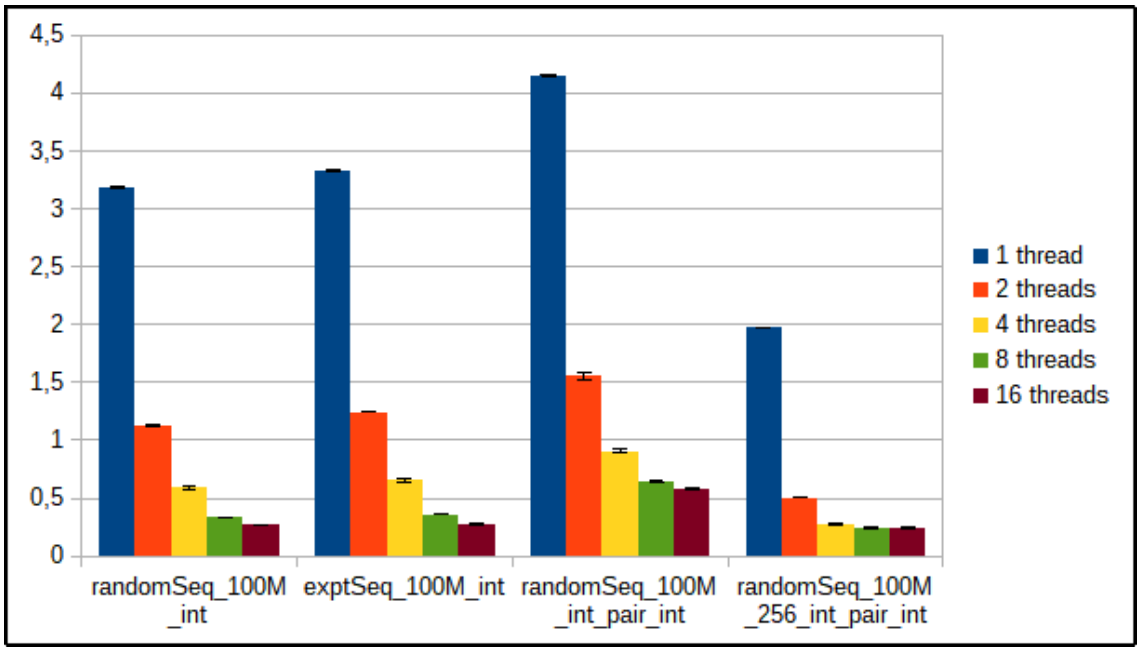


Figure A.8: Execution times (in minutes) of Integer Sort ran on Intel 12-24 using SBLCWS with WShare (**second version**)

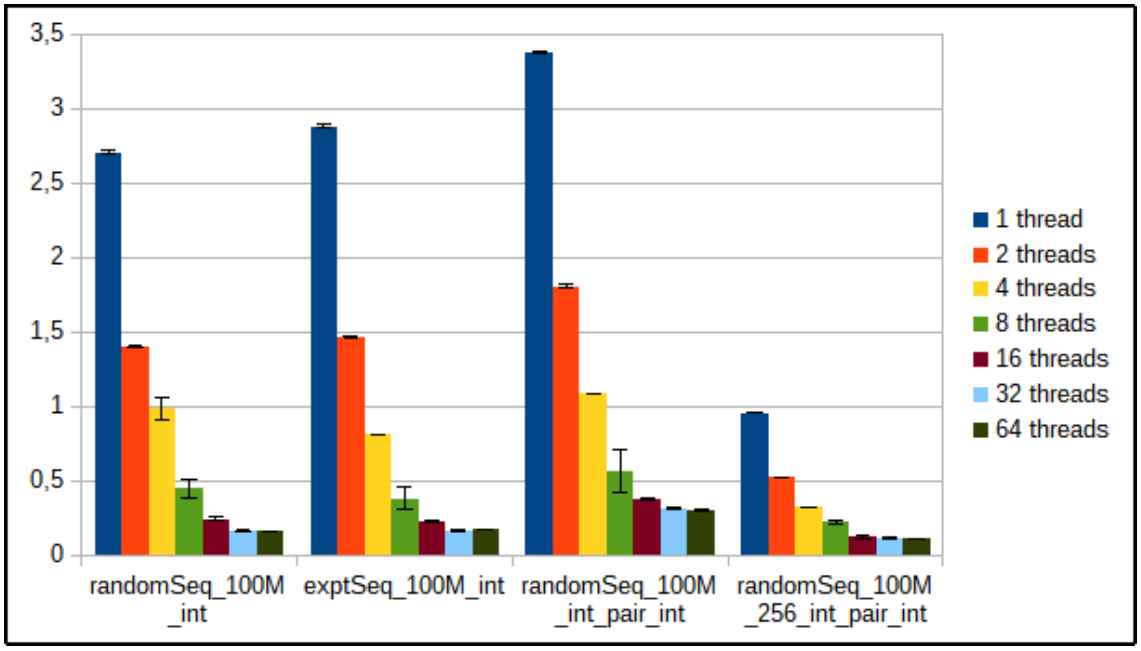


Figure A.9: Execution times (in minutes) of Integer Sort ran on AMD 32-64 using WSteal

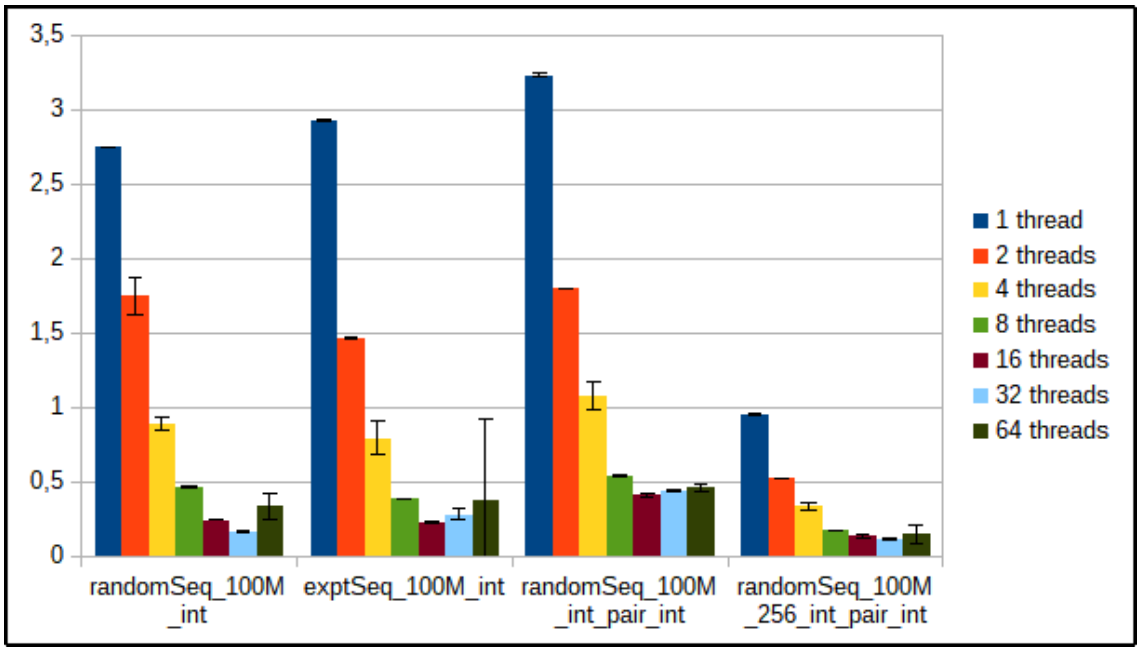


Figure A.10: Execution times (in minutes) of Integer Sort ran on AMD 32-64 using LCWS

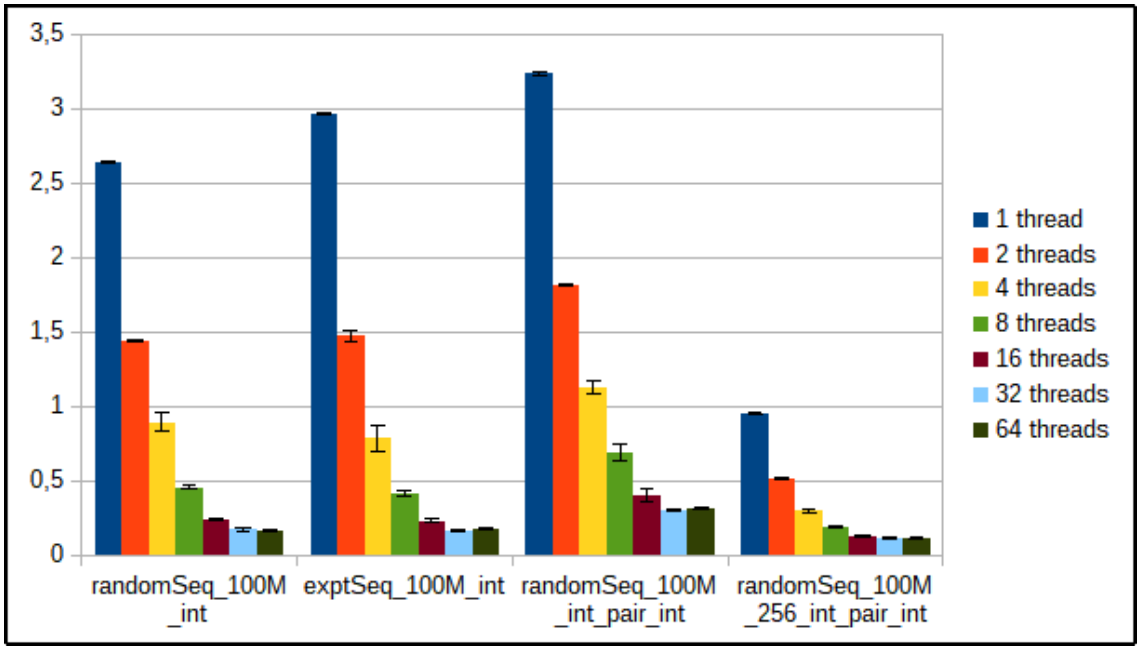


Figure A.11: Execution times (in minutes) of Integer Sort ran on AMD 32-64 using SBLCWS (**first version**)

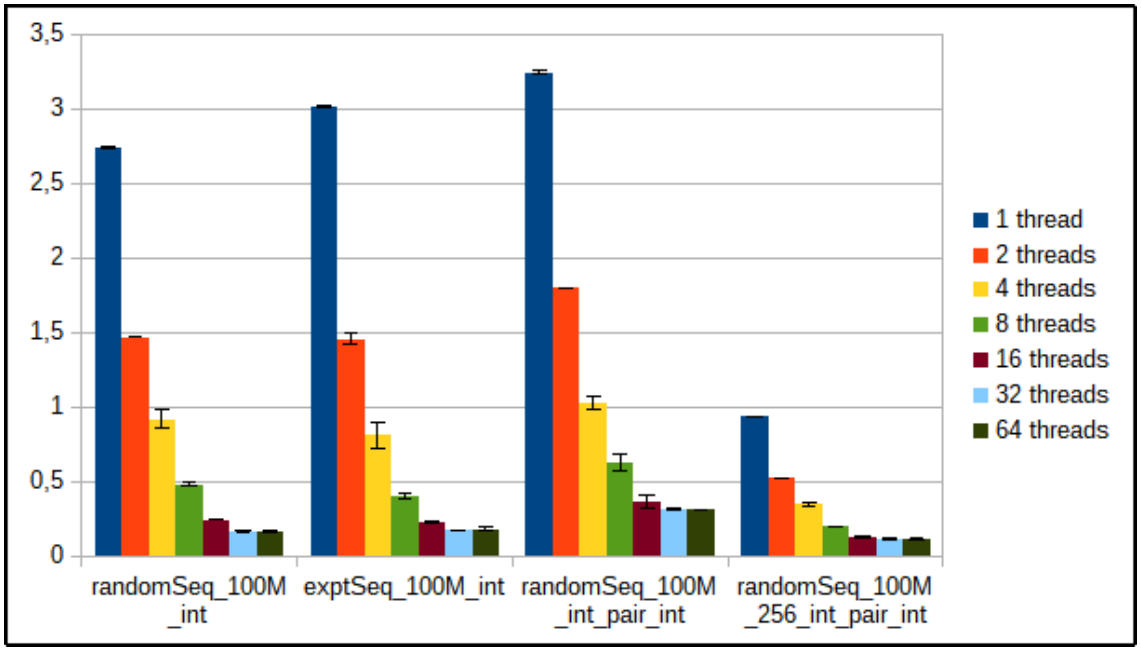


Figure A.12: Execution times (in minutes) of Integer Sort ran on AMD 32-64 using SBLCWS (**second version**)

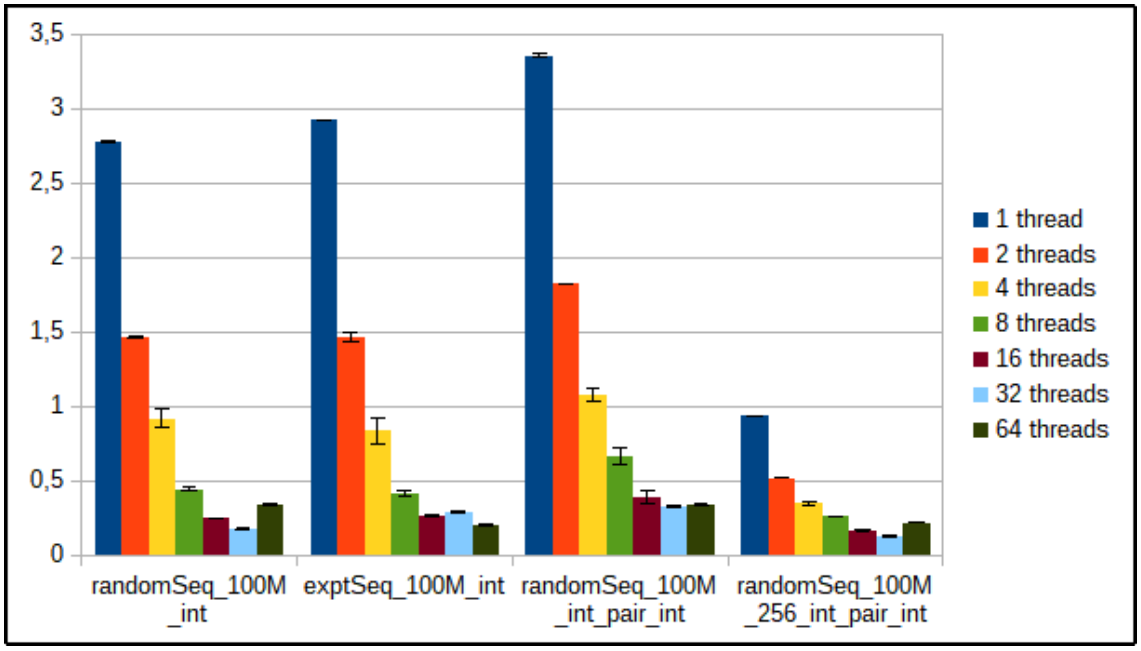


Figure A.13: Execution times (in minutes) of Integer Sort ran on AMD 32-64 using LCWS with WShare (**tit-for-tat**)

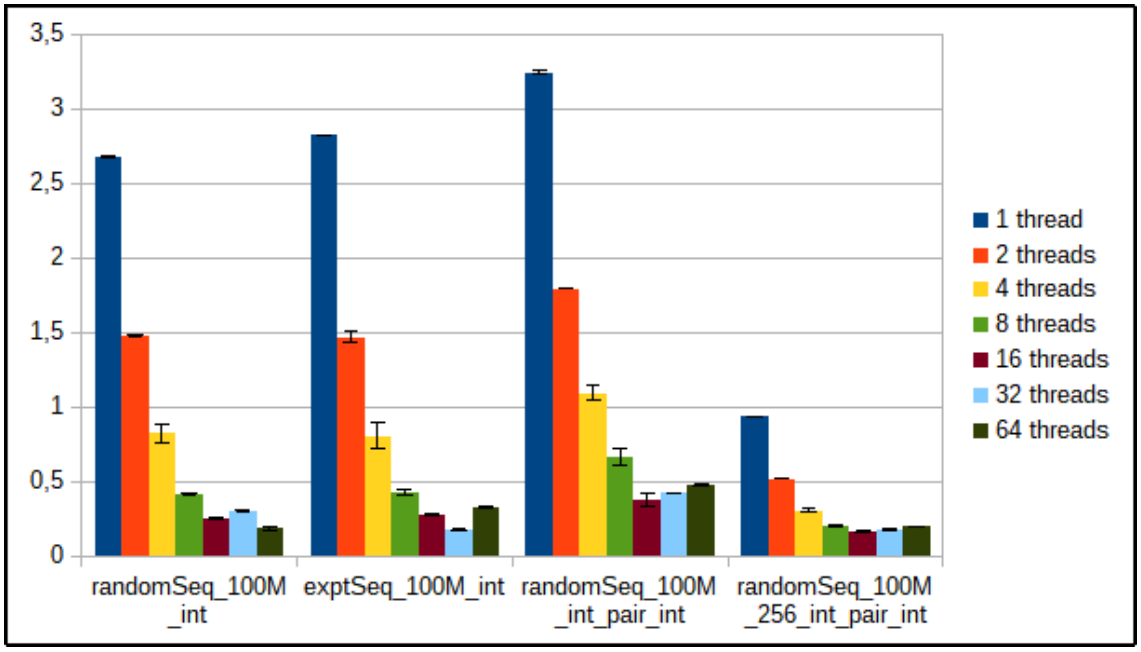


Figure A.14: Execution times (in minutes) of Integer Sort ran on AMD 32-64 using LCWS with WShare (**continous tit-for-tat**)

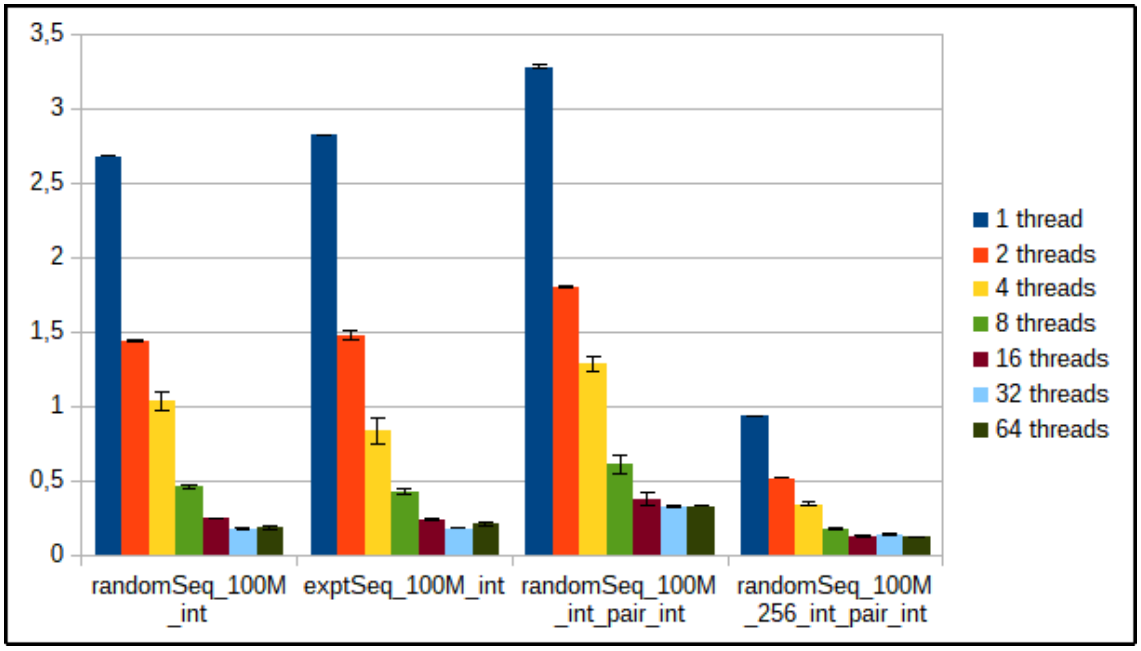


Figure A.15: Execution times (in minutes) of Integer Sort ran on AMD 32-64 using SBLCWS with WShare (**first version**)

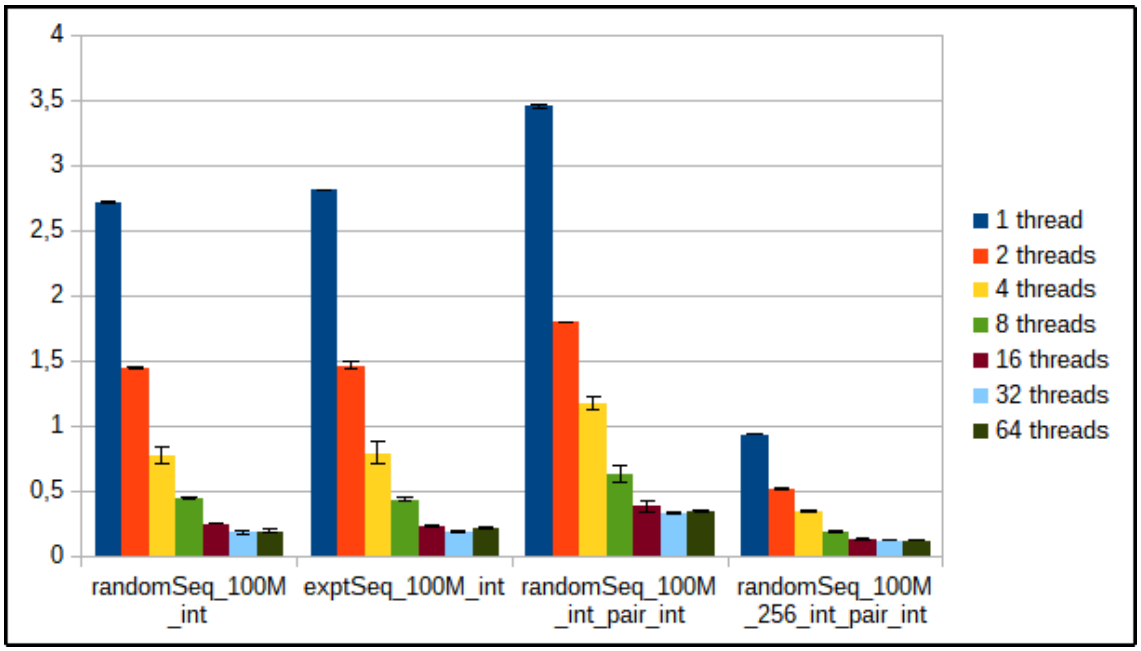


Figure A.16: Execution times (in minutes) of Integer Sort ran on AMD 32-64 using SBLCWS with WShare (**second version**)

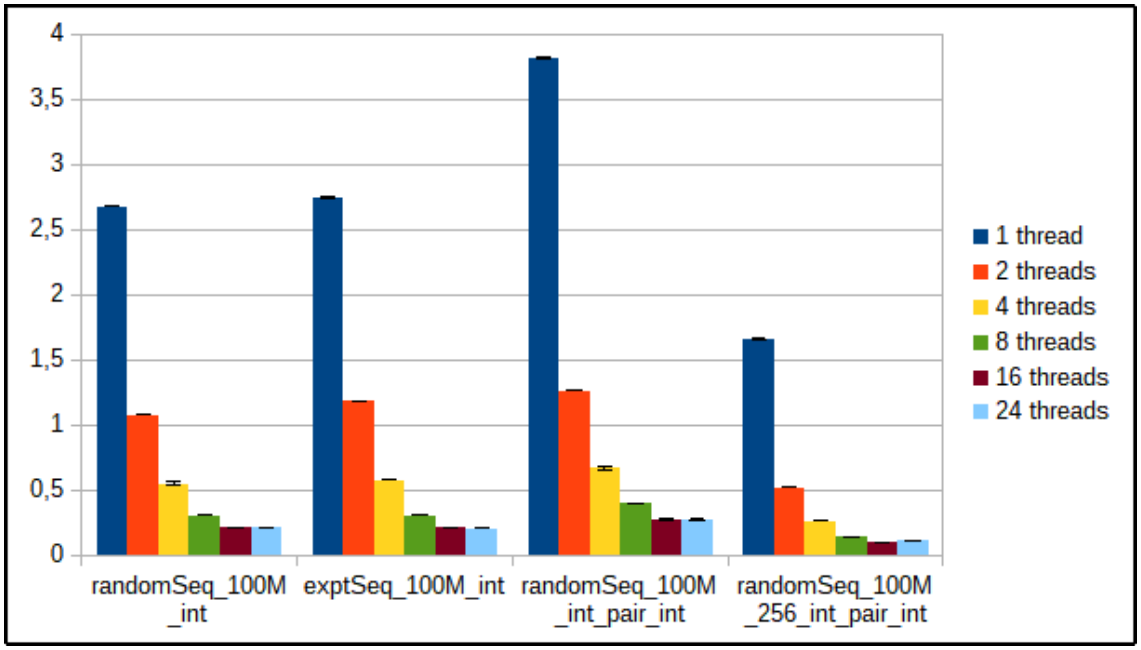


Figure A.17: Execution times (in minutes) of Integer Sort ran on Intel 16-16 using WSteal

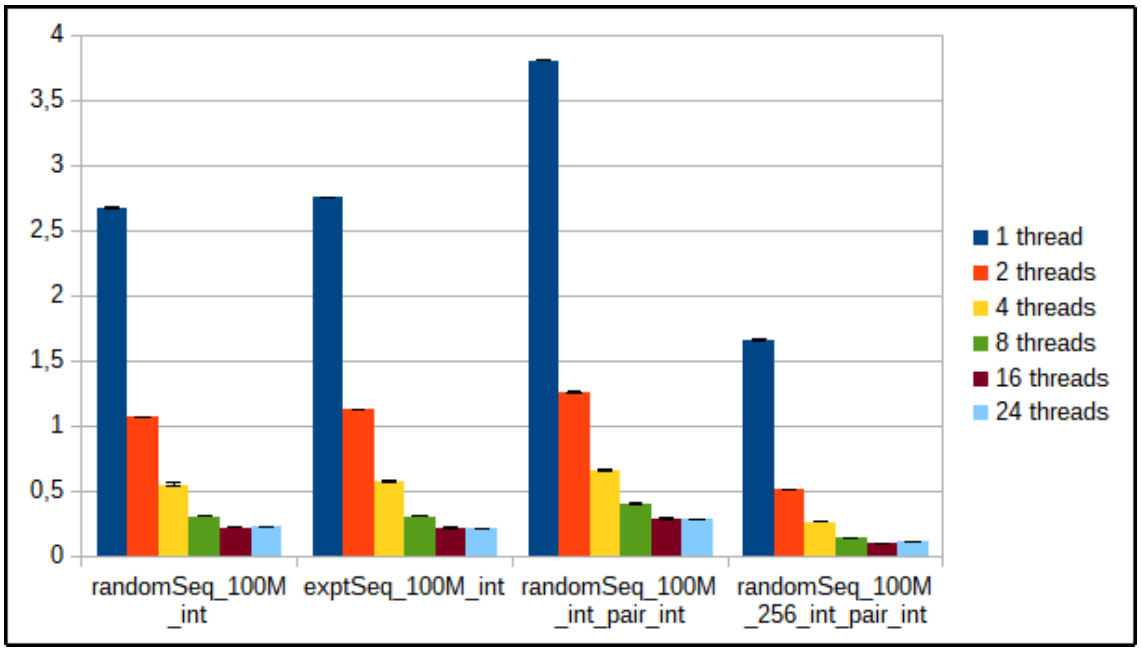


Figure A.18: Execution times (in minutes) of Integer Sort ran on Intel 16-16 using LCWS

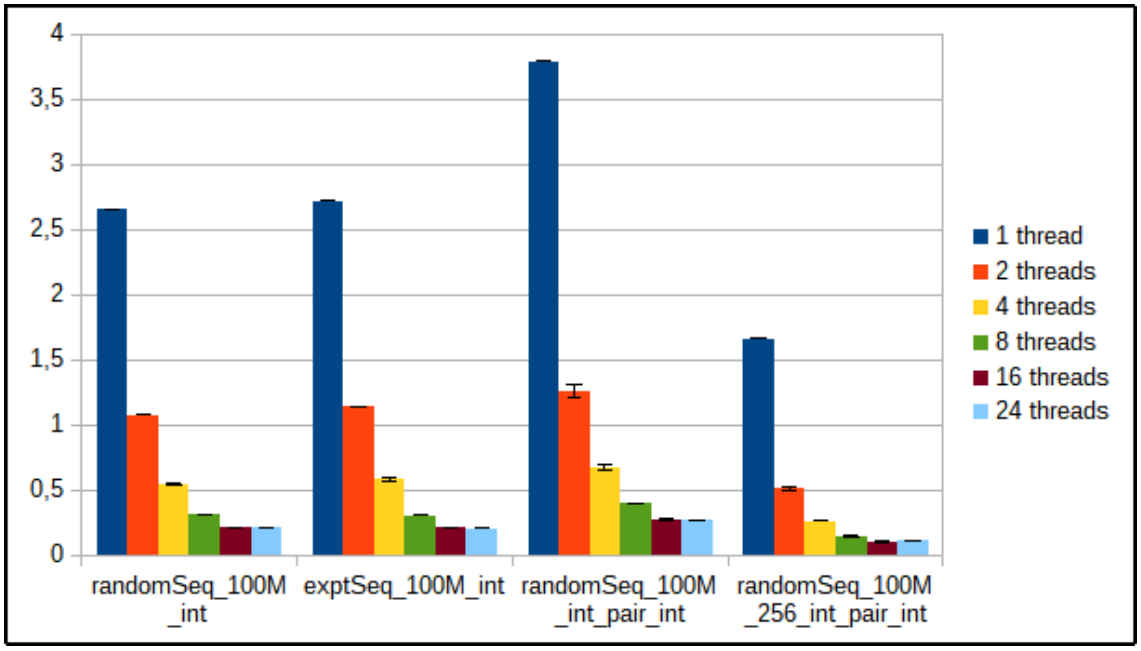


Figure A.19: Execution times (in minutes) of Integer Sort ran on Intel 16-16 using SBLCWS (**first version**)

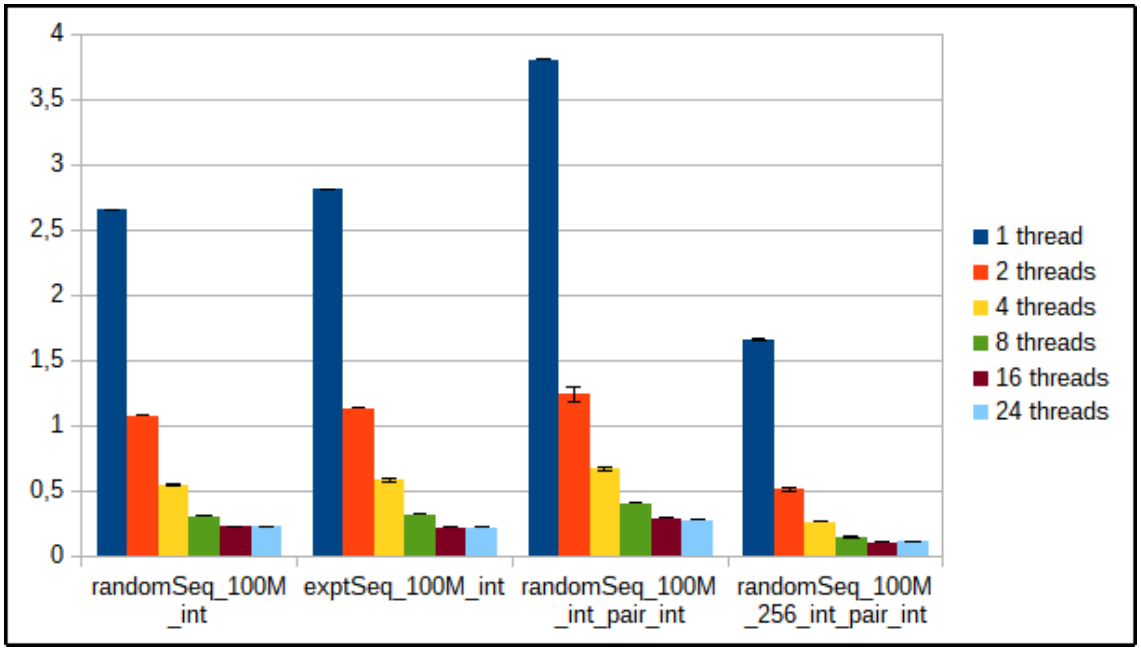


Figure A.20: Execution times (in minutes) of Integer Sort ran on Intel 16-16 using LCWS with WShare (**tit-for-tat**)

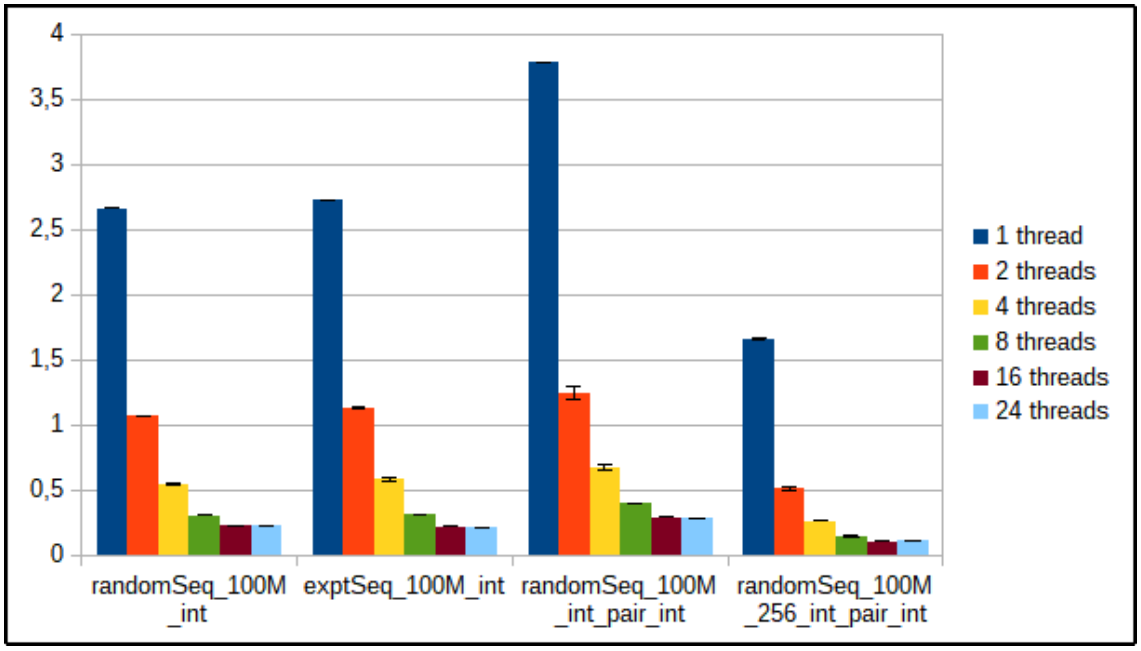


Figure A.21: Execution times (in minutes) of Integer Sort ran on Intel 16-16 using LCWS with WShare (**continuous tit-for-tat**)

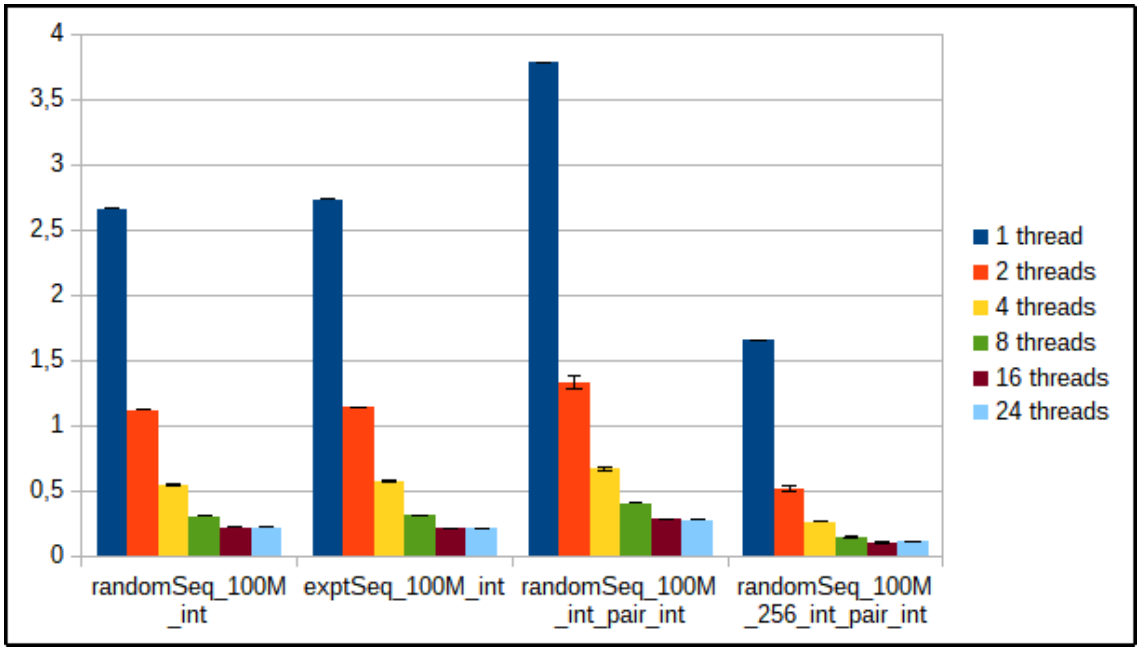


Figure A.22: Execution times (in minutes) of Integer Sort ran on Intel 16-16 using SBLCWS with WShare (**first version**)

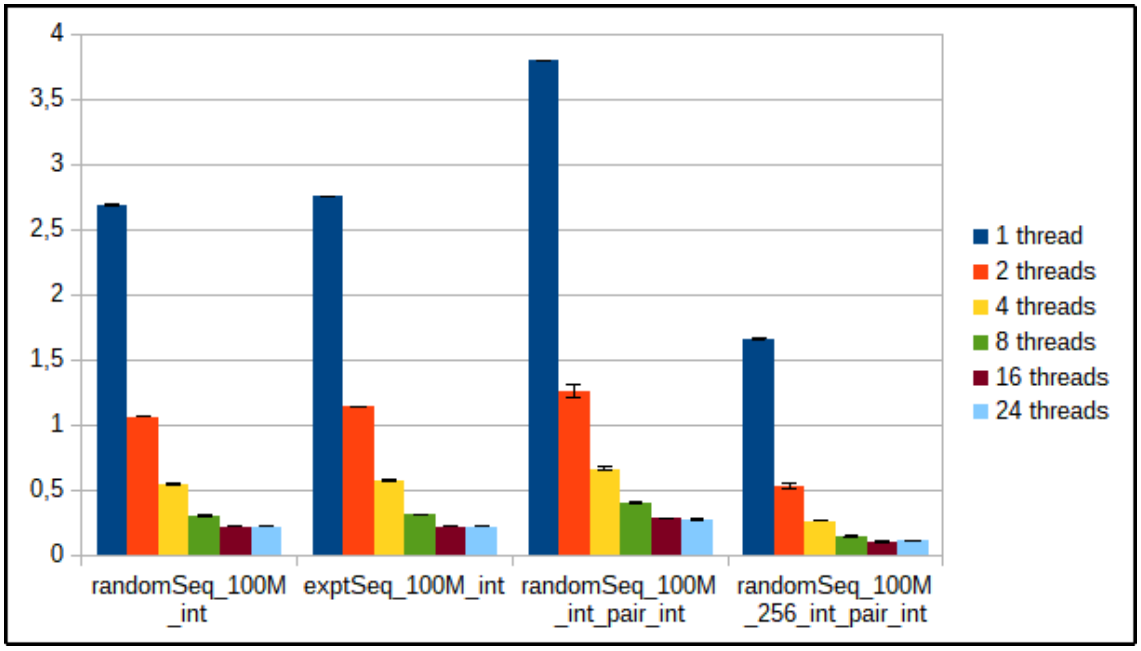


Figure A.23: Execution times (in minutes) of Integer Sort ran on Intel 16-16 using SBLCWS with WShare (**second version**)

A.2 Comparison Sort

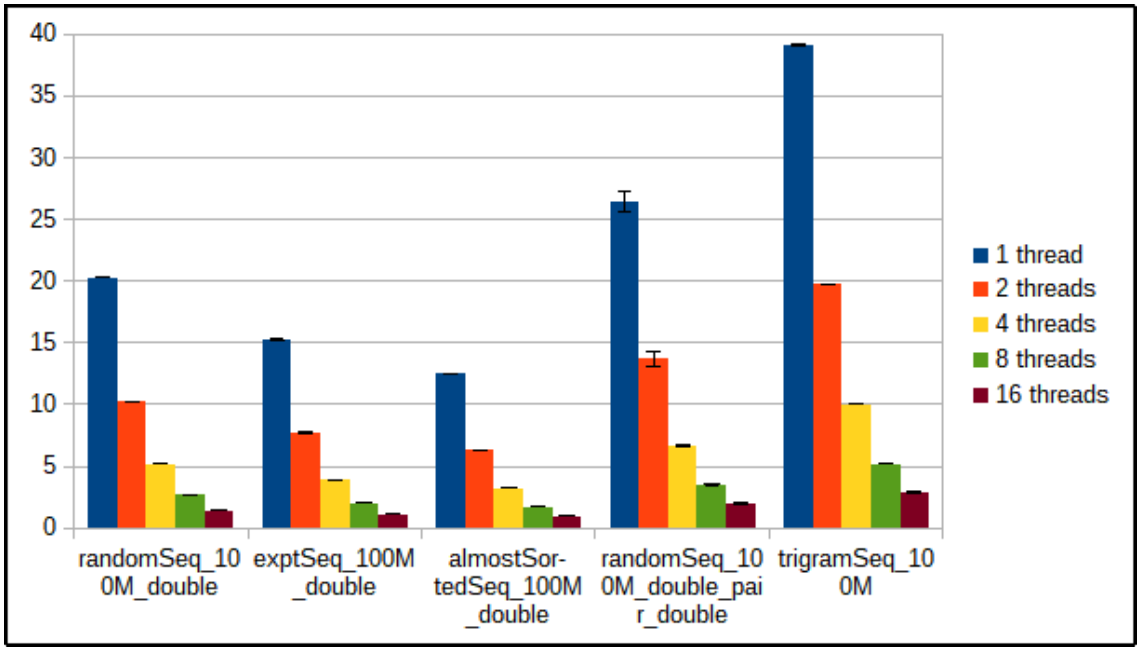


Figure A.24: Execution times (in minutes) of Comparison Sort ran on Intel 12-24 using WSteal

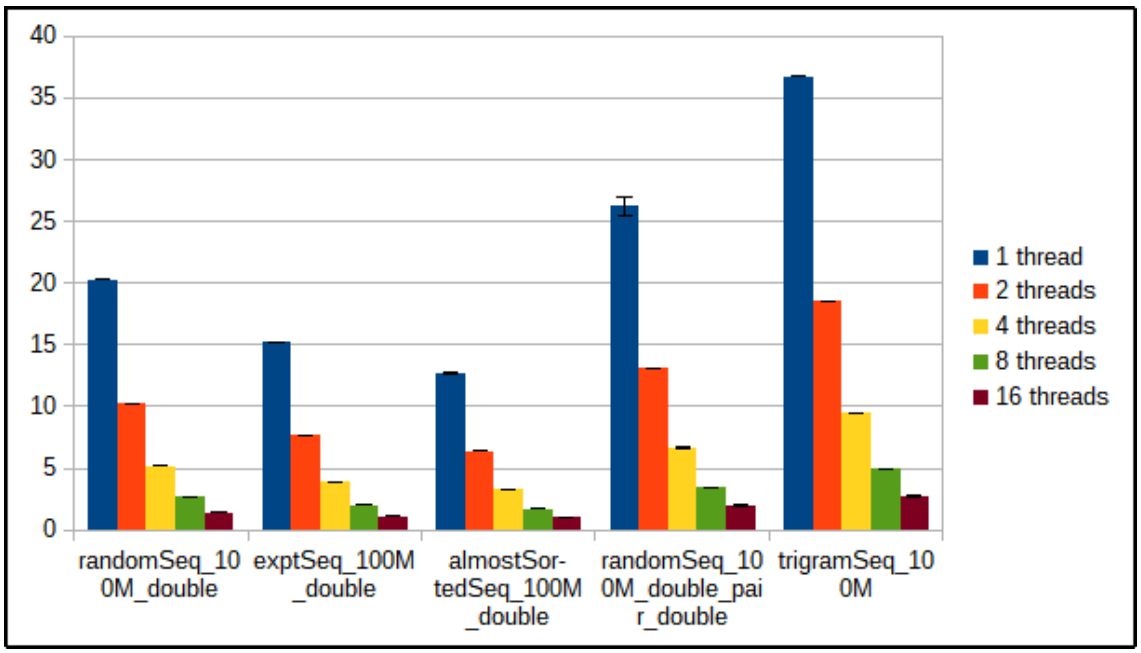


Figure A.25: Execution times (in minutes) of Comparison Sort ran on Intel 12-24 using LCWS

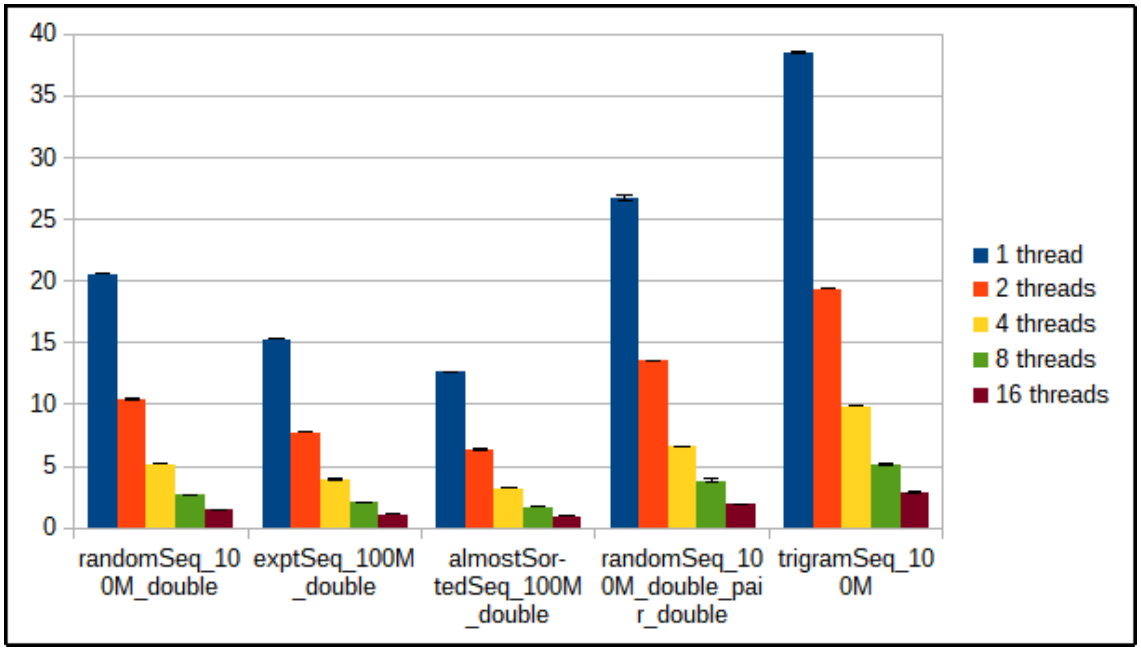


Figure A.26: Execution times (in minutes) of Comparison Sort ran on Intel 12-24 using SBLCWS (**first version**)

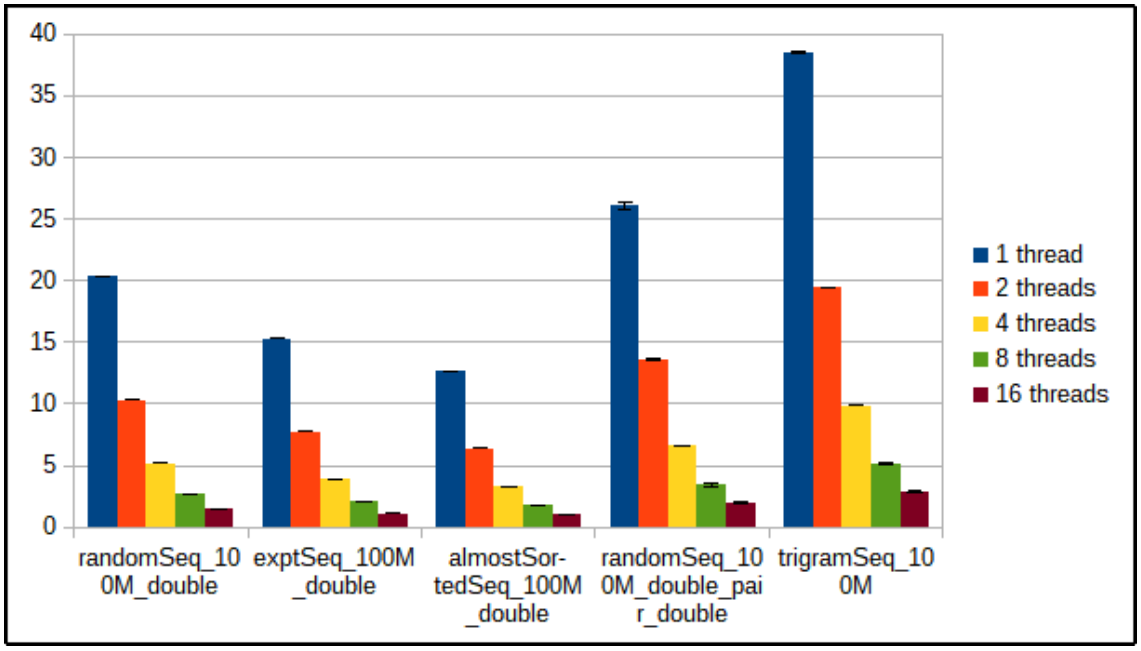


Figure A.27: Execution times (in minutes) of Comparison Sort ran on Intel 12-24 using SBLCWS (**second version**)

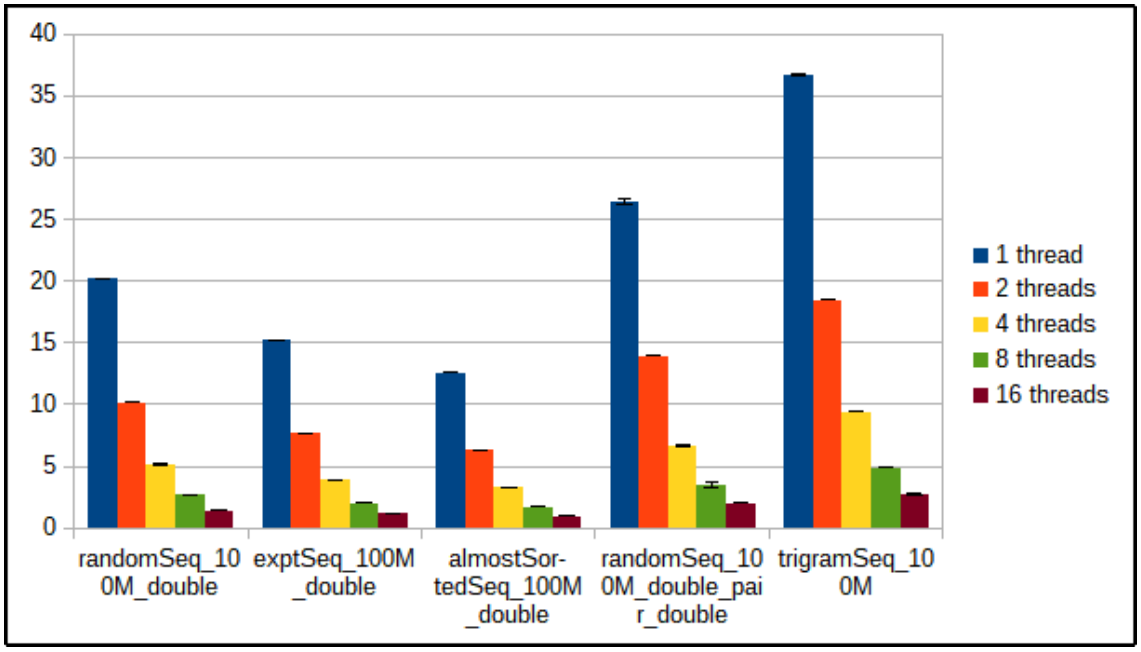


Figure A.28: Execution times (in minutes) of Comparison Sort ran on Intel 12-24 using LCWS with WShare (**tit-for-tat**)

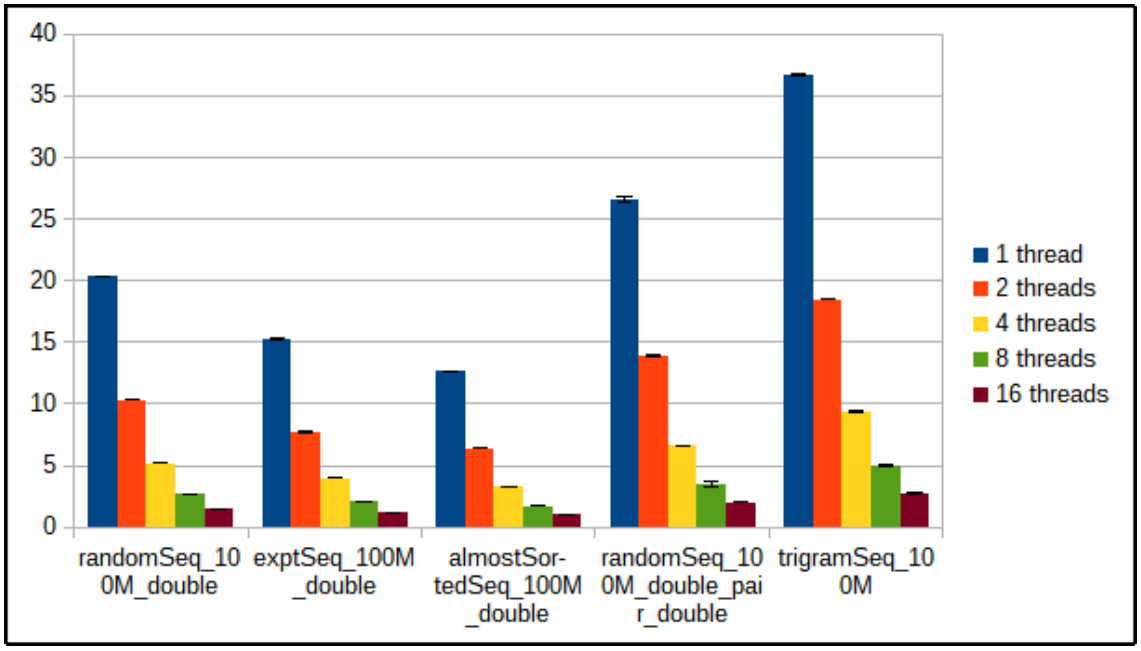


Figure A.29: Execution times (in minutes) of Comparison Sort ran on Intel 12-24 using LCWS with WShare (**continuous tit-for-tat**)

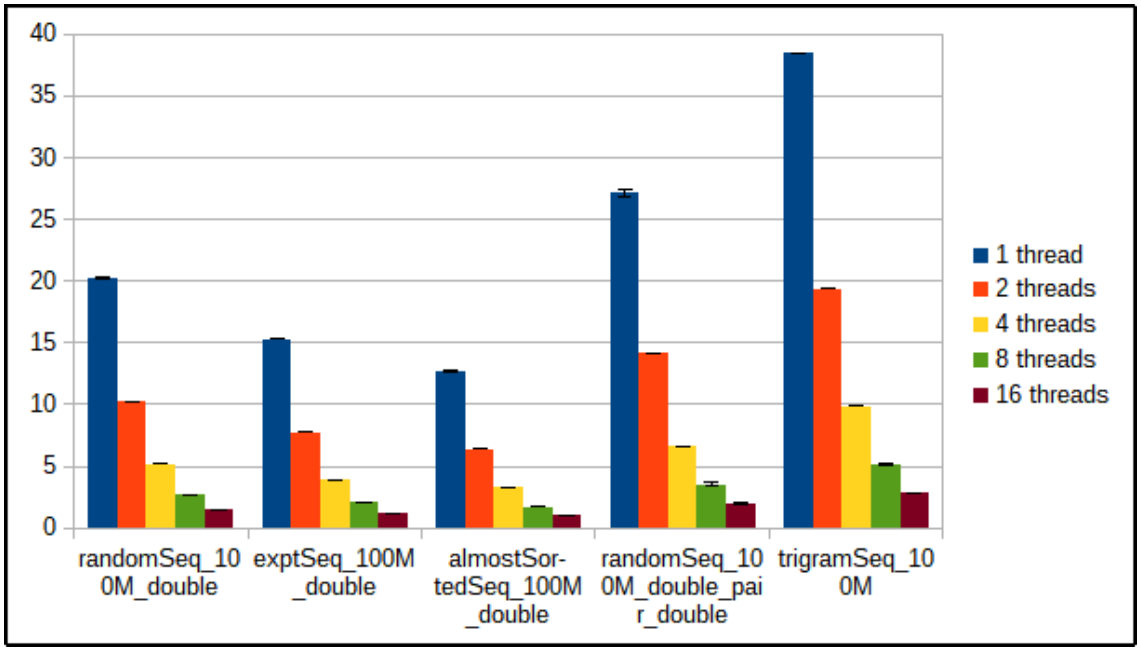


Figure A.30: Execution times (in minutes) of Comparison Sort ran on Intel 12-24 using SBLCWS with WShare (**first version**)

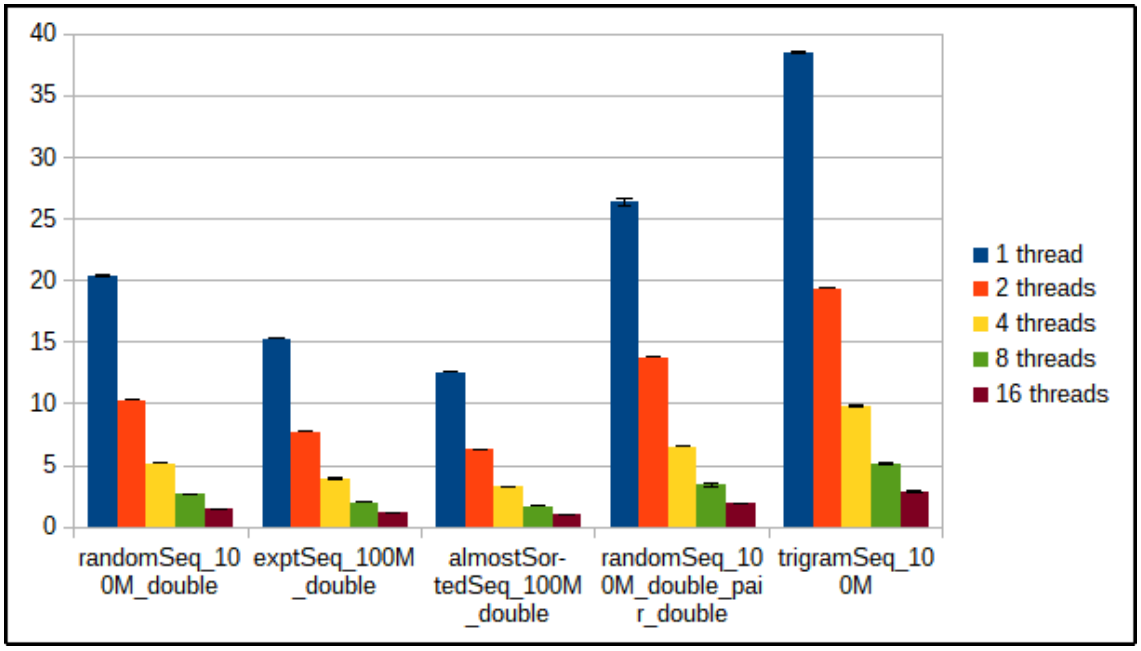


Figure A.31: Execution times (in minutes) of Comparison Sort ran on Intel 12-24 using SBLCWS with WShare (**second version**)

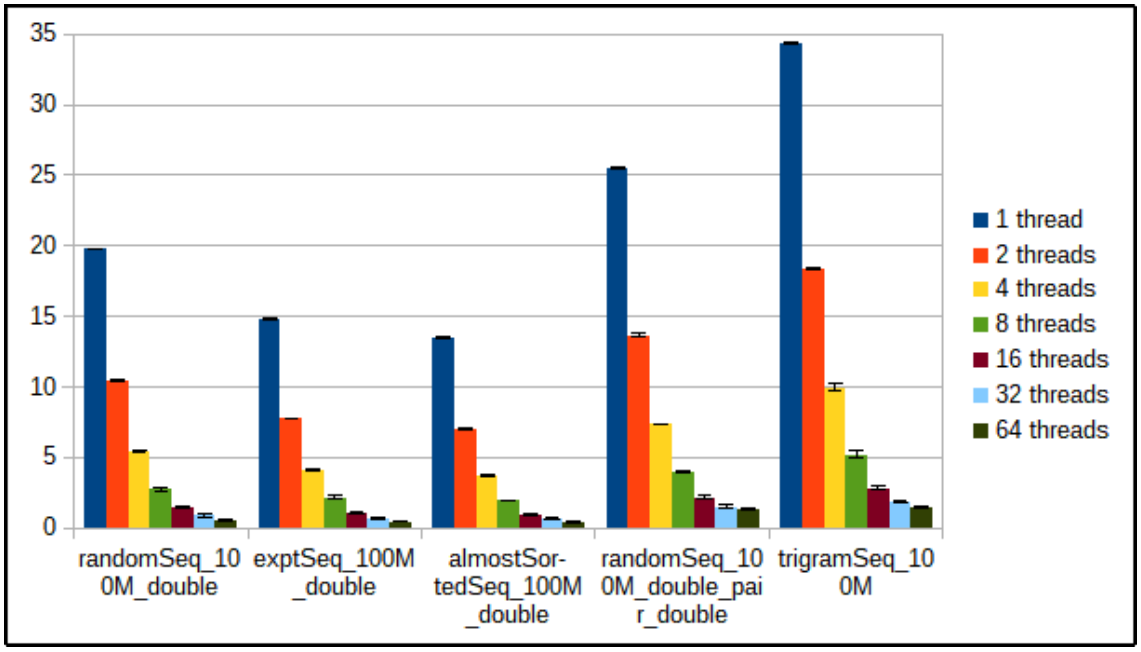


Figure A.32: Execution times (in minutes) of Comparison Sort ran on AMD 32-64 using WSteal

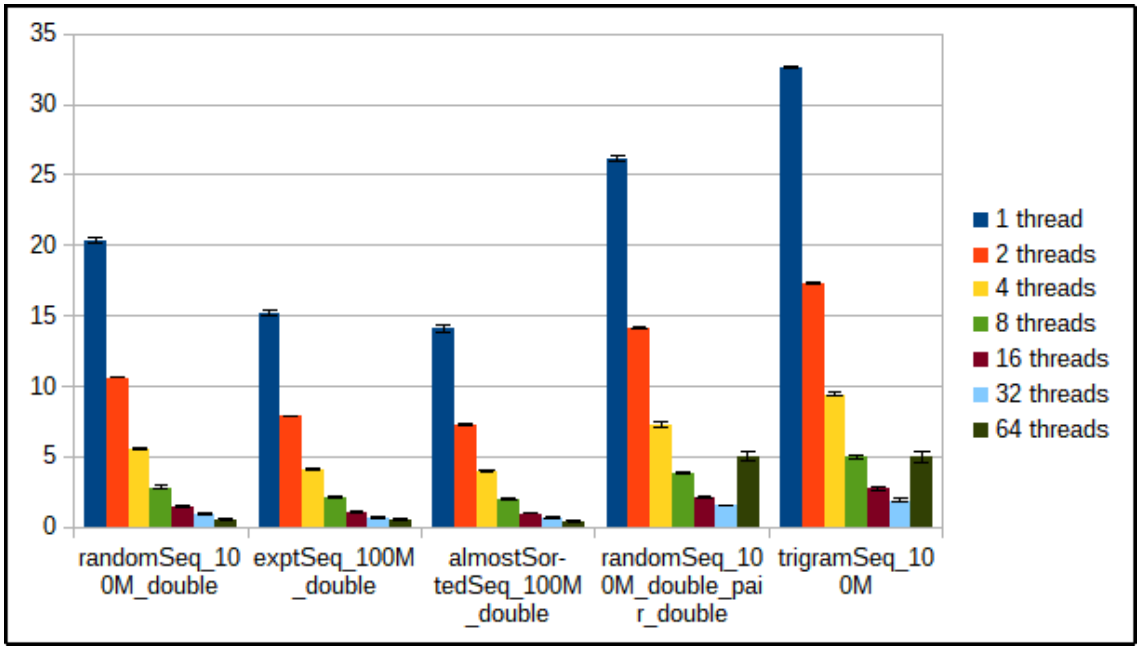


Figure A.33: Execution times (in minutes) of Comparison Sort ran on AMD 32-64 using LCWS

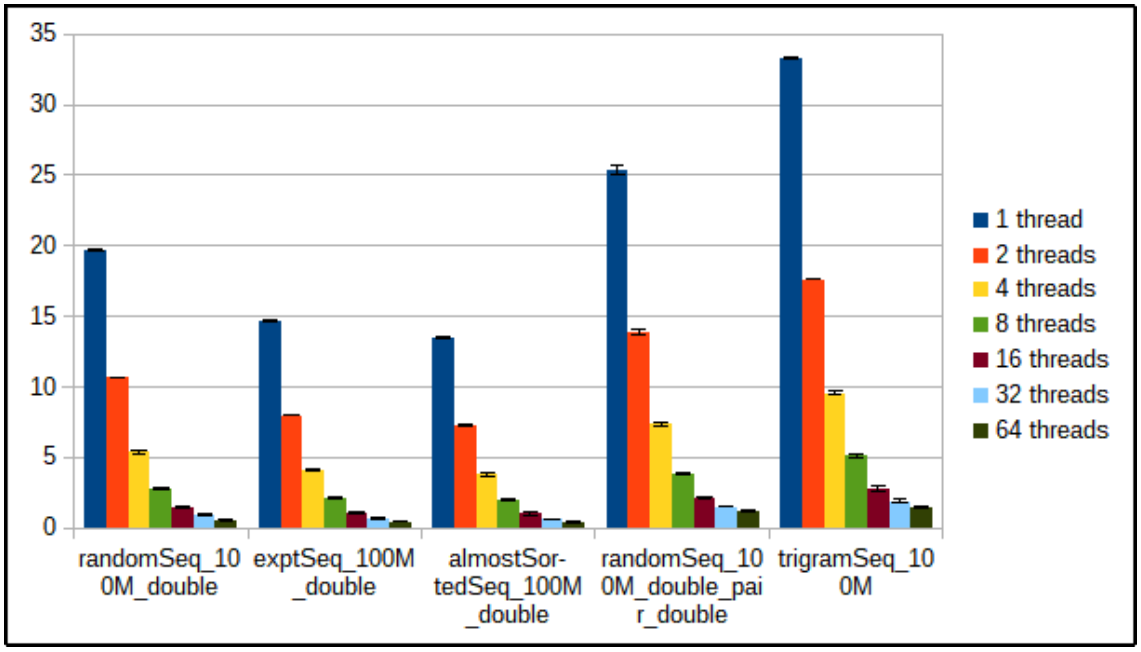


Figure A.34: Execution times (in minutes) of Comparison Sort ran on AMD 32-64 using SBLCWS (**first version**)

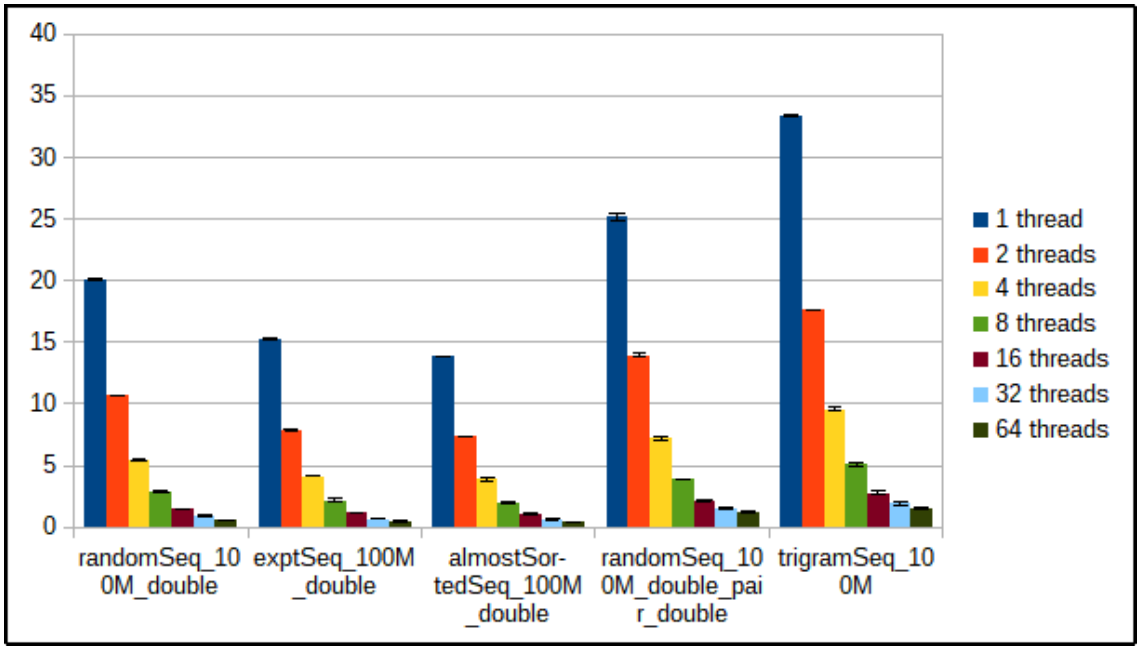


Figure A.35: Execution times (in minutes) of Comparison Sort ran on AMD 32-64 using SBLCWS (**second version**)

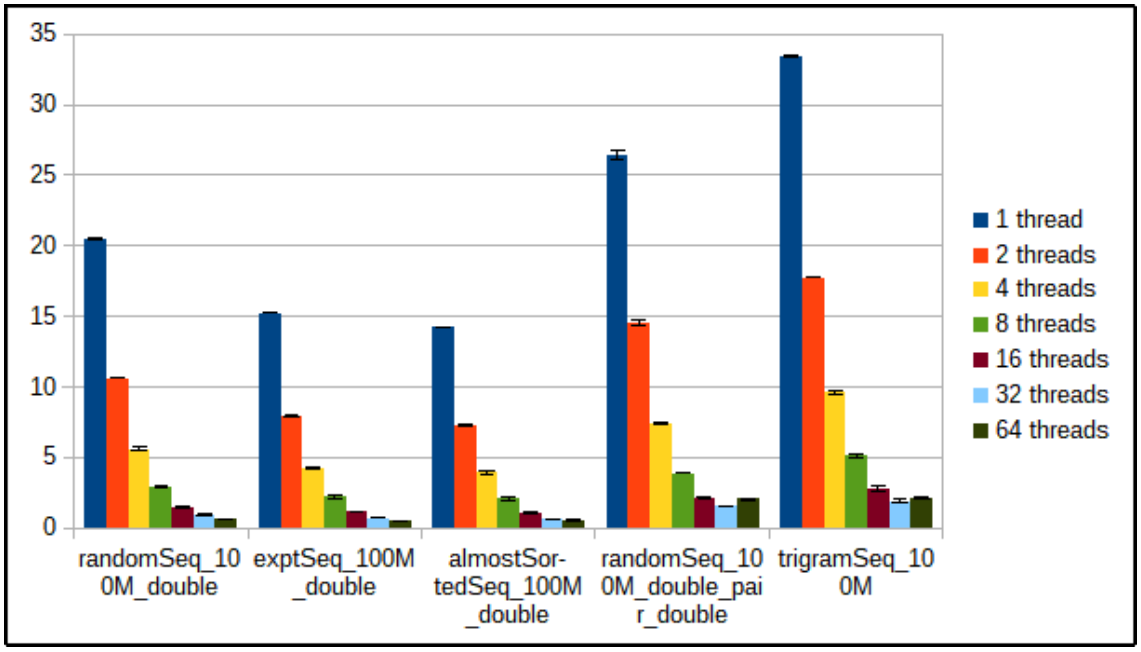


Figure A.36: Execution times (in minutes) of Comparison Sort ran on AMD 32-64 using LCWS with WShare (**tit-for-tat**)

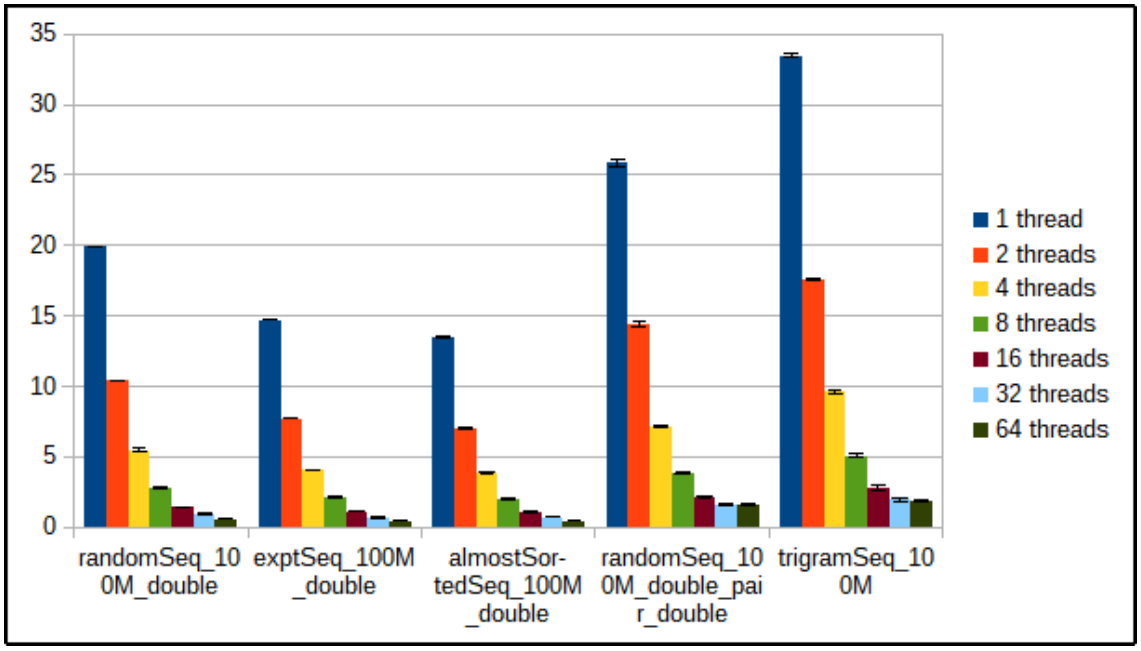


Figure A.37: Execution times (in minutes) of Comparison Sort ran on AMD 32-64 using LCWS with WShare (**continous tit-for-tat**)

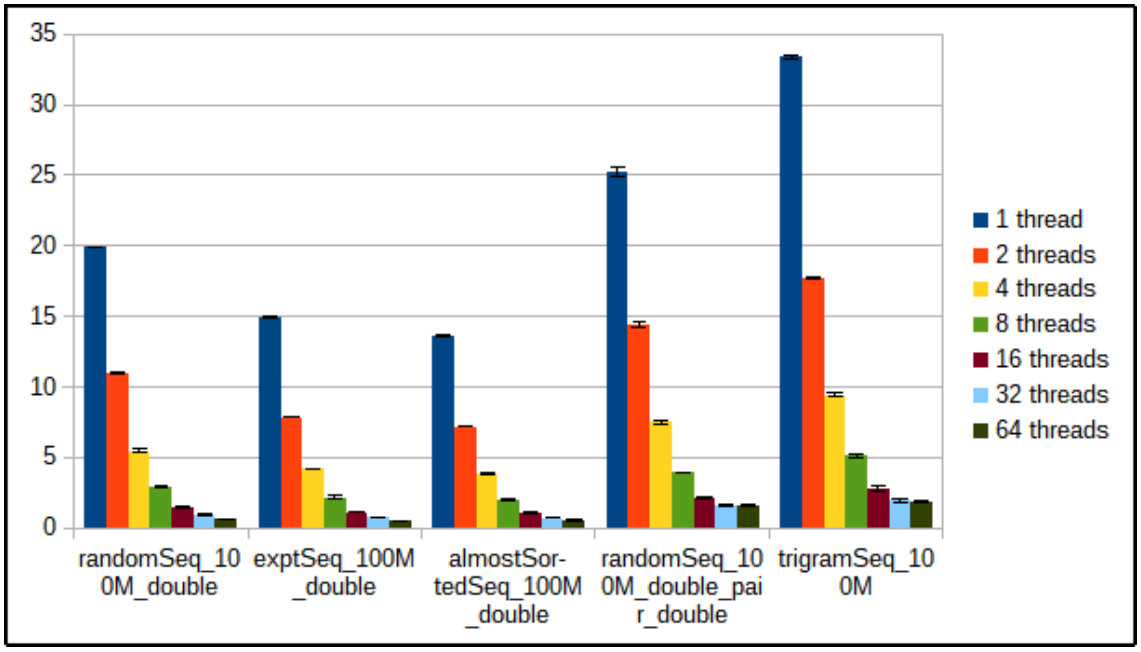


Figure A.38: Execution times (in minutes) of Comparison Sort ran on AMD 32-64 using SBLCWS with WShare (**first version**)

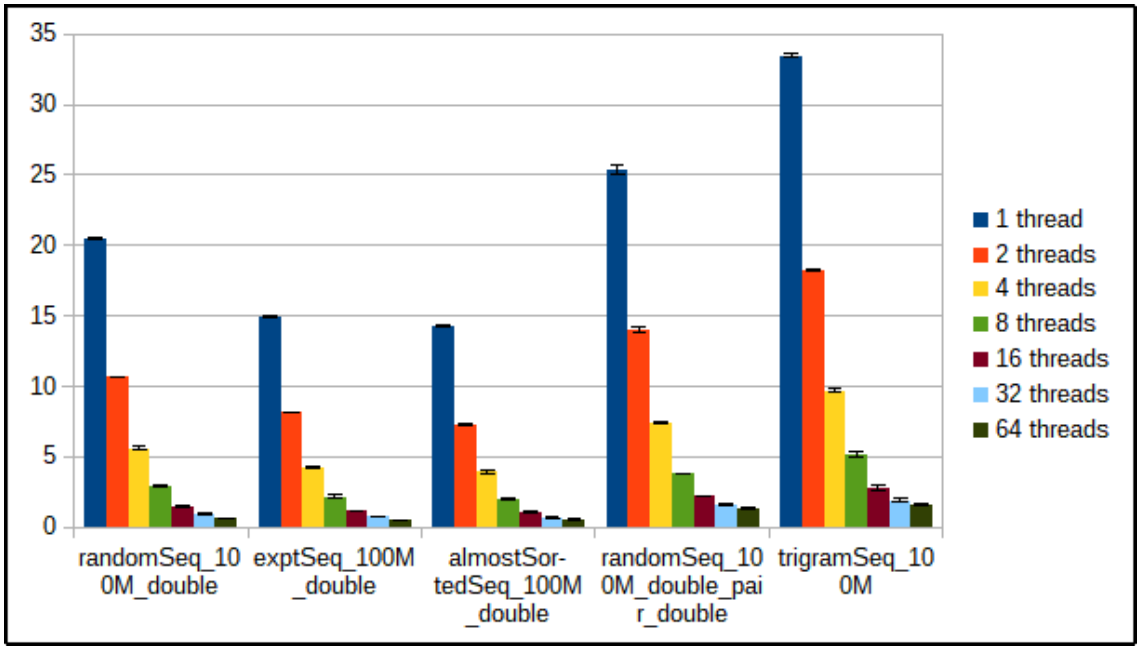


Figure A.39: Execution times (in minutes) of Comparison Sort ran on AMD 32-64 using SBLCWS with WShare (**second version**)

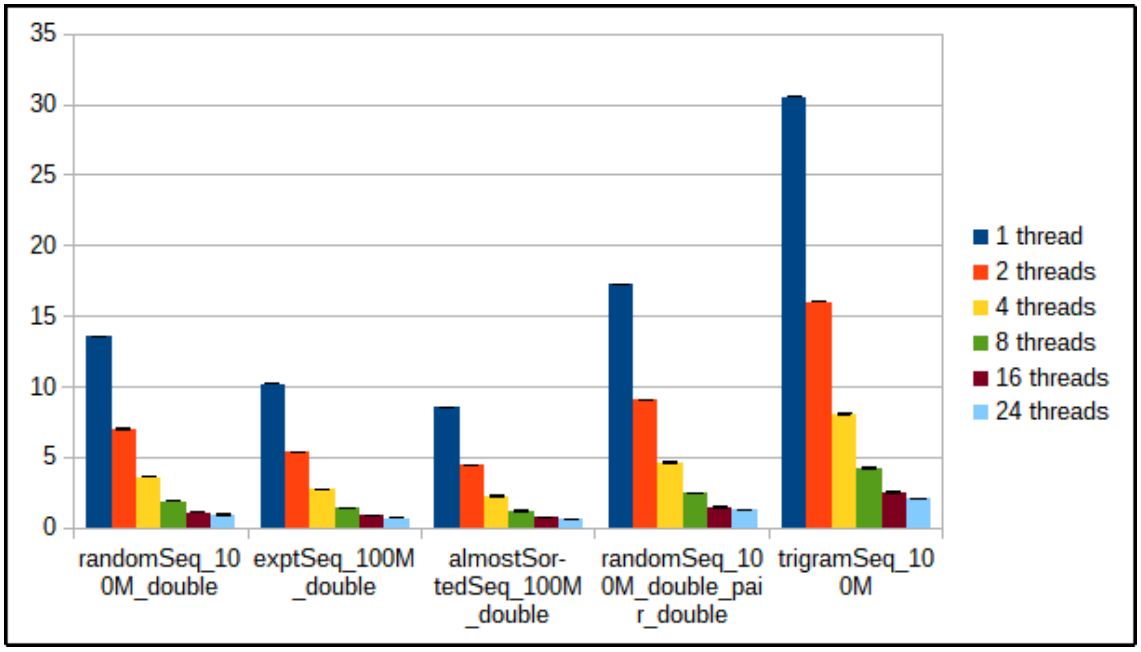


Figure A.40: Execution times (in minutes) of Comparison Sort ran on Intel 16-16 using WSteal

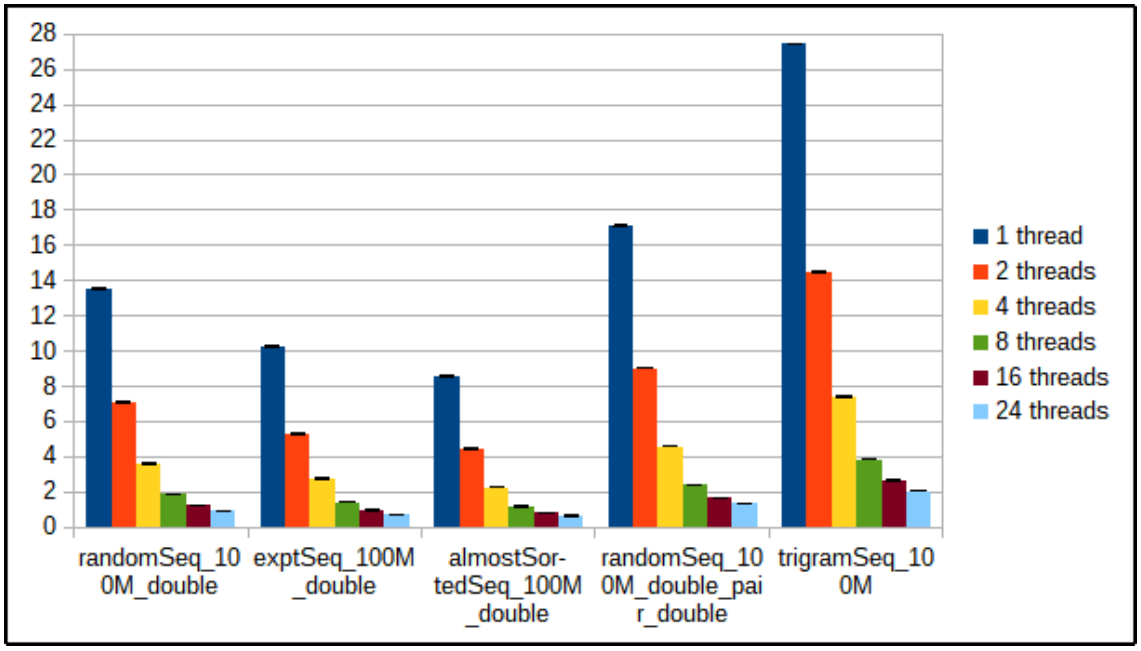


Figure A.41: Execution times (in minutes) of Comparison Sort ran on Intel 16-16 using LCWS

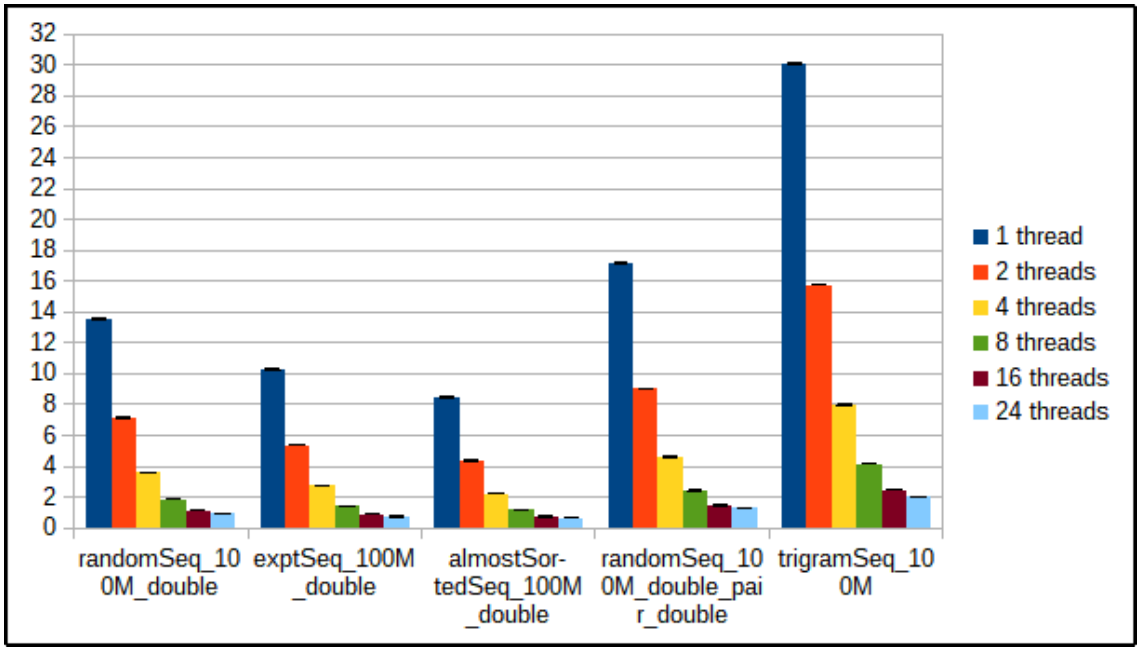


Figure A.42: Execution times (in minutes) of Comparison Sort ran on Intel 16-16 using SBLCWS (**first version**)

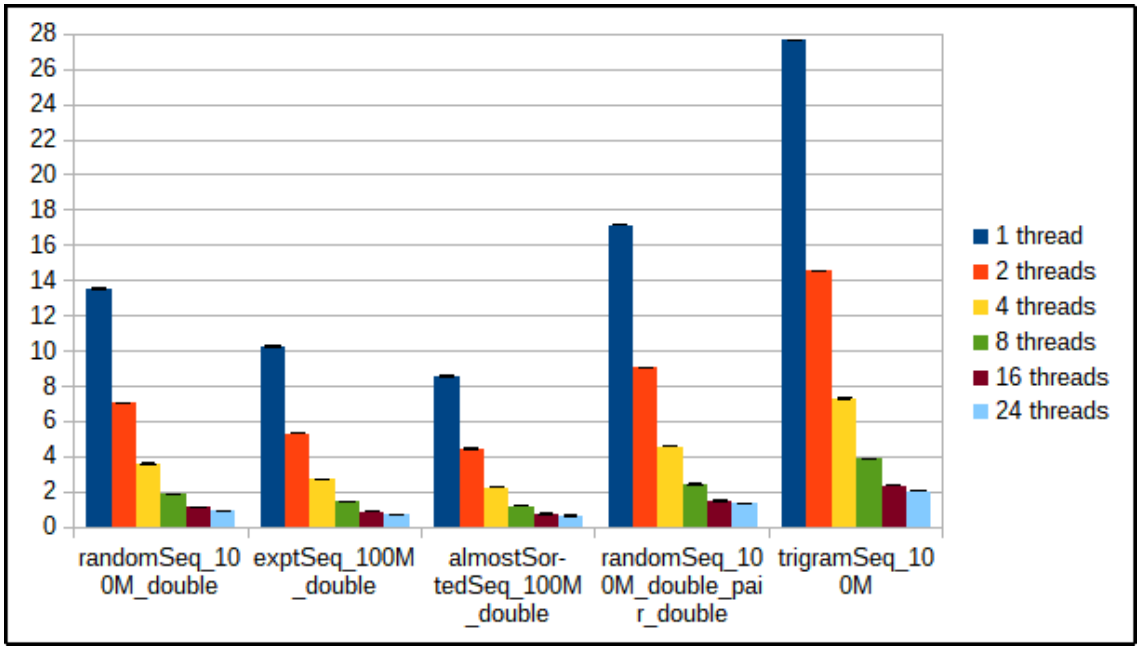


Figure A.43: Execution times (in minutes) of Comparison Sort ran on Intel 16-16 using LCWS with WShare (**tit-for-tat**)

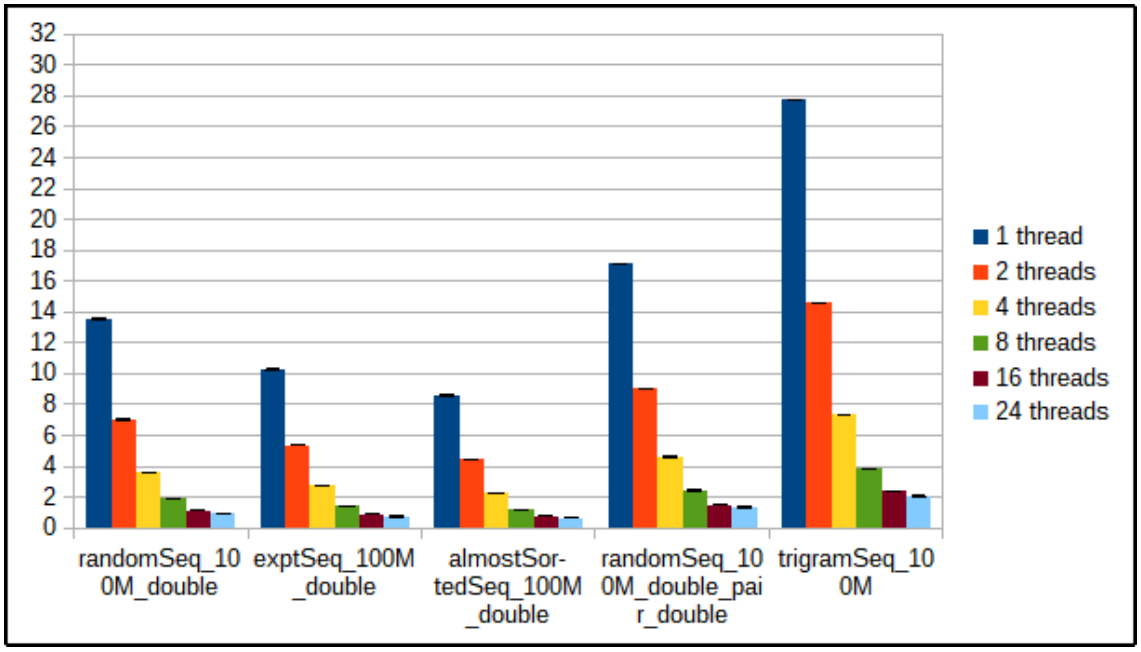


Figure A.44: Execution times (in minutes) of Comparison Sort ran on Intel 16-16 using LCWS with WShare (**continuous tit-for-tat**)

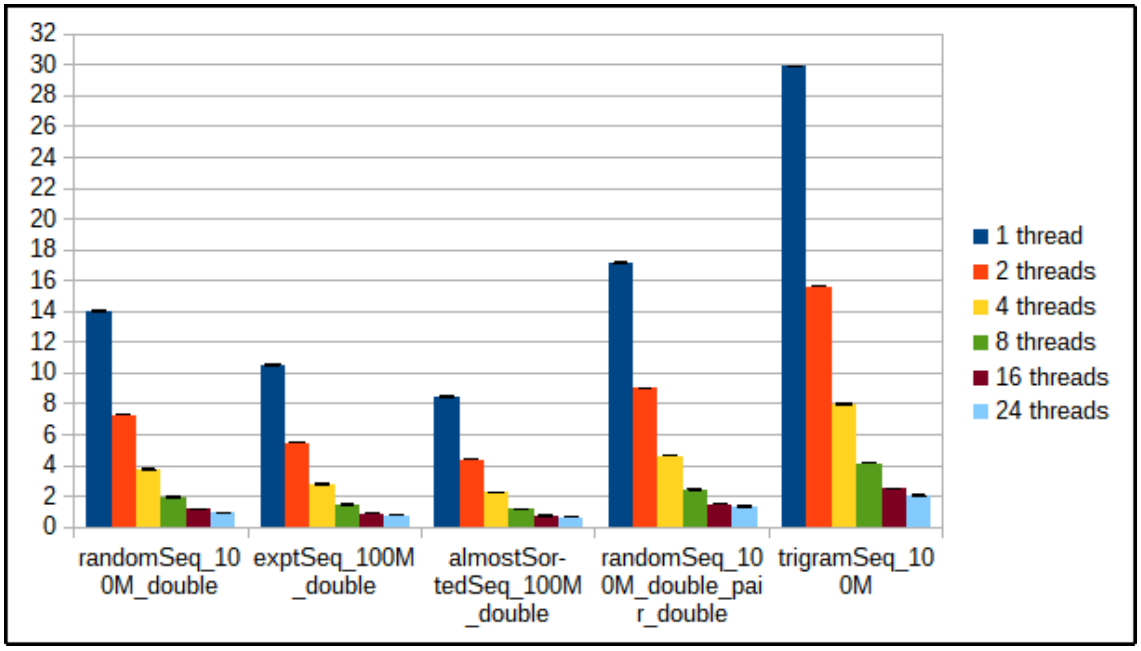


Figure A.45: Execution times (in minutes) of Comparison Sort ran on Intel 16-16 using SBLCWS with WShare (**first version**)

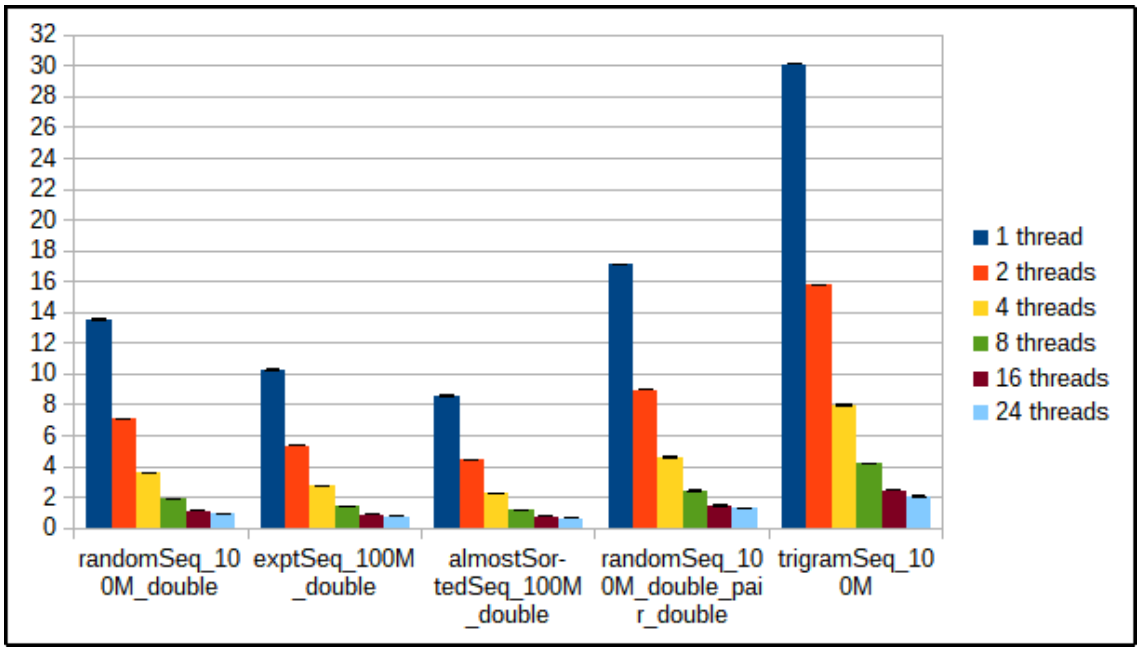


Figure A.46: Execution times (in minutes) of Comparison Sort ran on Intel 16-16 using SBLCWS with WShare (**second version**)

A.3 Remove Duplicates

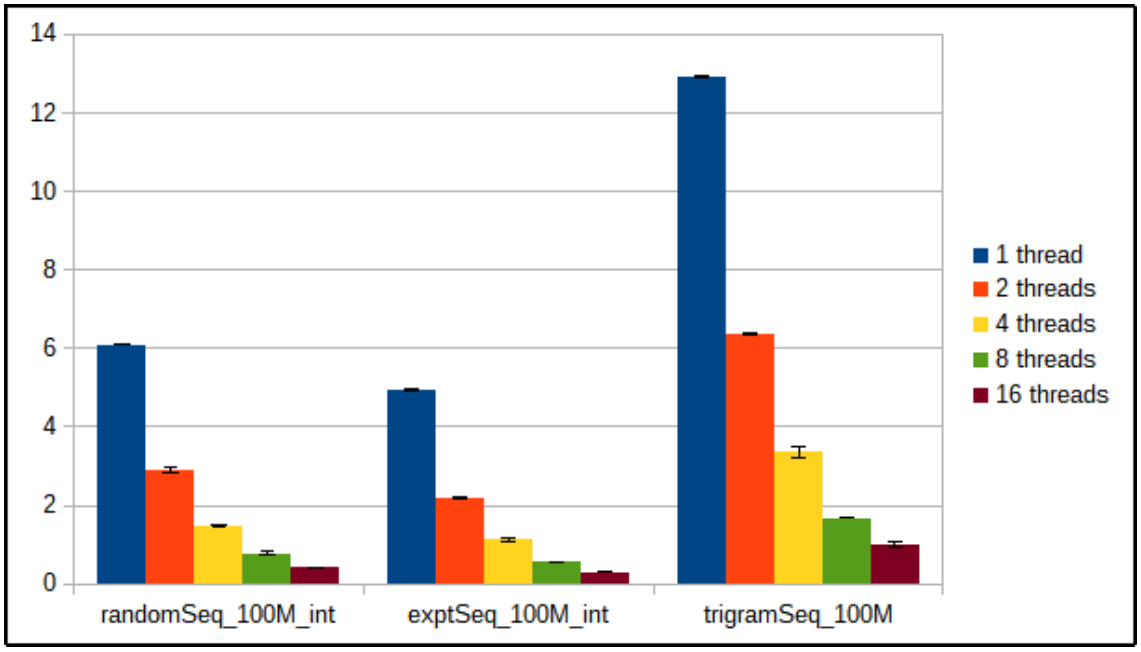


Figure A.47: Execution times (in minutes) of Remove Duplicates ran on Intel 12-24 using WSteal

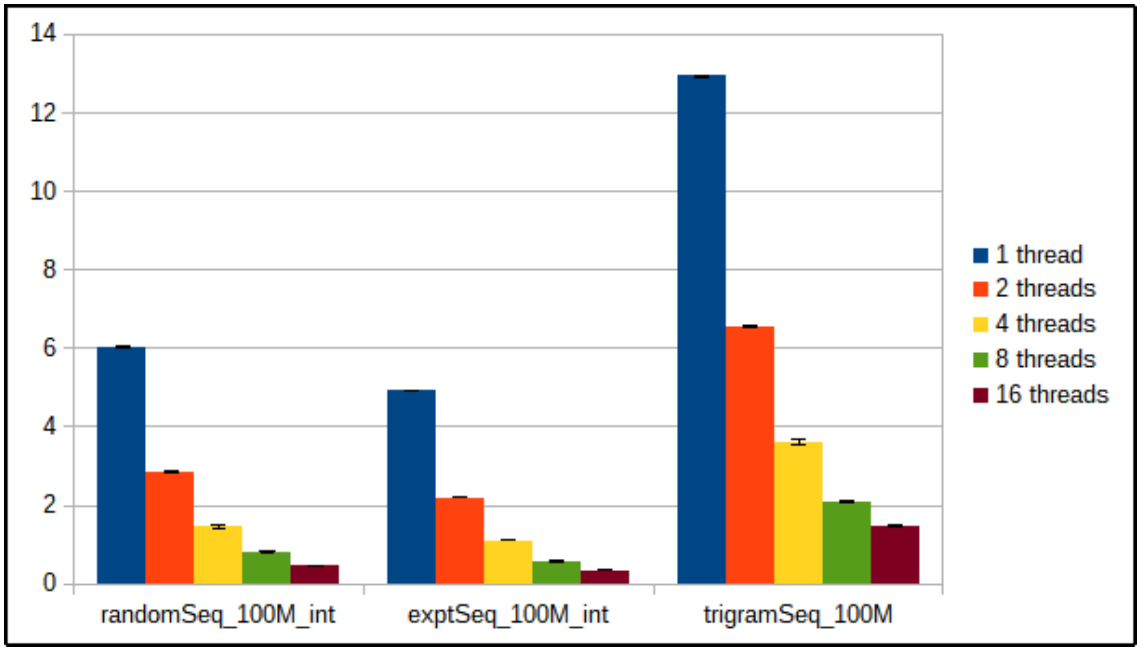


Figure A.48: Execution times (in minutes) of Remove Duplicates ran on Intel 12-24 using LCWS

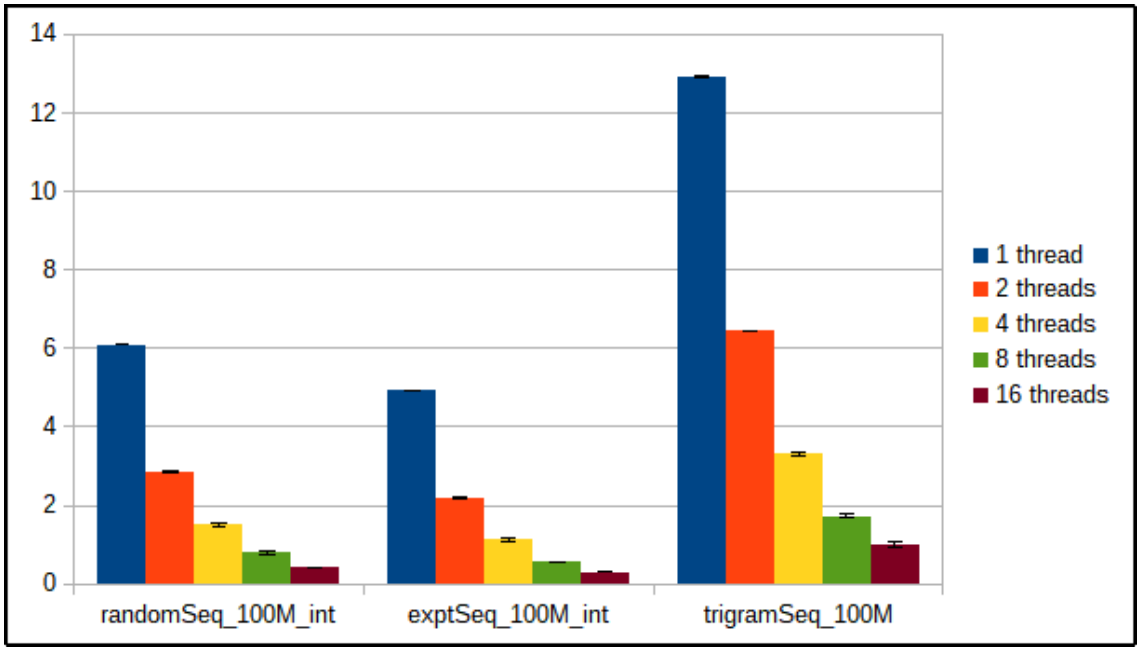


Figure A.49: Execution times (in minutes) of Remove Duplicates ran on Intel 12-24 using SBLCWS (**first version**)

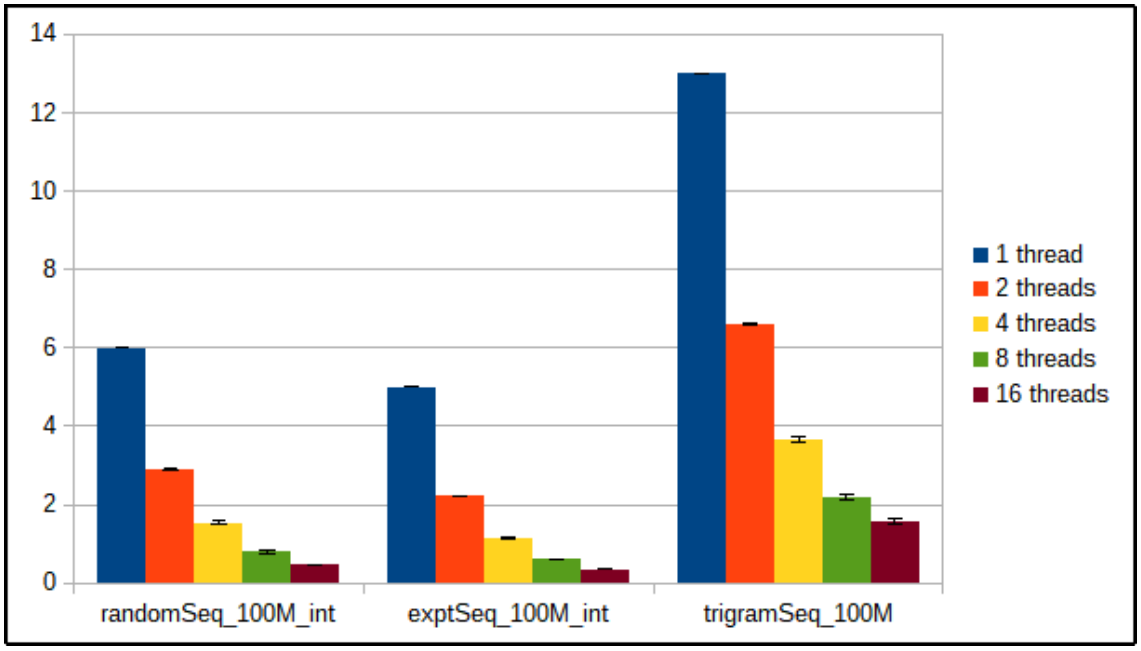


Figure A.50: Execution times (in minutes) of Remove Duplicates ran on Intel 12-24 using LCWS with WShare (**tit-for-tat**)

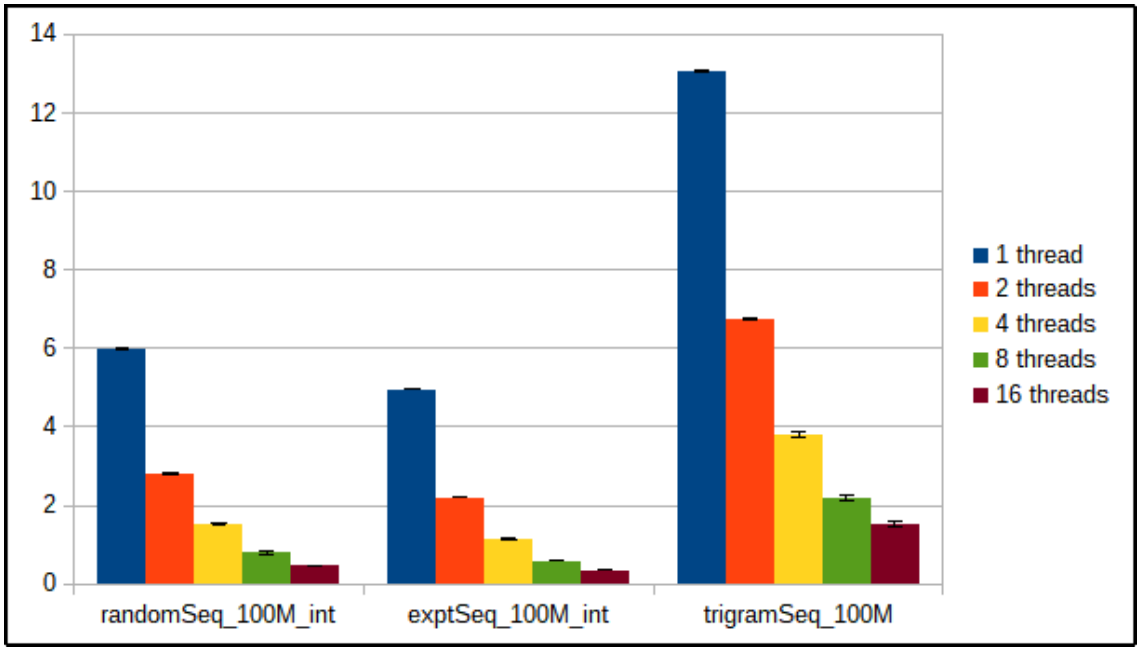


Figure A.51: Execution times (in minutes) of Remove Duplicates ran on Intel 12-24 using LCWS with WShare (**continuous tit-for-tat**)

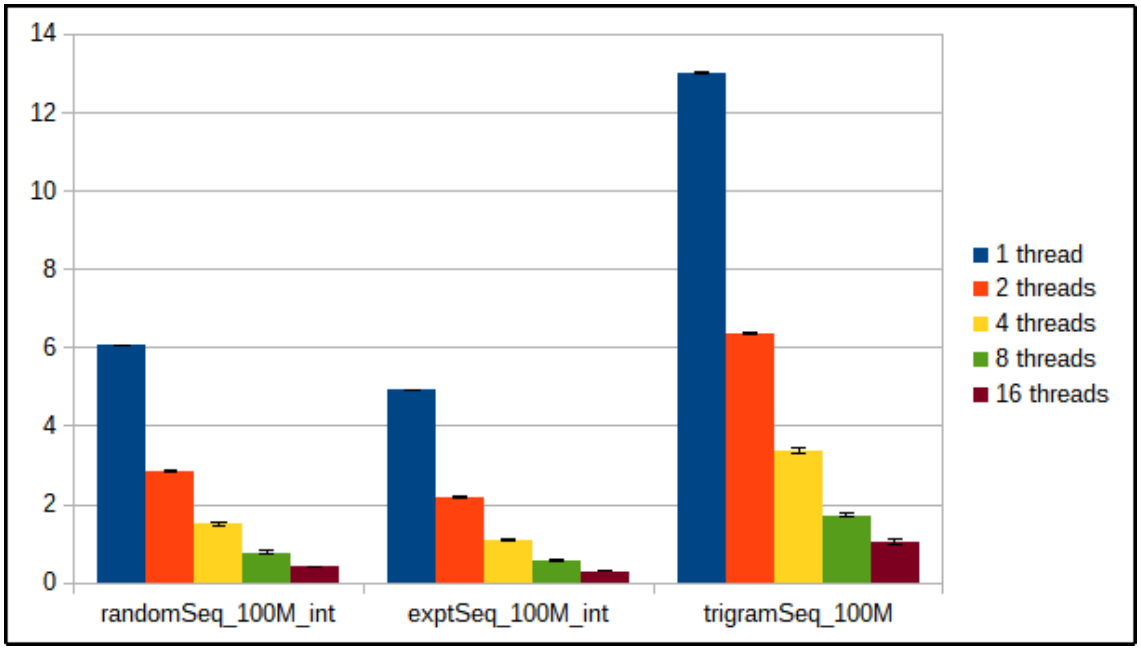


Figure A.52: Execution times (in minutes) of Remove Duplicates ran on Intel 12-24 using SBLCWS with WShare (**first version**)

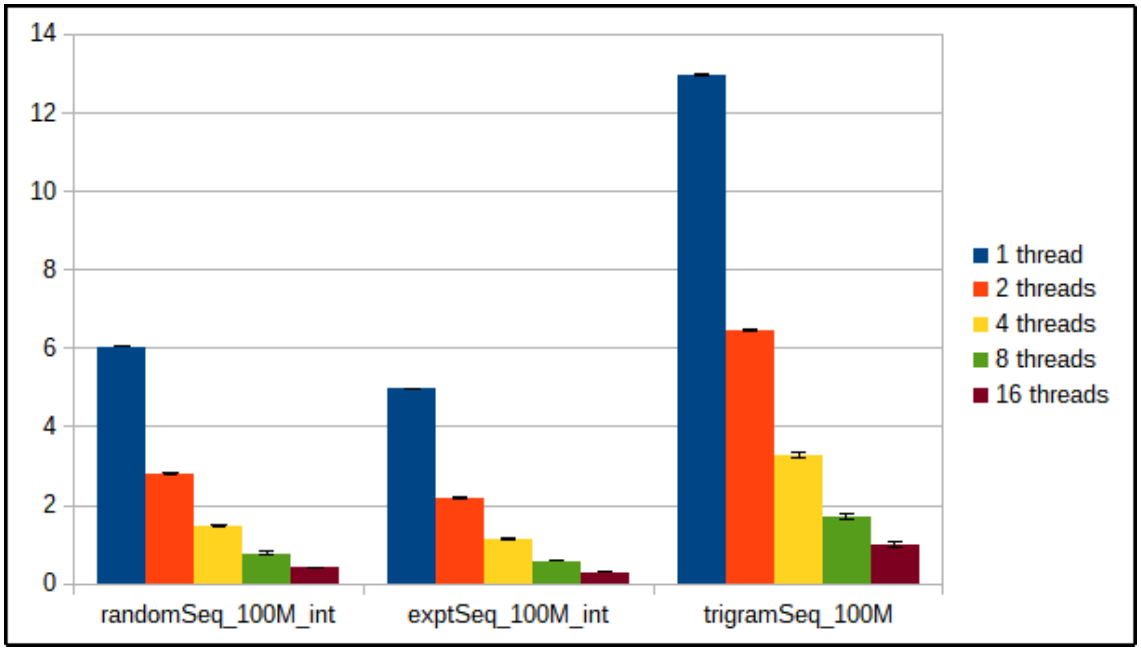


Figure A.53: Execution times (in minutes) of Remove Duplicates ran on Intel 12-24 using SBLCWS with WShare (**second version**)

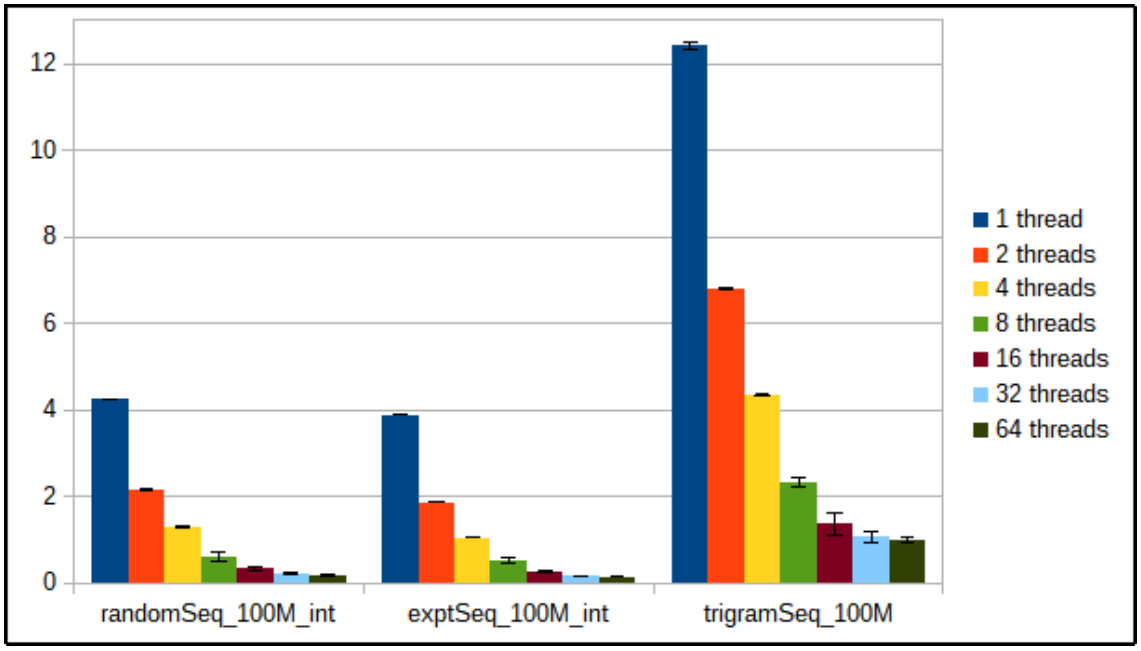


Figure A.54: Execution times (in minutes) of Remove Duplicates ran on AMD 32-64 using WSteal

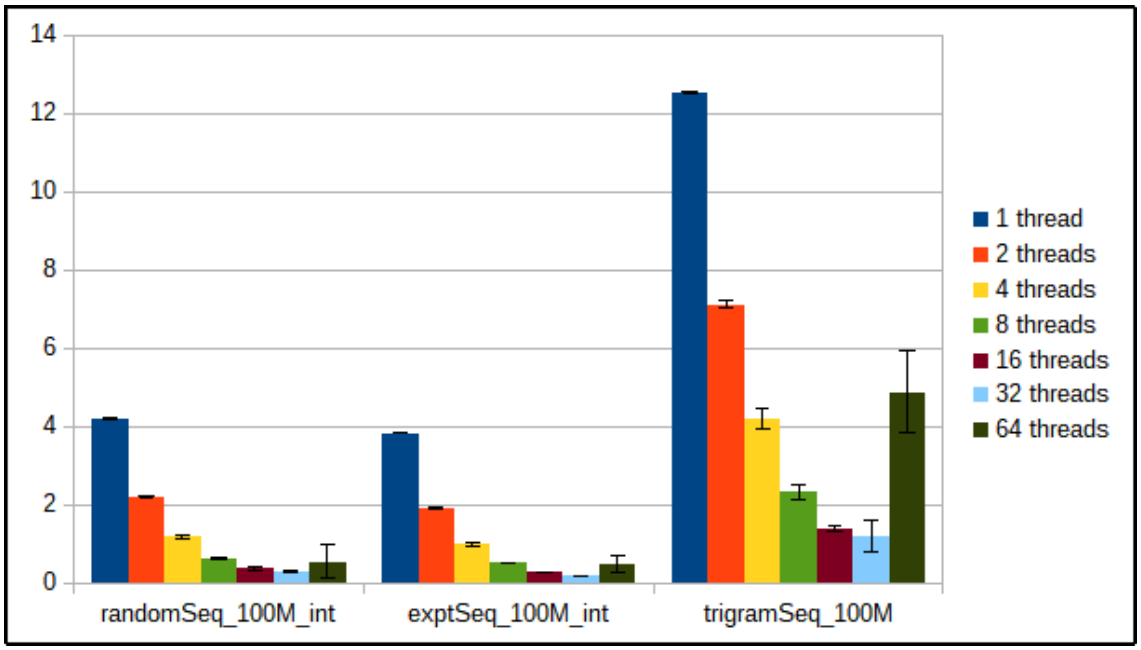


Figure A.55: Execution times (in minutes) of Remove Duplicates ran on AMD 32-64 using LCWS

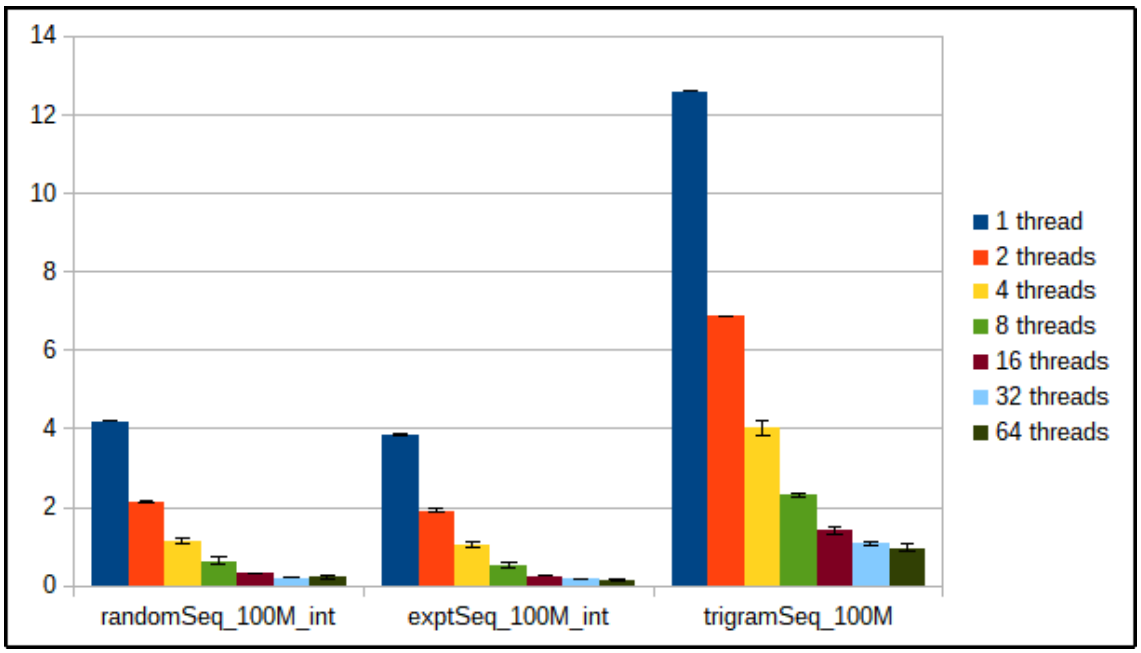


Figure A.56: Execution times (in minutes) of Remove Duplicates ran on AMD 32-64 using SBLCWS (**first version**)

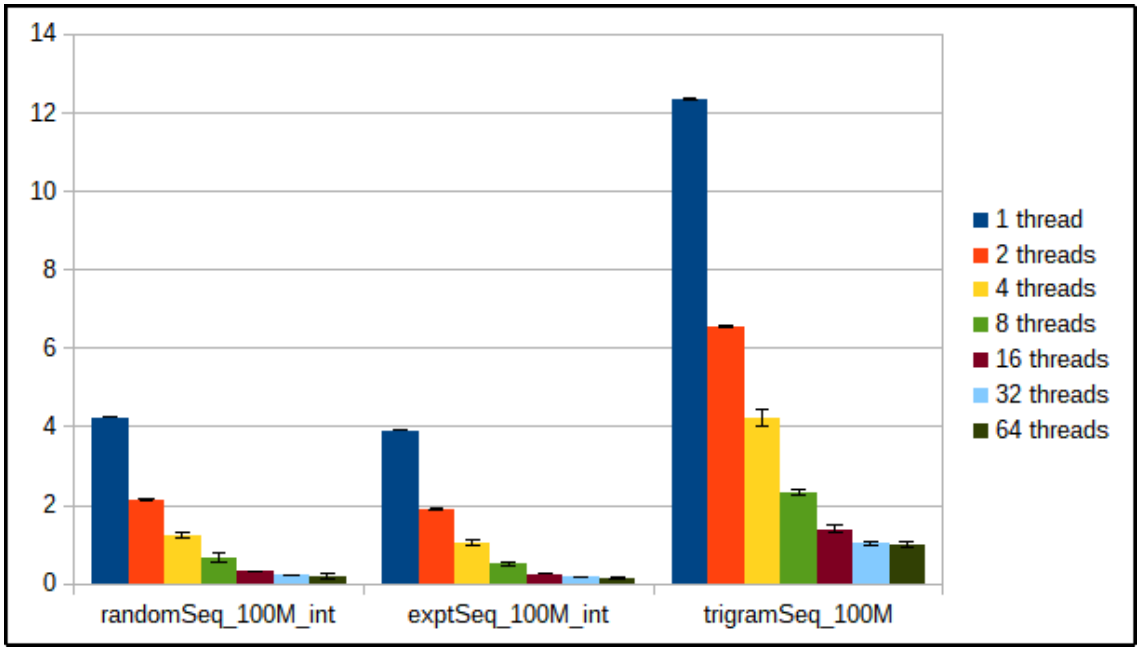


Figure A.57: Execution times (in minutes) of Remove Duplicates ran on AMD 32-64 using SBLCWS (**second version**)

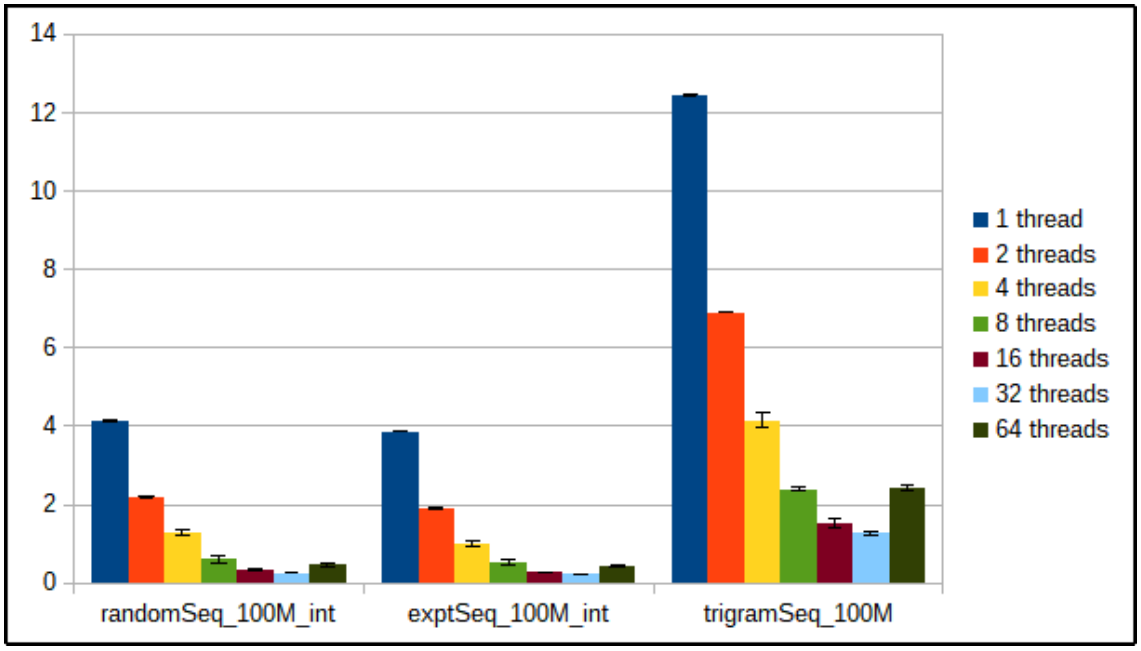


Figure A.58: Execution times (in minutes) of Remove Duplicates ran on AMD 32-64 using LCWS with WShare (**tit-for-tat**)

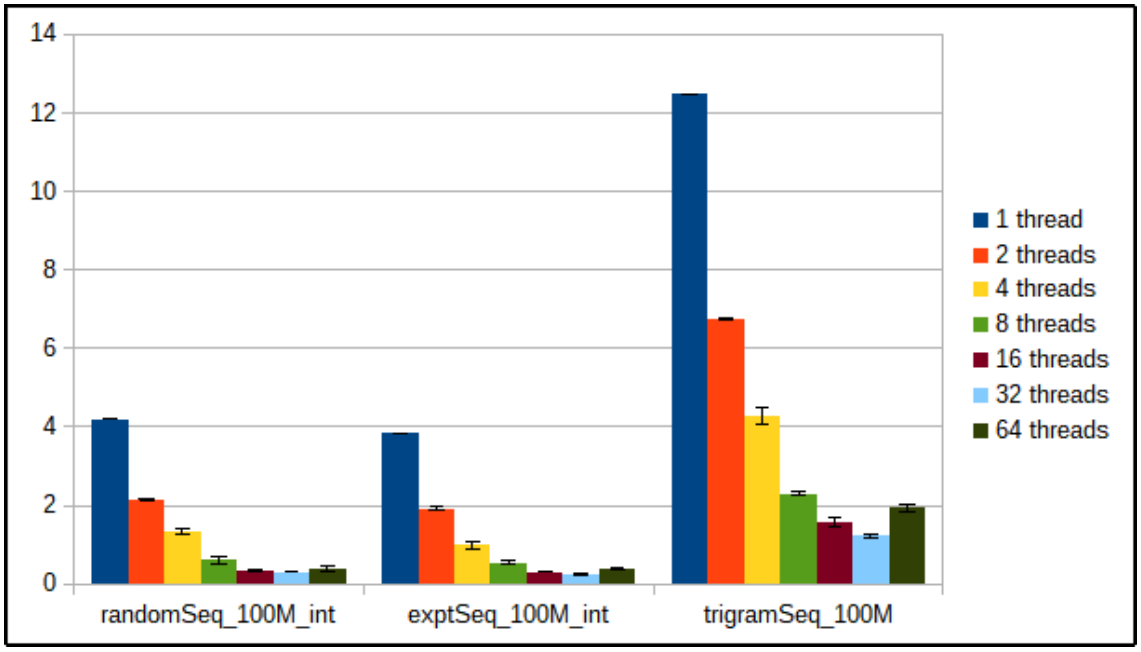


Figure A.59: Execution times (in minutes) of Remove Duplicates ran on AMD 32-64 using LCWS with WShare (**continuous tit-for-tat**)

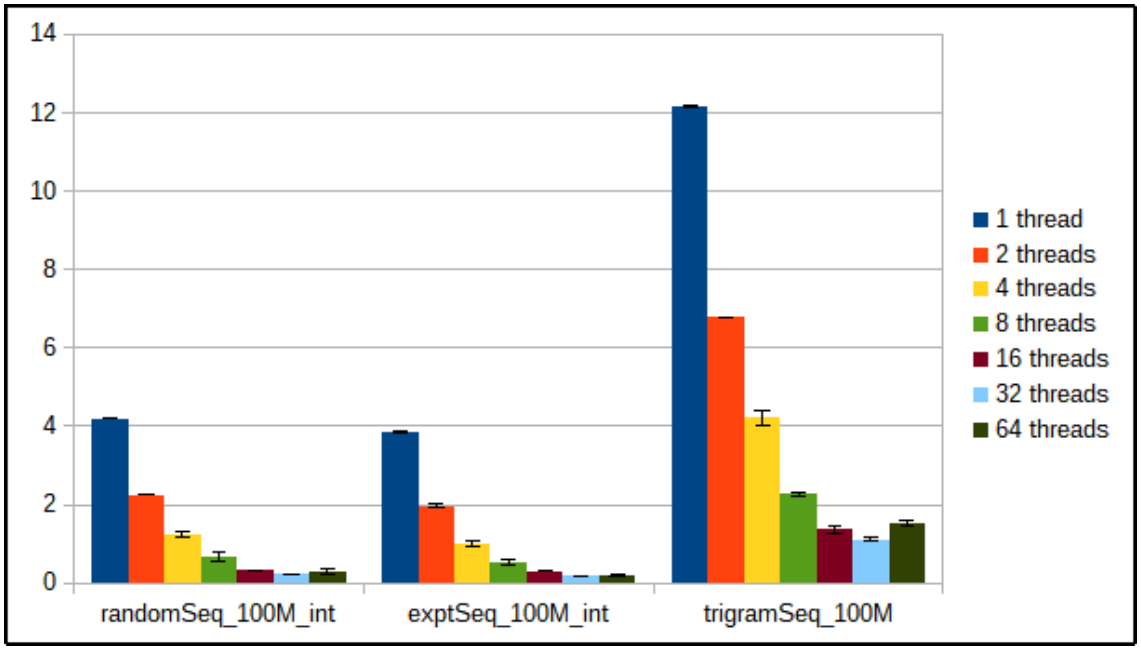


Figure A.60: Execution times (in minutes) of Remove Duplicates ran on AMD 32-64 using SBLCWS with WShare (**first version**)

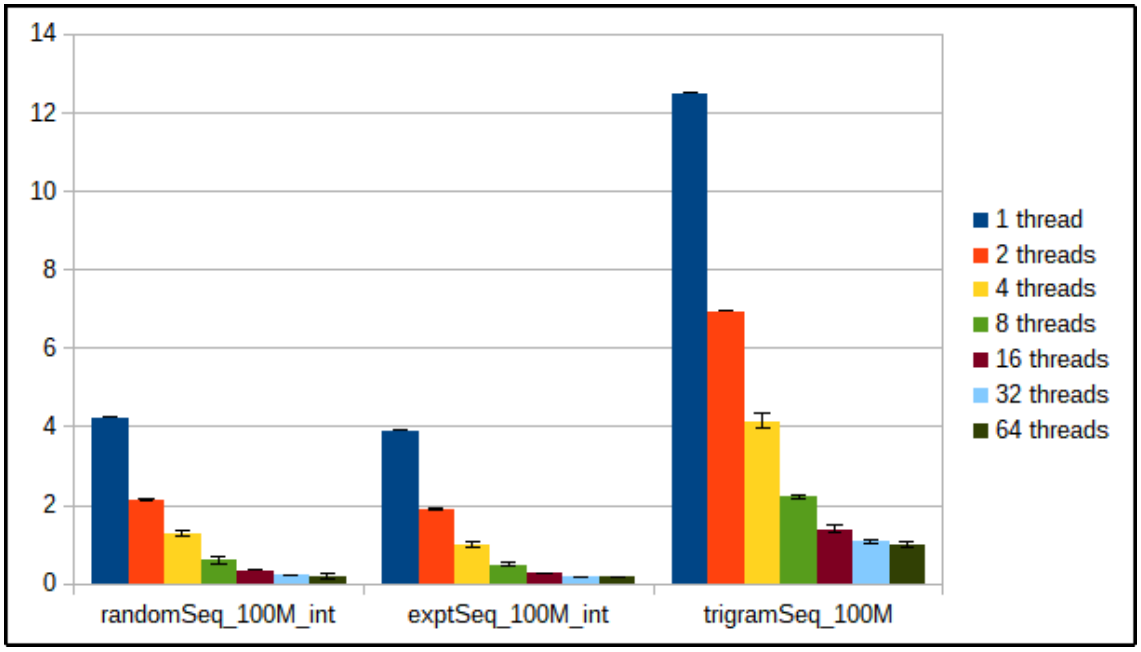


Figure A.61: Execution times (in minutes) of Remove Duplicates ran on AMD 32-64 using SBLCWS with WShare (**second version**)

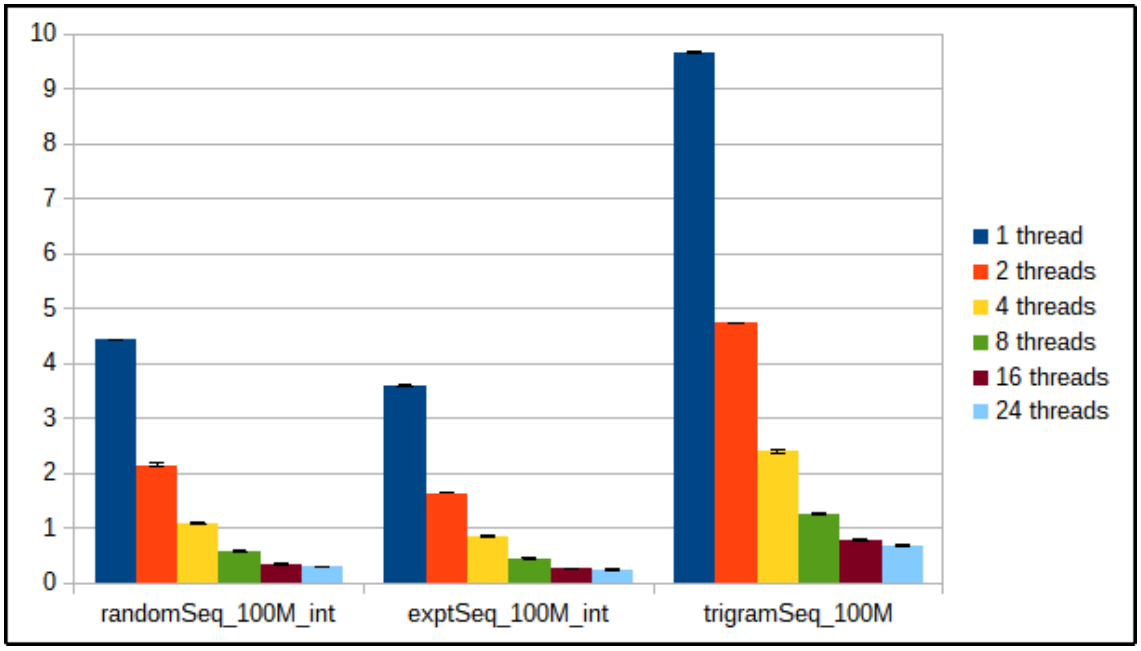


Figure A.62: Execution times (in minutes) of Remove Duplicates ran on Intel 16-16 using WSteal

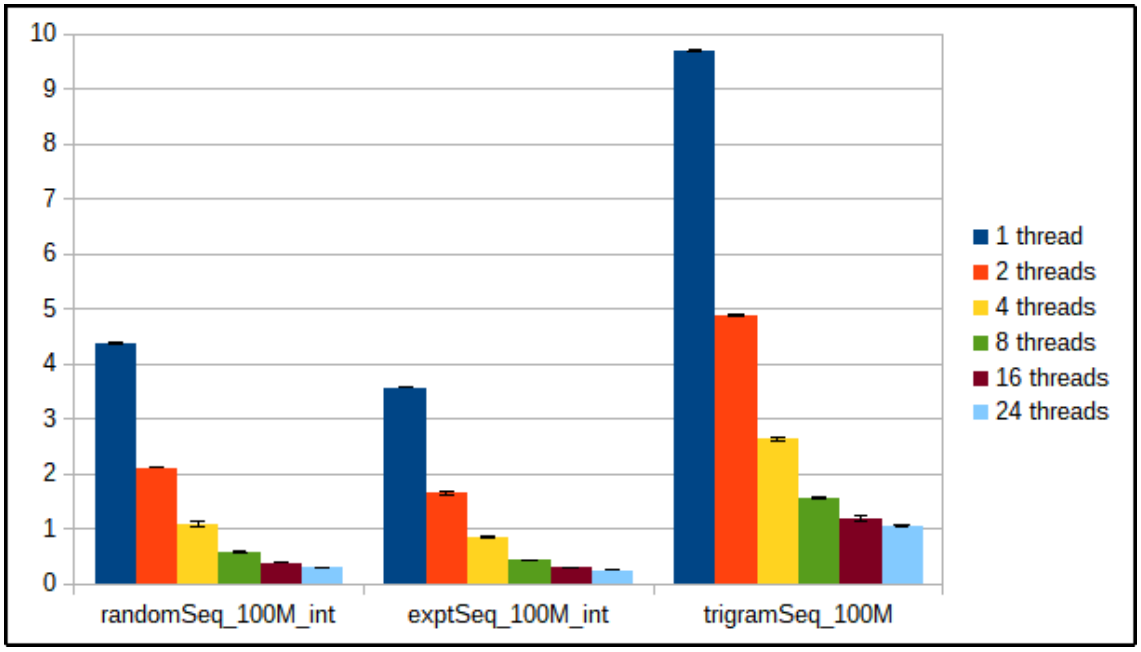


Figure A.63: Execution times (in minutes) of Remove Duplicates ran on Intel 16-16 using LCWS

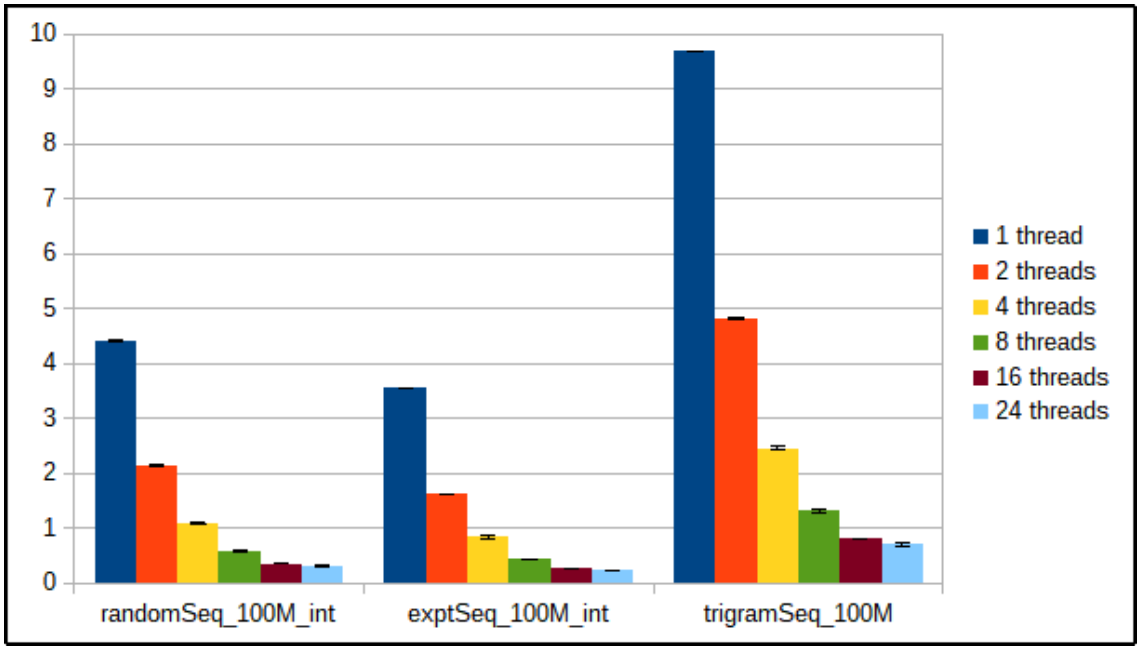


Figure A.64: Execution times (in minutes) of Remove Duplicates ran on Intel 16-16 using SBLCWS (**first version**)

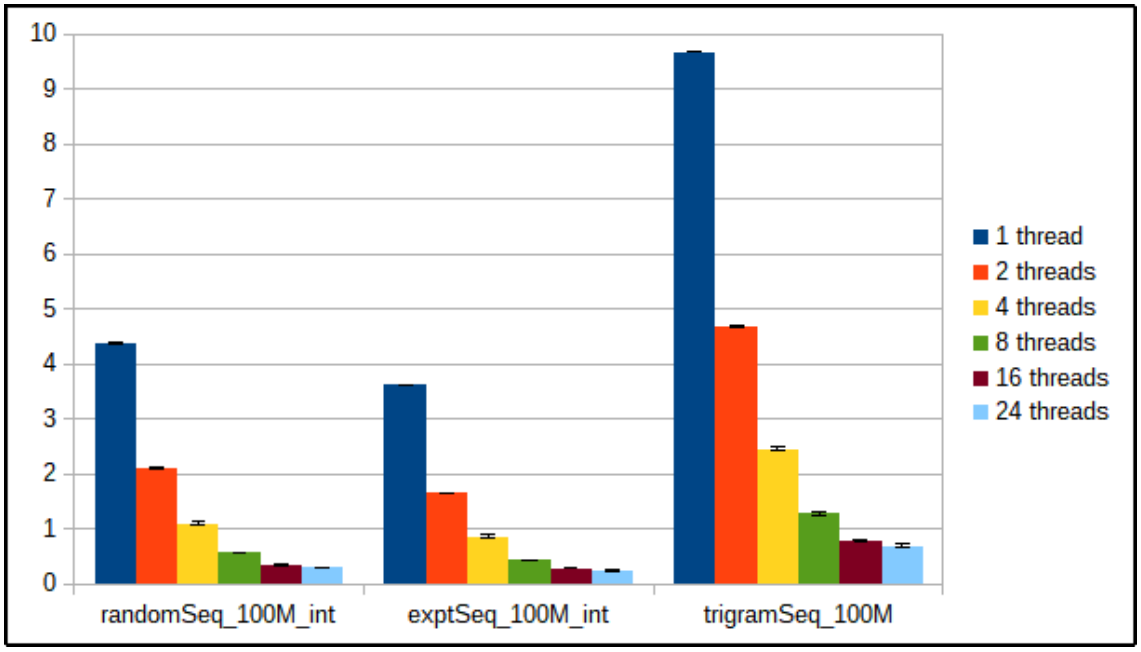


Figure A.65: Execution times (in minutes) of Remove Duplicates ran on Intel 16-16 using SBLCWS (**second version**)

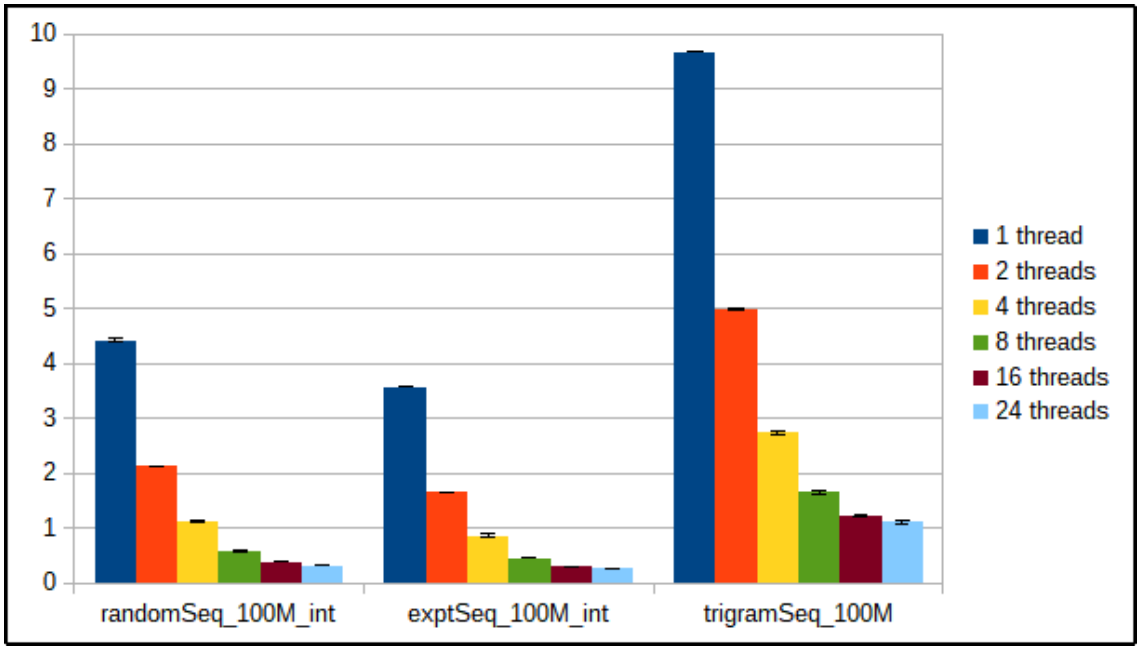


Figure A.66: Execution times (in minutes) of Remove Duplicates ran on Intel 16-16 using LCWS with WShare (**tit-for-tat**).

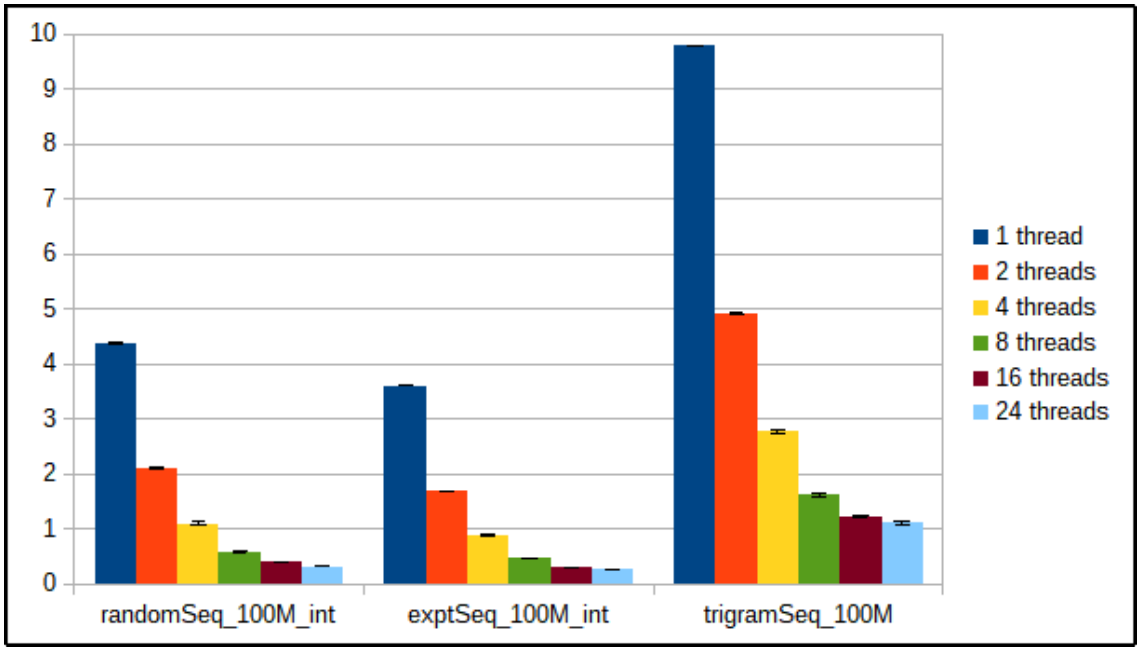


Figure A.67: Execution times (in minutes) of Remove Duplicates ran on Intel 16-16 using LCWS with WShare (**continuous tit-for-tat**).

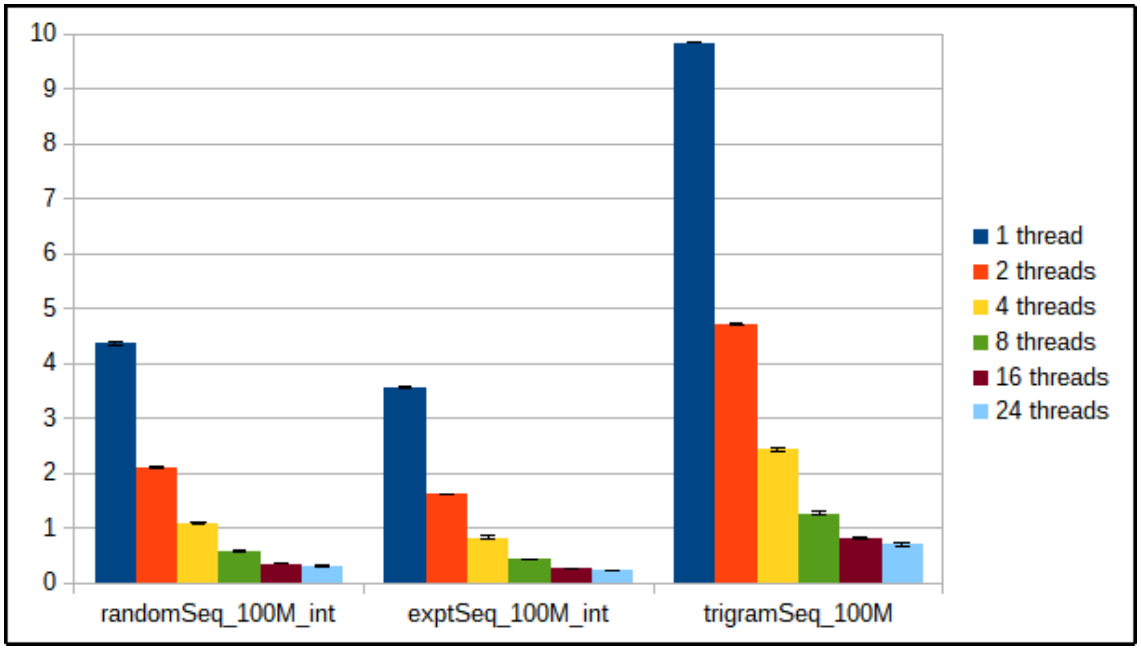


Figure A.68: Execution times (in minutes) of Remove Duplicates ran on Intel 16-16 using SBLCWS with WShare (**first version**).

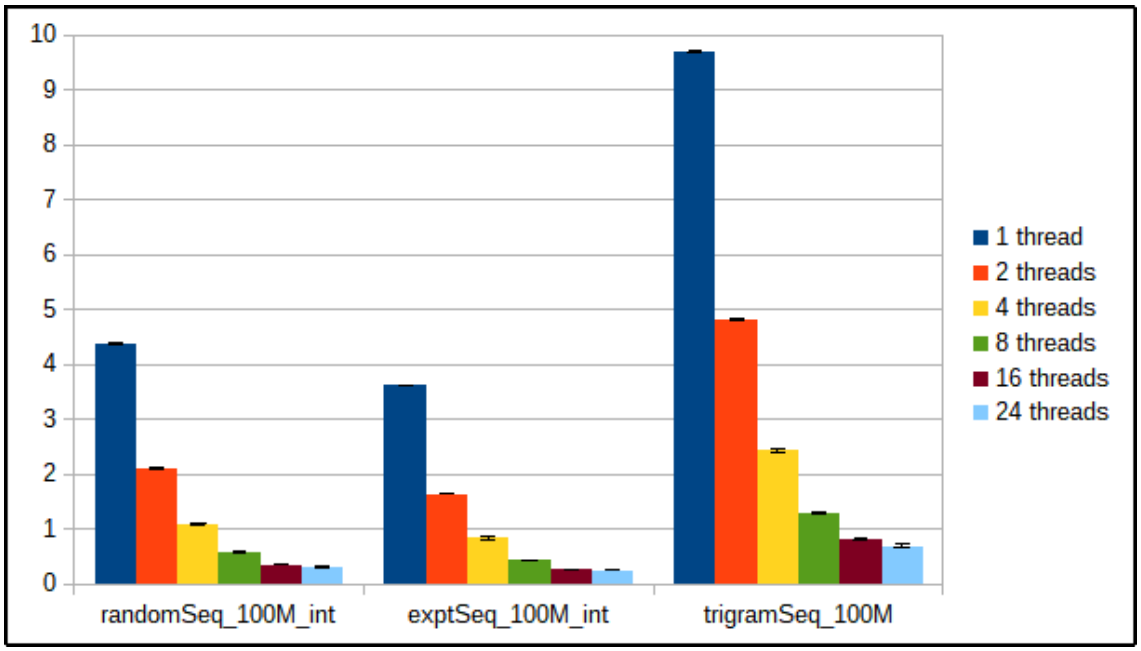


Figure A.69: Execution times (in minutes) of Remove Duplicates ran on Intel 16-16 using SBLCWS with WShare (**second version**).

A.4 Histogram

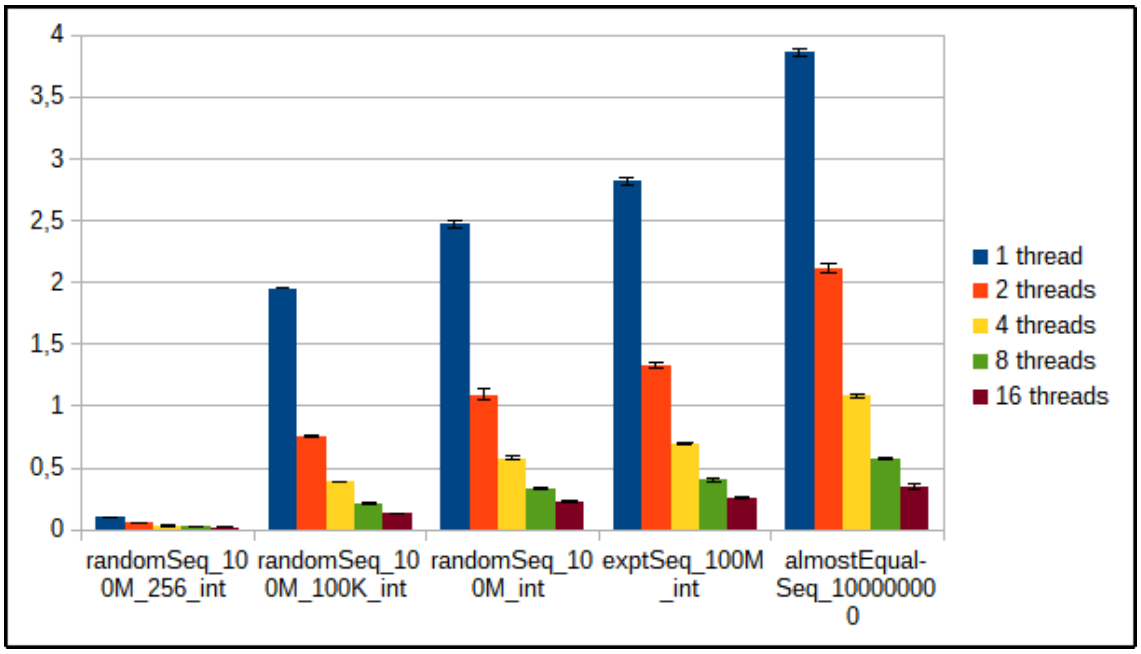


Figure A.70: Execution times (in minutes) of Histogram ran on Intel 12-24 using WSteal

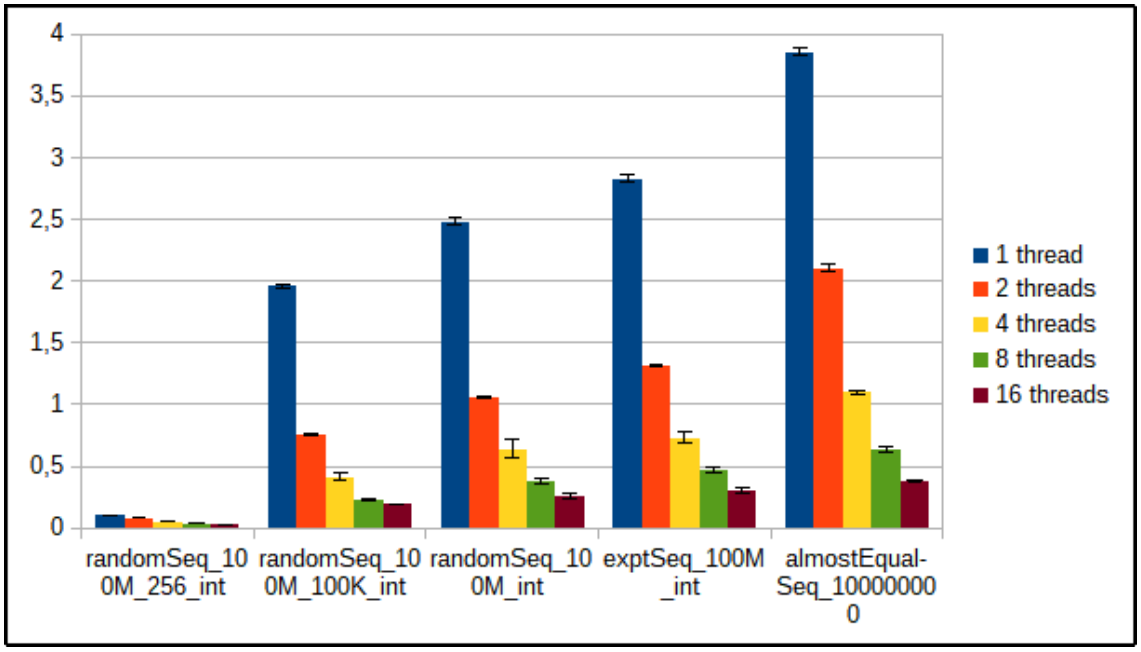


Figure A.71: Execution times (in minutes) of Histogram ran on Intel 12-24 using LCWS

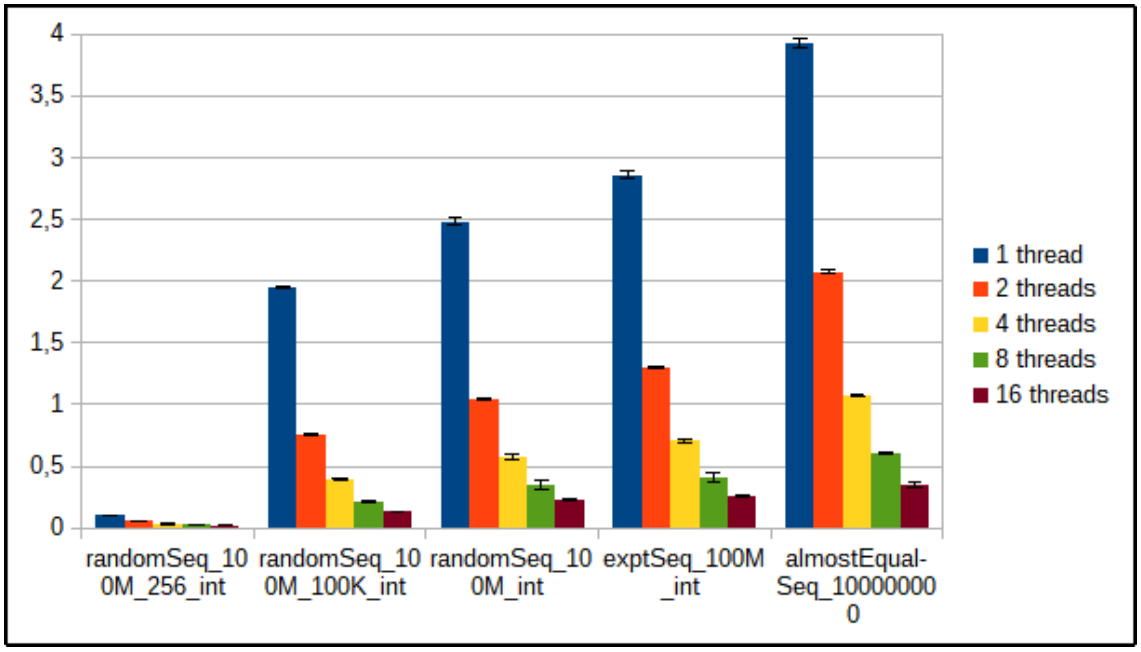


Figure A.72: Execution times (in minutes) of Histogram ran on Intel 12-24 using SBLCWS (**first version**)

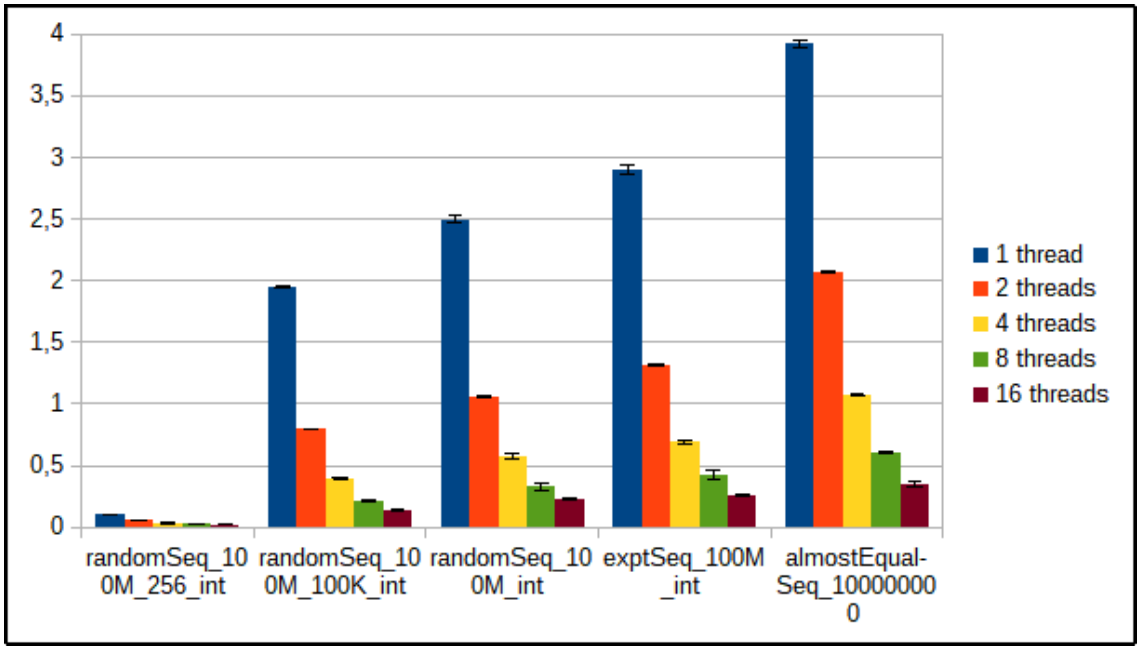


Figure A.73: Execution times (in minutes) of Histogram ran on Intel 12-24 using SBLCWS (**second version**)

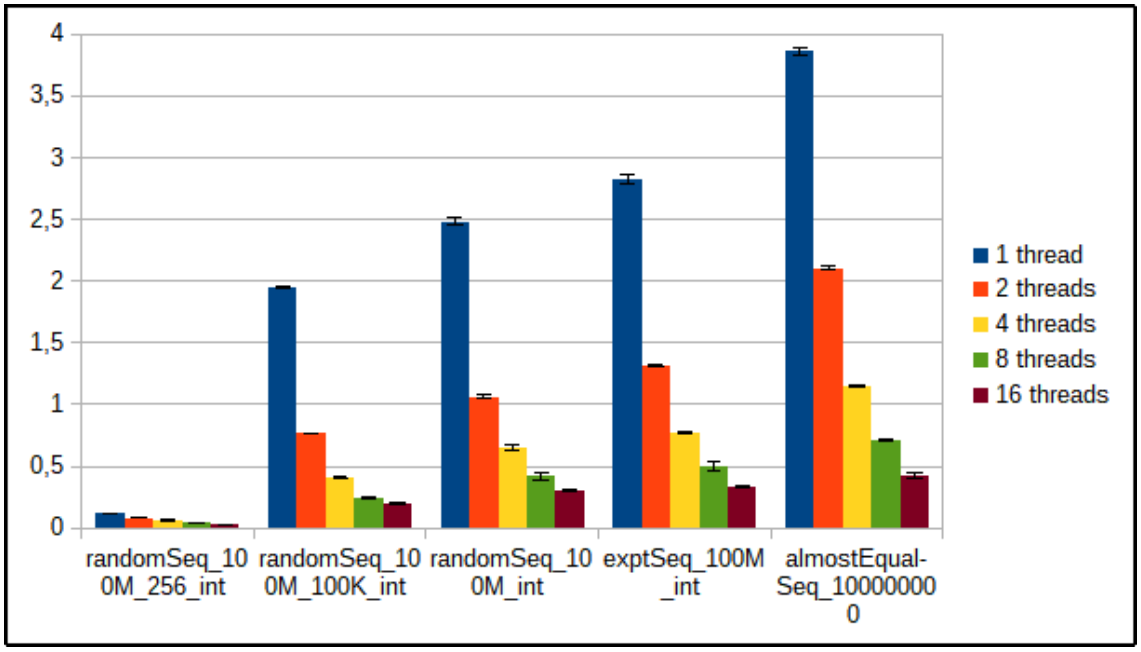


Figure A.74: Execution times (in minutes) of Histogram ran on Intel 12-24 using LCWS with WShare (**tit-for-tat**).

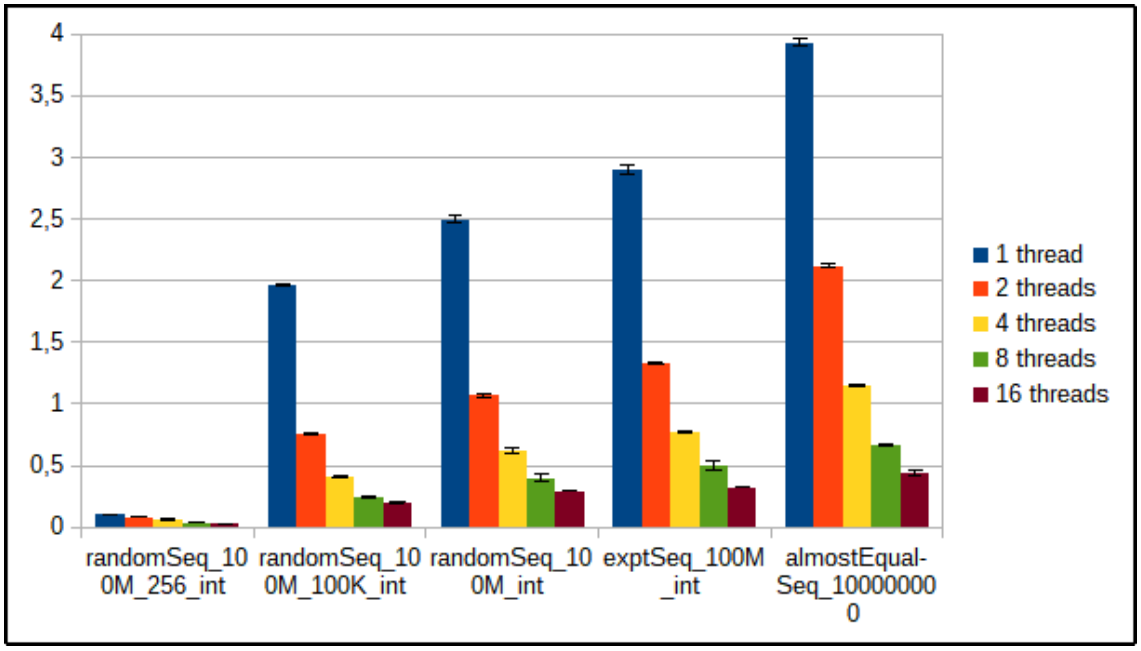


Figure A.75: Execution times (in minutes) of Histogram ran on Intel 12-24 using LCWS with WShare (**continuous tit-for-tat**).

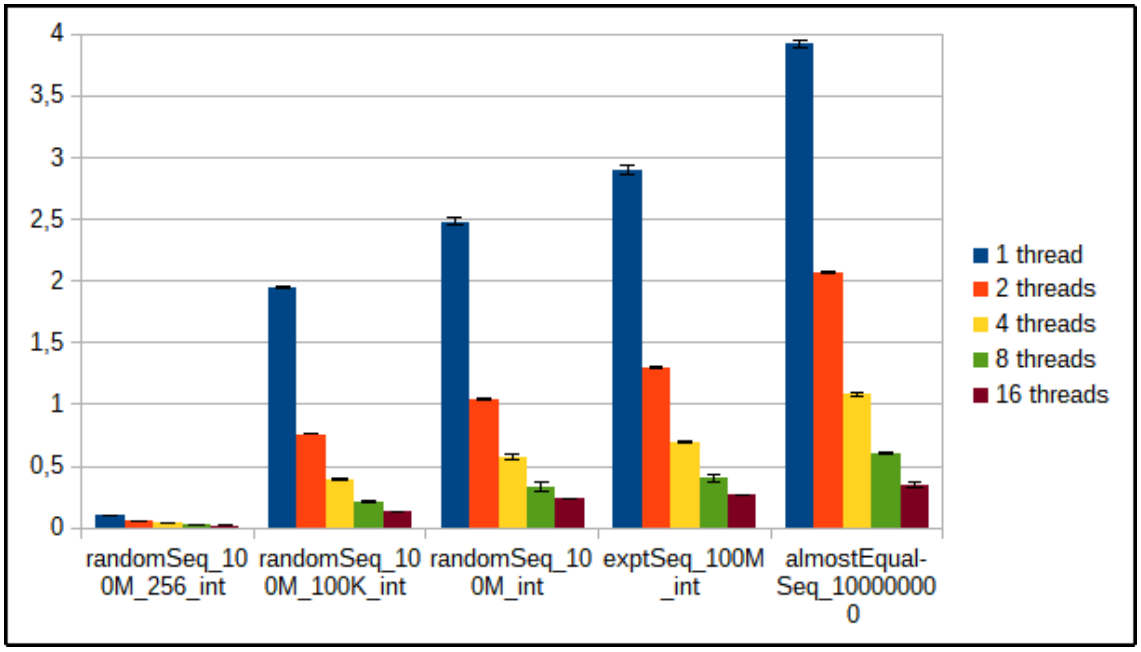


Figure A.76: Execution times (in minutes) of Histogram ran on Intel 12-24 using SBLCWS with WShare (**first version**).

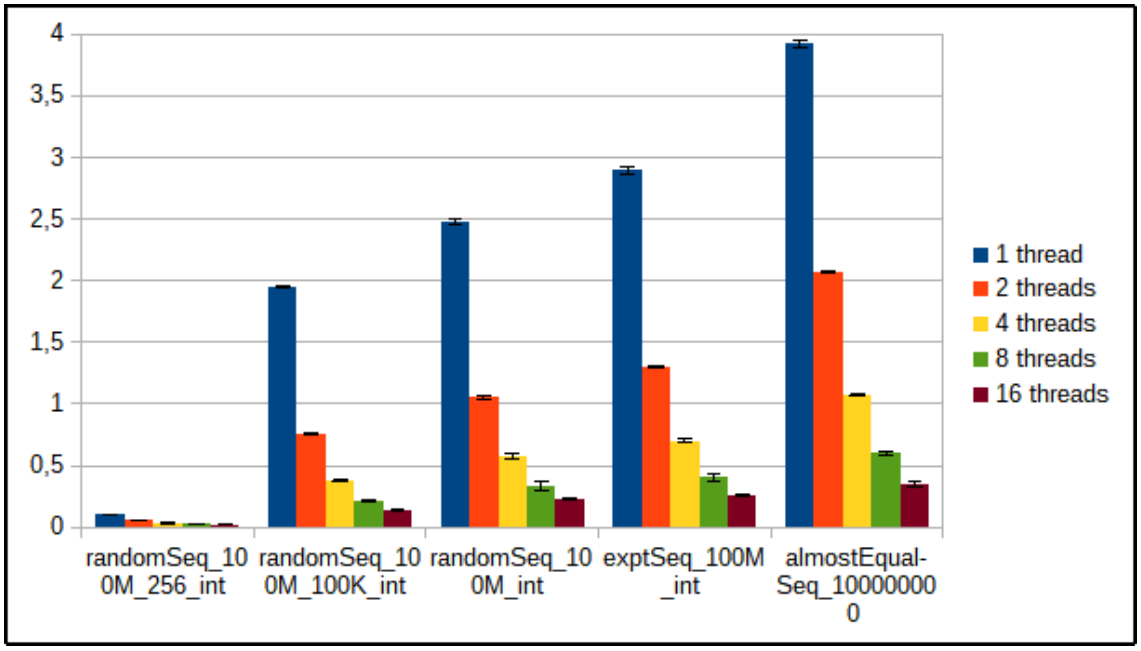


Figure A.77: Execution times (in minutes) of Histogram ran on Intel 12-24 using SBLCWS with WShare (**second version**).

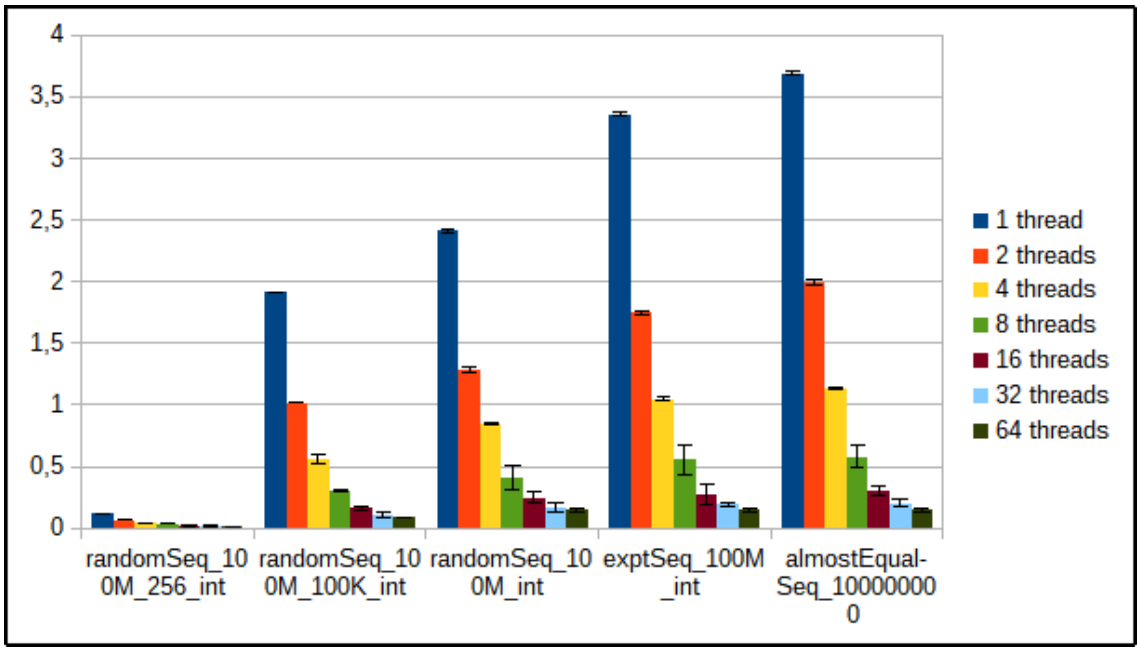


Figure A.78: Execution times (in minutes) of Histogram ran on AMD 32-64 using WSteal

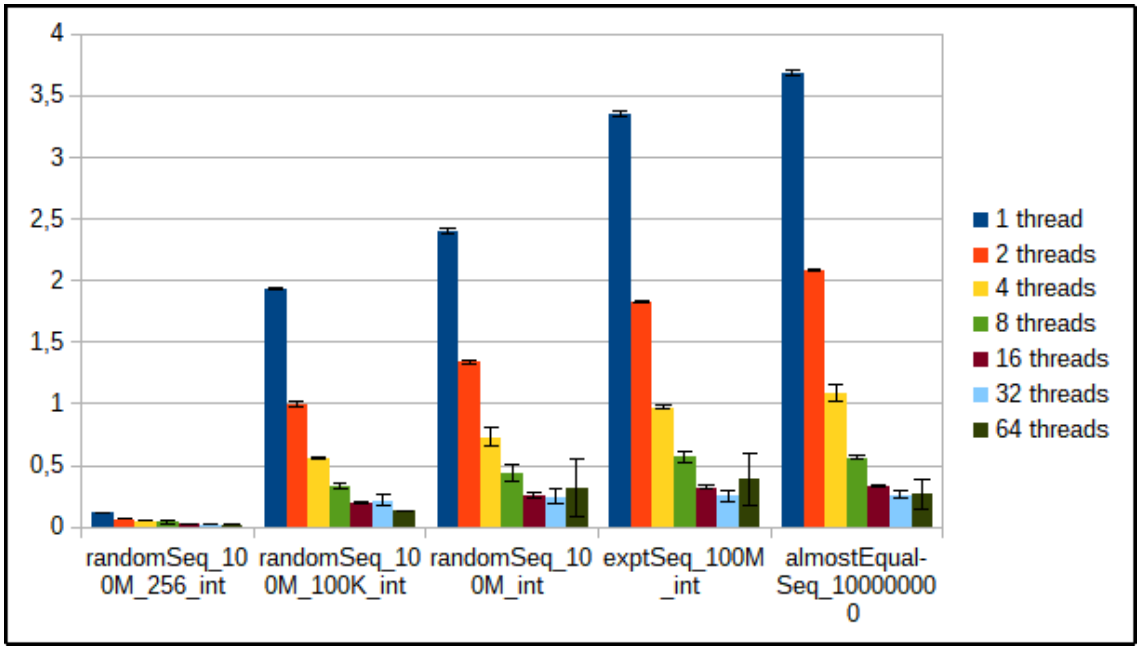


Figure A.79: Execution times (in minutes) of Histogram ran on AMD 32-64 using LCWS

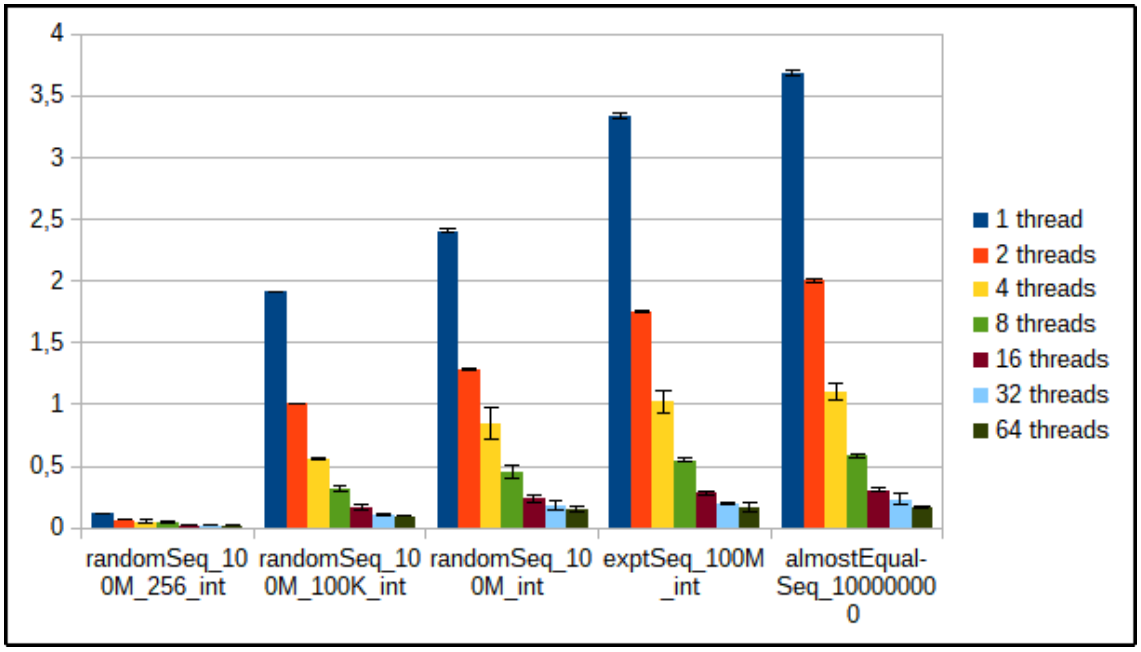


Figure A.80: Execution times (in minutes) of Histogram ran on AMD 32-64 using SBLCWS (**first version**)

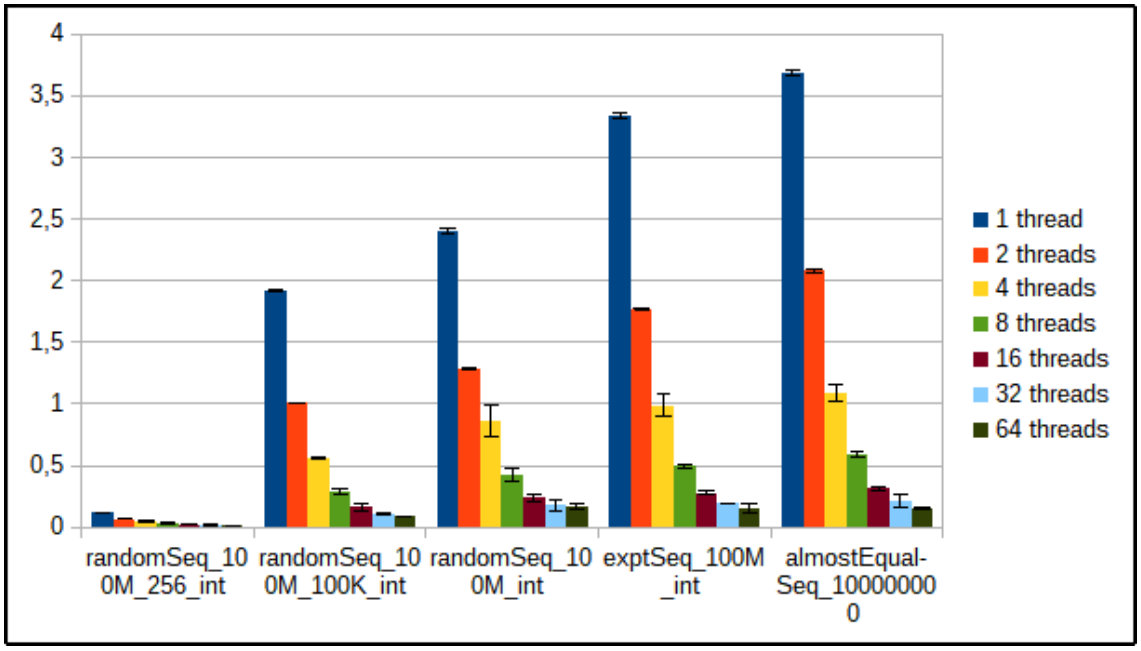


Figure A.81: Execution times (in minutes) of Histogram ran on AMD 32-64 using SBLCWS (**second version**)

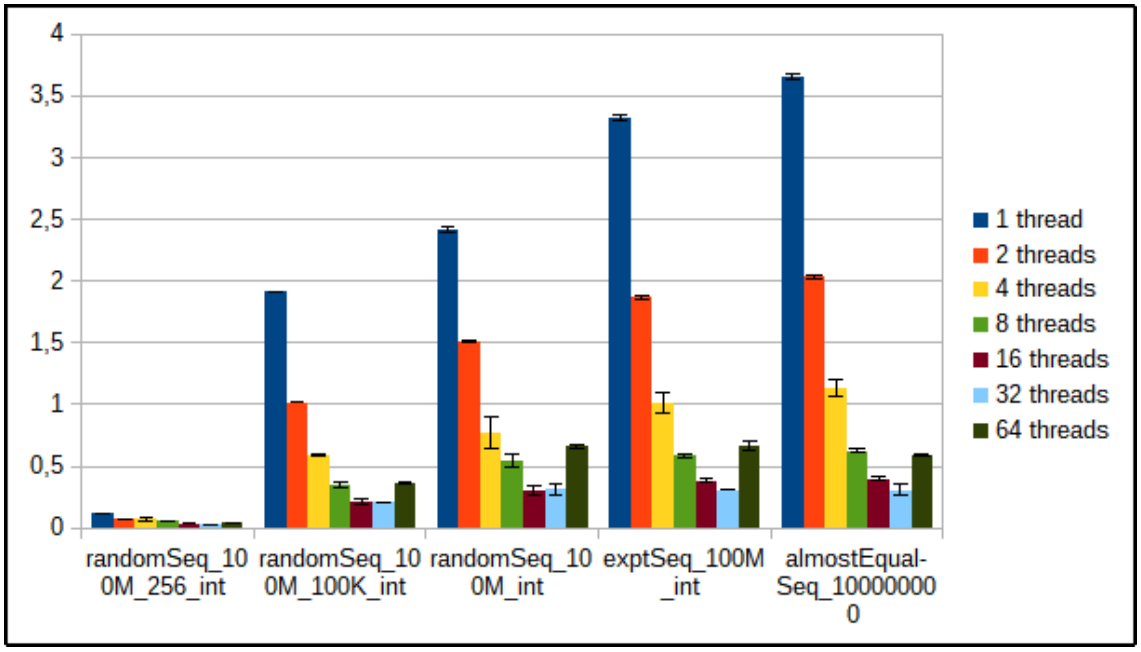


Figure A.82: Execution times (in minutes) of Histogram ran on AMD 32-64 using LCWS with WShare (**tit-for-tat**).

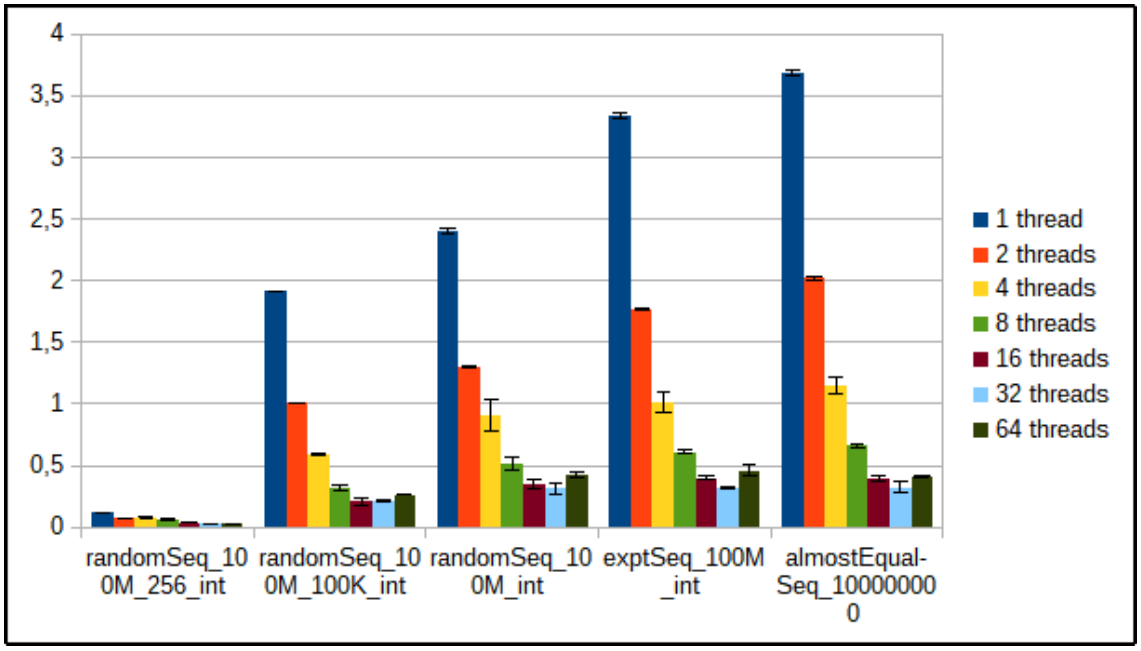


Figure A.83: Execution times (in minutes) of Histogram ran on AMD 32-64 using LCWS with WShare (**continuous tit-for-tat**).

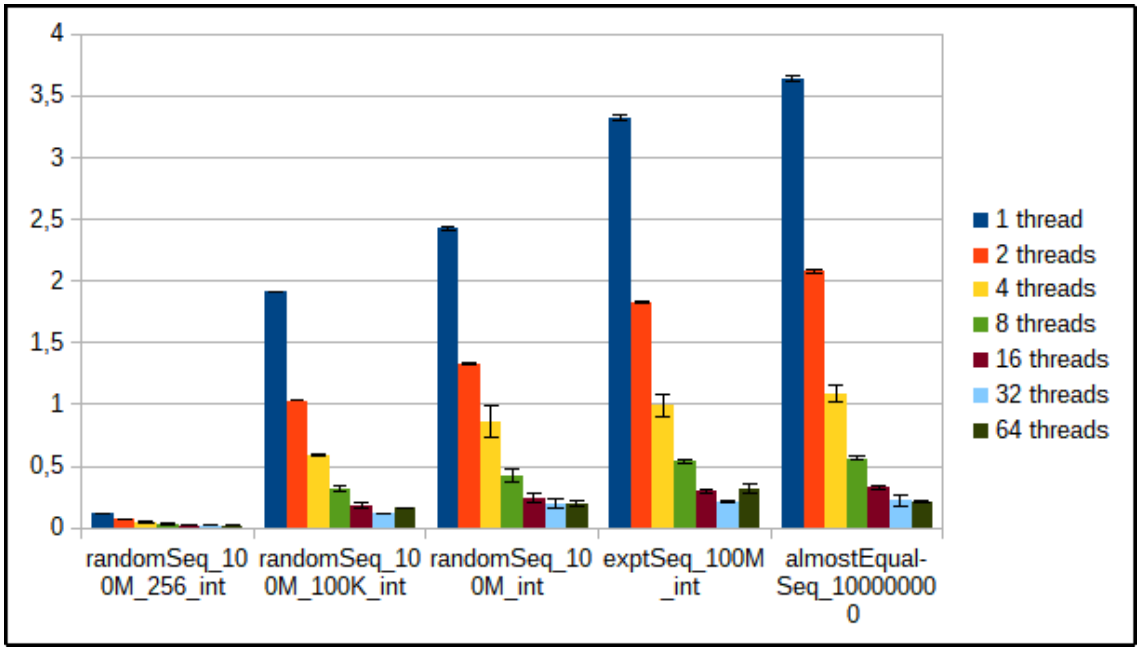


Figure A.84: Execution times (in minutes) of Histogram ran on AMD 32-64 using SBLCWS with WShare (**first version**).

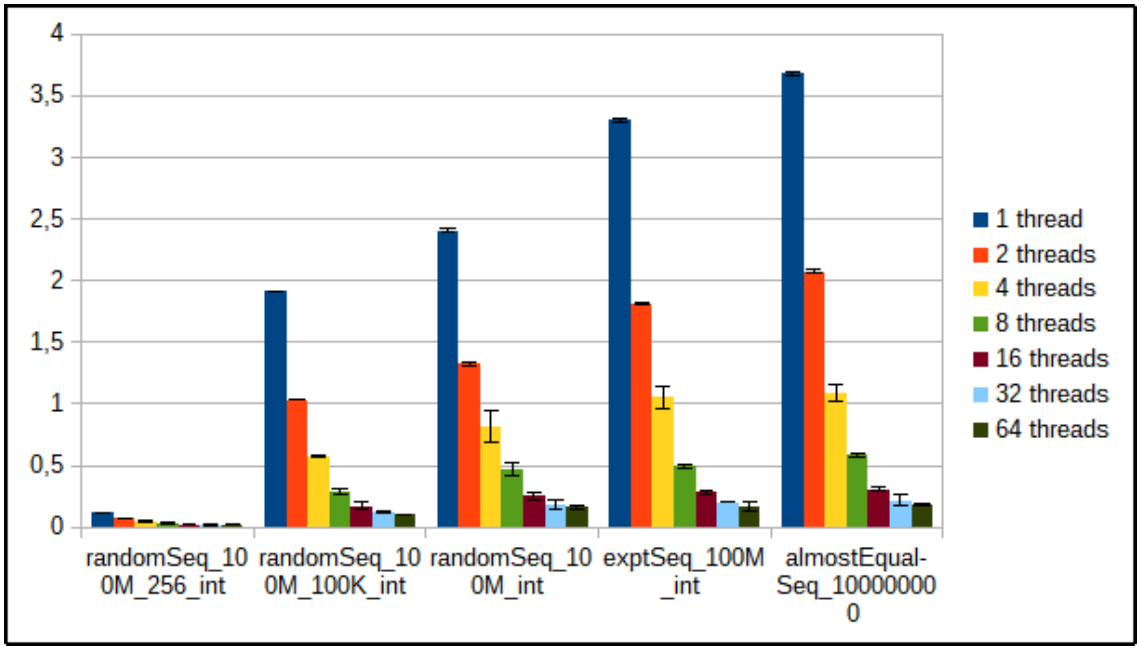


Figure A.85: Execution times (in minutes) of Histogram ran on AMD 32-64 using SBLCWS with WShare (**second version**).

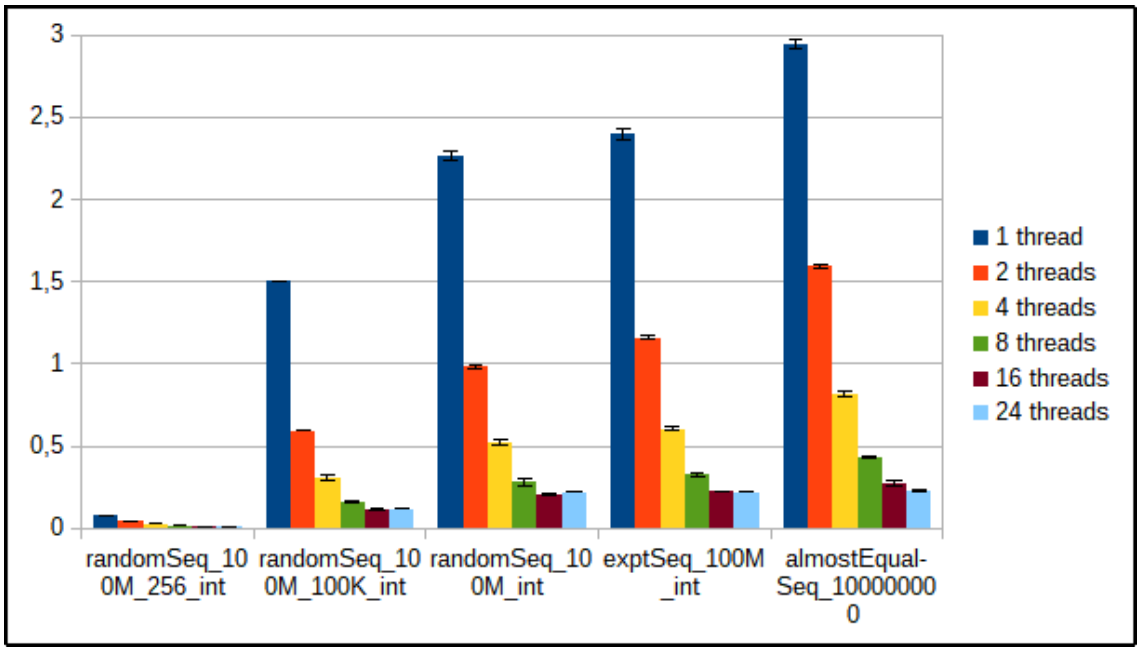


Figure A.86: Execution times (in minutes) of Histogram ran on Intel 16-16 using WSteal

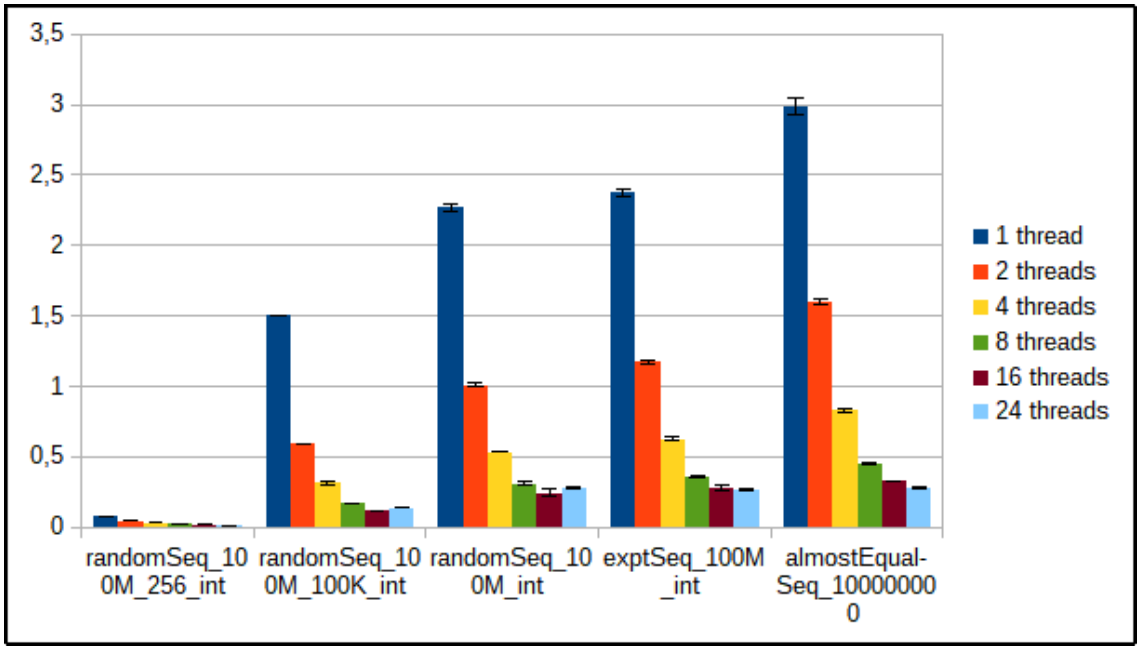


Figure A.87: Execution times (in minutes) of Histogram ran on Intel 16-16 using LCWS

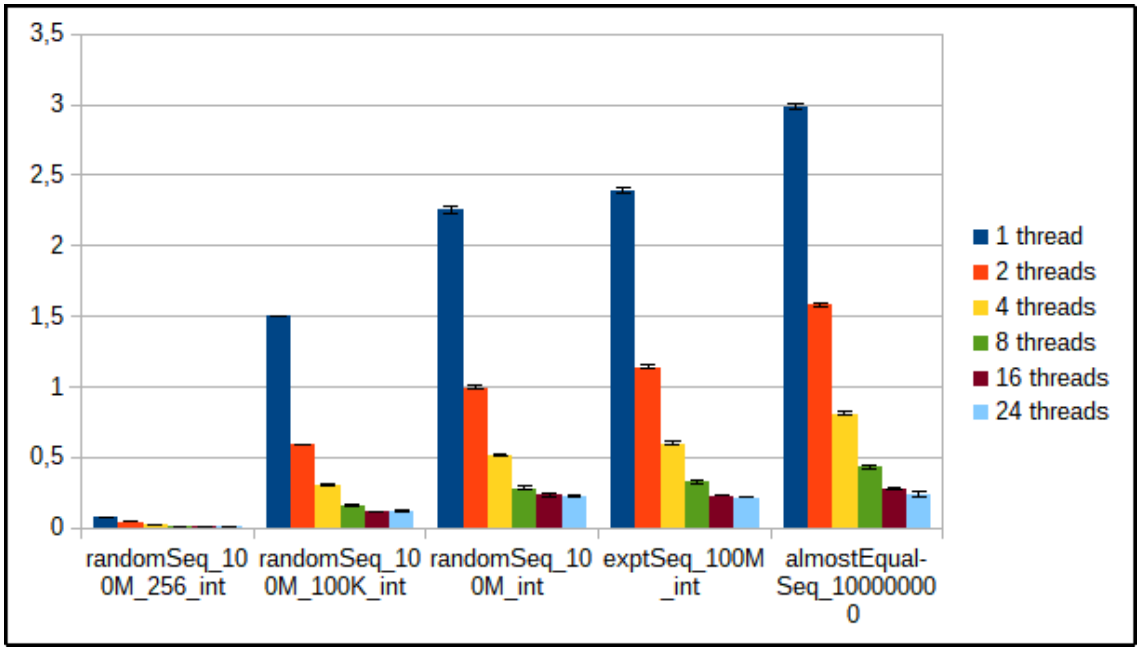


Figure A.88: Execution times (in minutes) of Histogram ran on Intel 16-16 using SBLCWS (**first version**)

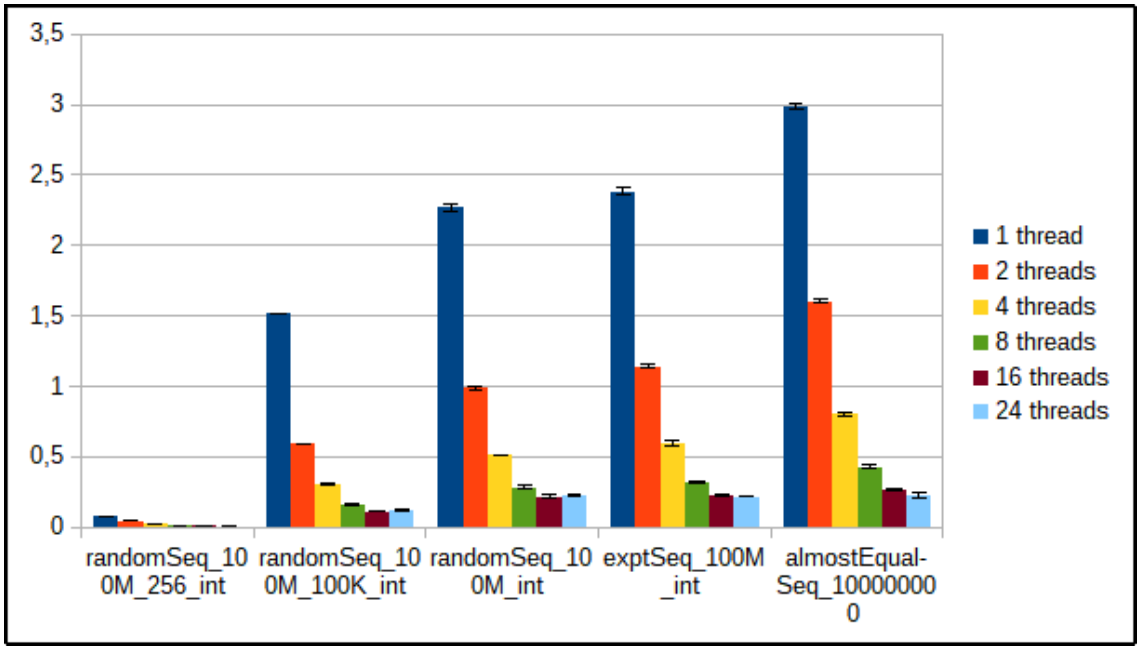


Figure A.89: Execution times (in minutes) of Histogram ran on Intel 16-16 using SBLCWS (**second version**)

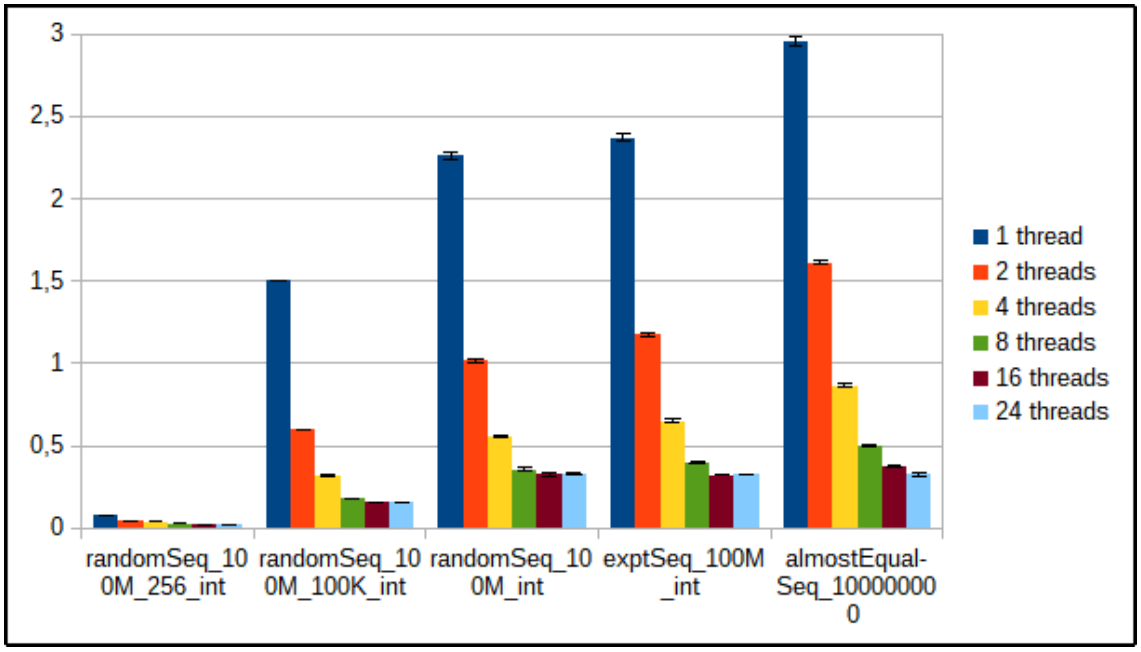


Figure A.90: Execution times (in minutes) of Histogram ran on Intel 16-16 using LCWS with WShare (**tit-for-tat**).

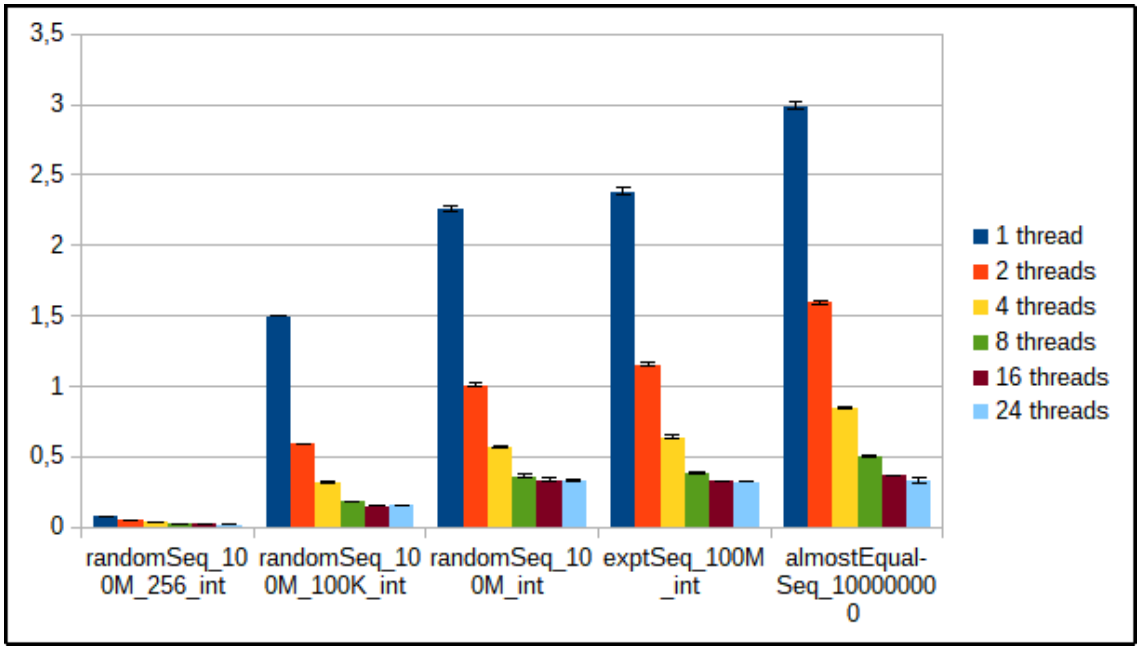


Figure A.91: Execution times (in minutes) of Histogram ran on Intel 16-16 using LCWS with WShare (**continuous tit-for-tat**).

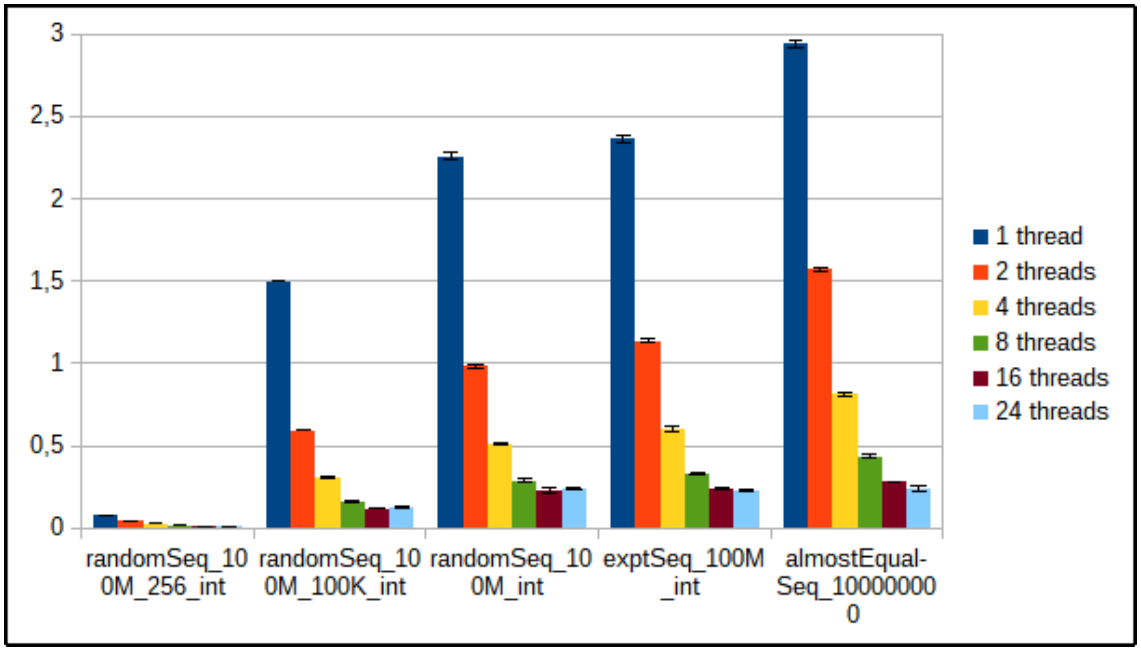


Figure A.92: Execution times (in minutes) of Histogram ran on Intel 16-16 using SBLCWS with WShare (**first version**).

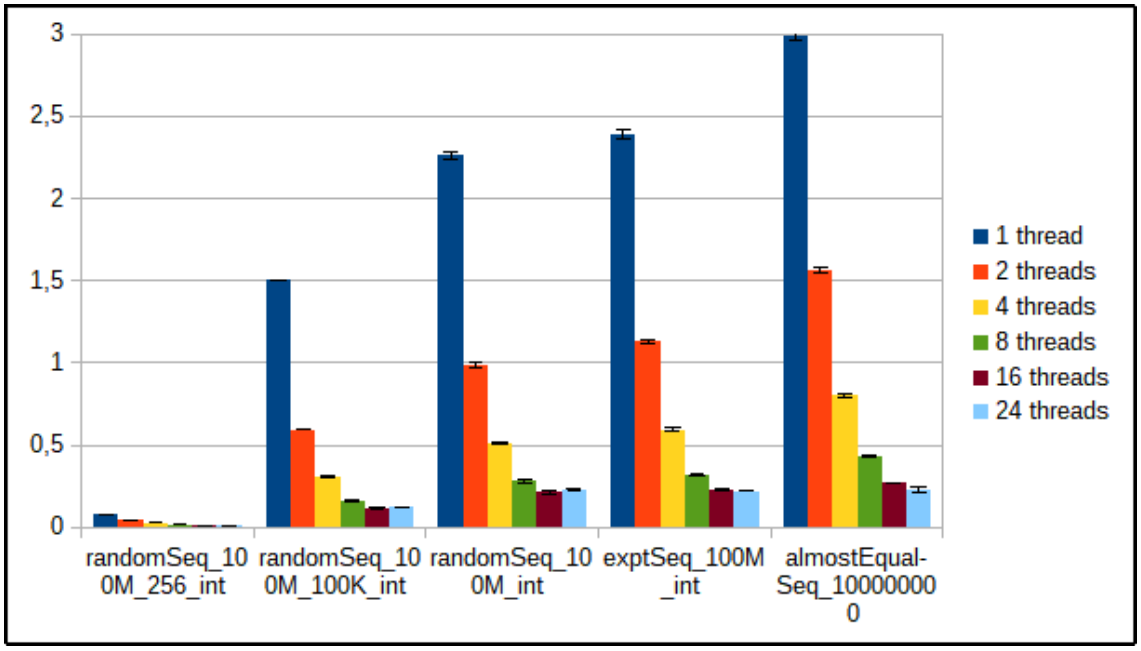


Figure A.93: Execution times (in minutes) of Histogram ran on Intel 16-16 using SBLCWS with WShare (**second version**).

A.5 Word Count

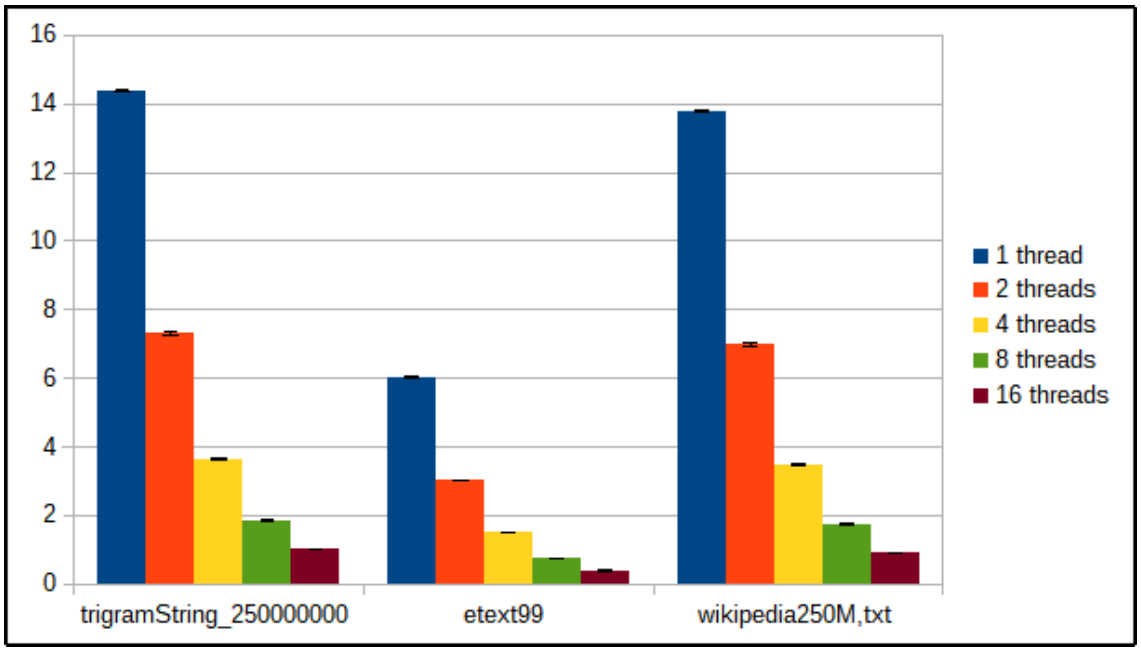


Figure A.94: Execution times (in minutes) of Word Counts ran on Intel 12-24 using WSteal

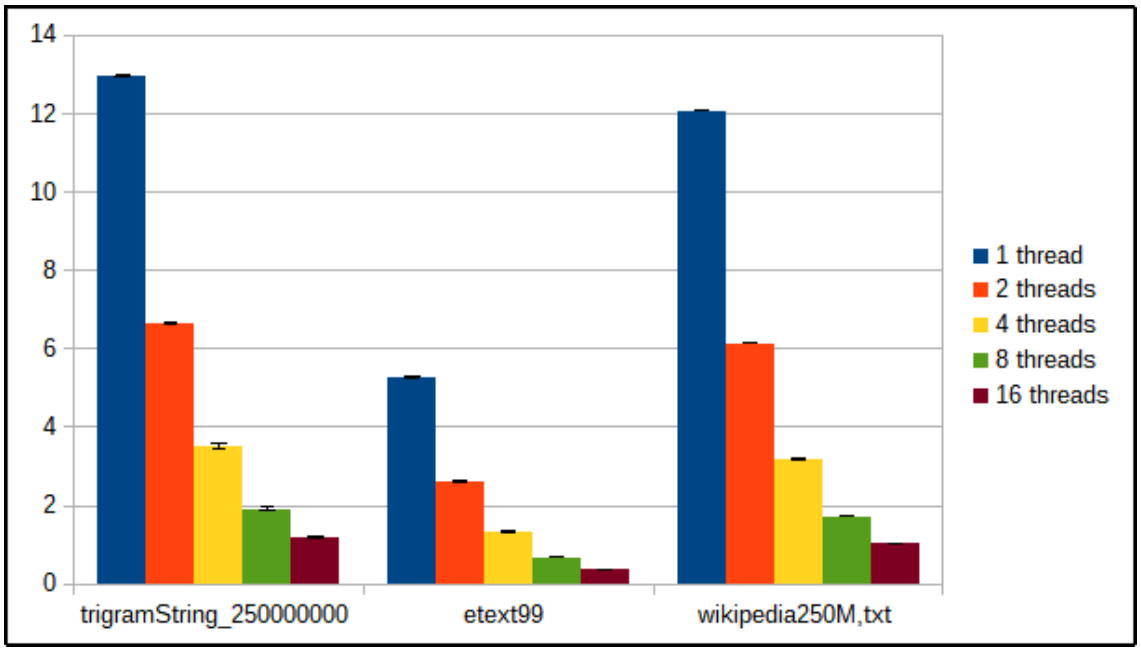


Figure A.95: Execution times (in minutes) of Word Counts ran on Intel 12-24 using LCWS

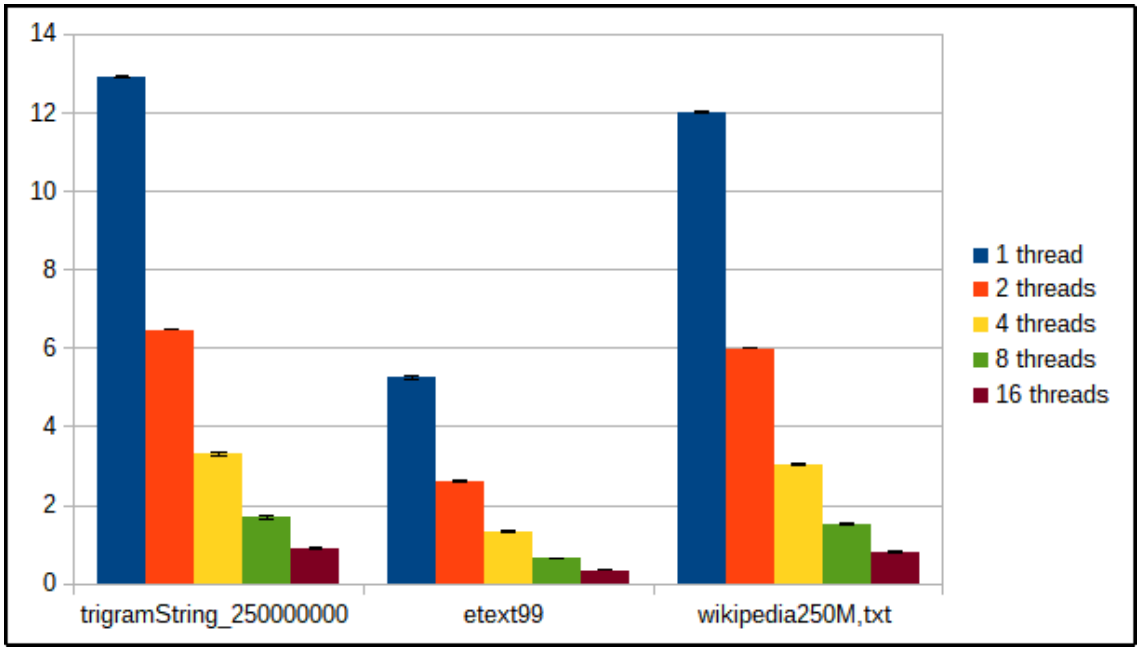


Figure A.96: Execution times (in minutes) of Word Counts ran on Intel 12-24 using SBLCWS (**first version**)

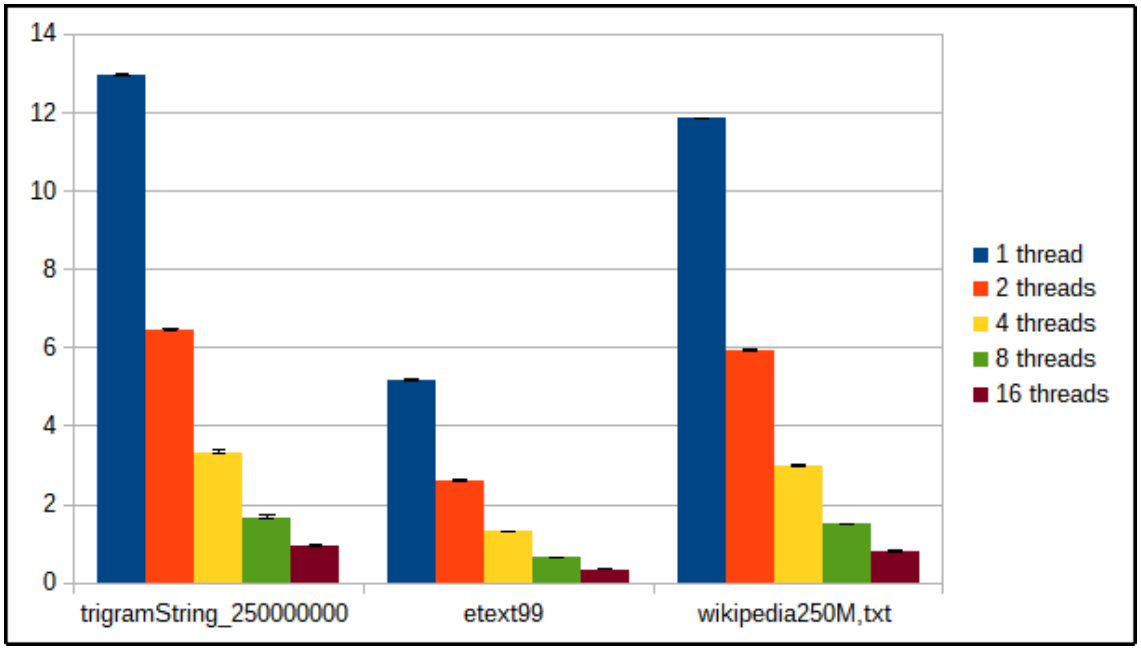


Figure A.97: Execution times (in minutes) of Word Counts ran on Intel 12-24 using SBLCWS (**second version**)

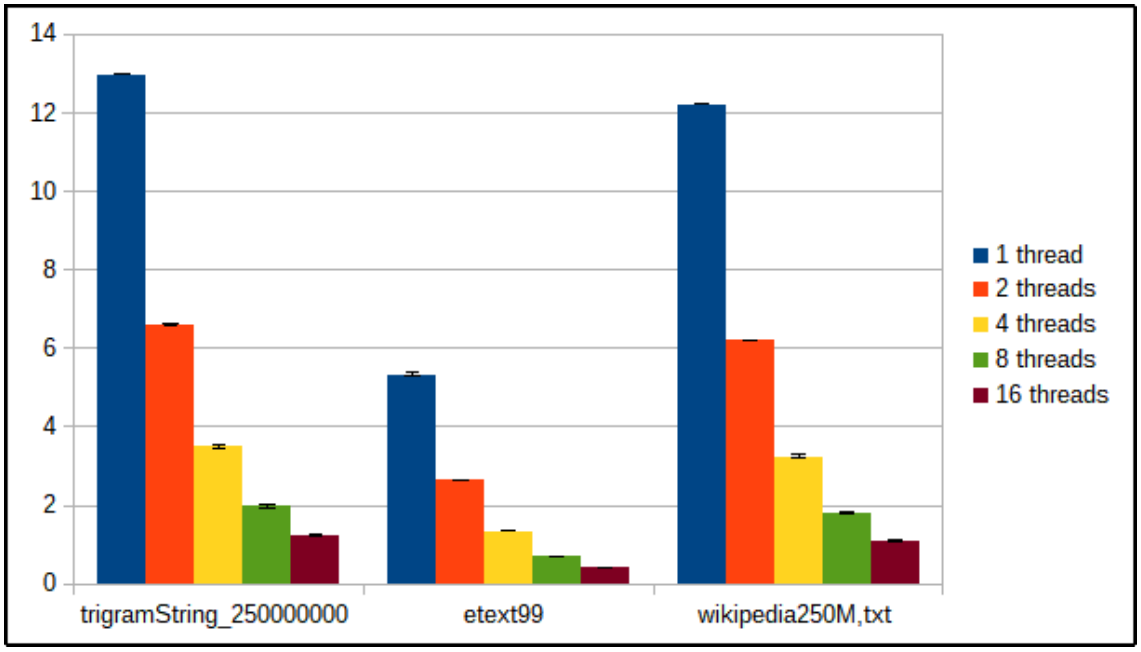


Figure A.98: Execution times (in minutes) of Word Counts ran on Intel 12-24 using LCWS with WShare (**tit-for-tat**).

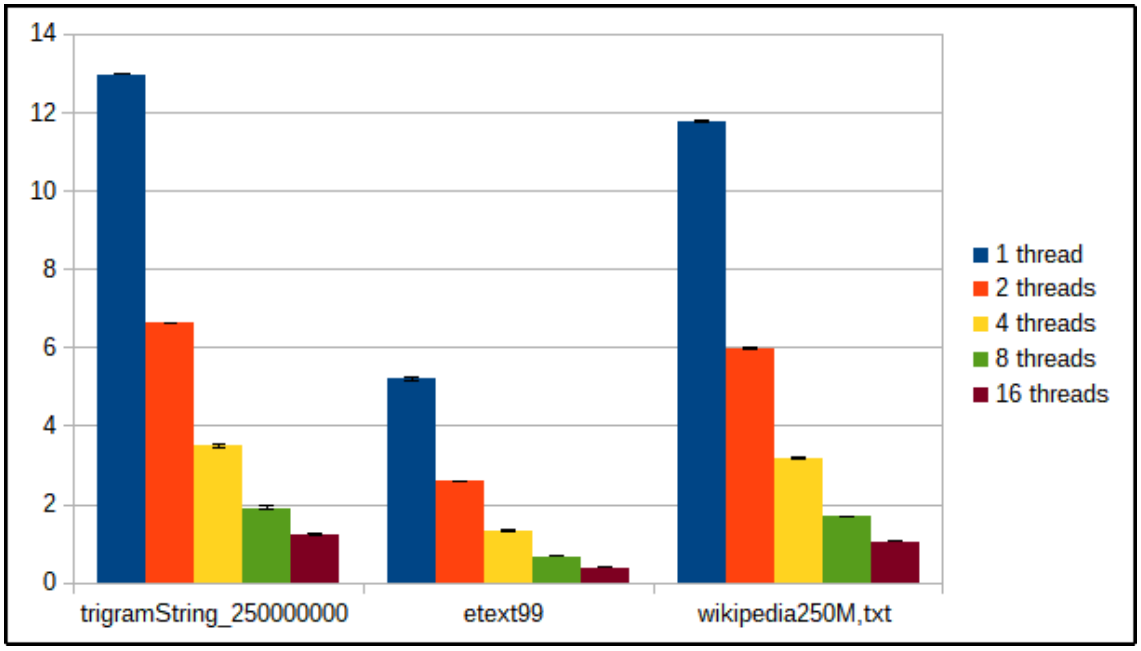


Figure A.99: Execution times (in minutes) of Word Counts ran on Intel 12-24 using LCWS with WShare (**continuous tit-for-tat**).

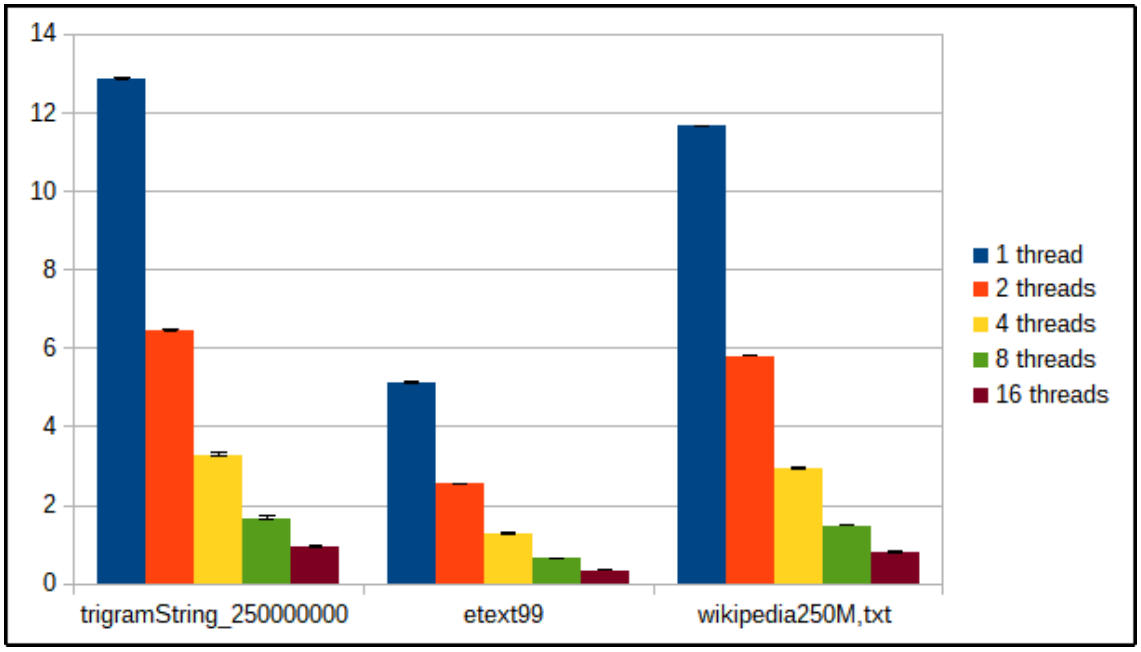


Figure A.100: Execution times (in minutes) of Word Counts ran on Intel 12-24 using SBLCWS with WShare (**first version**).

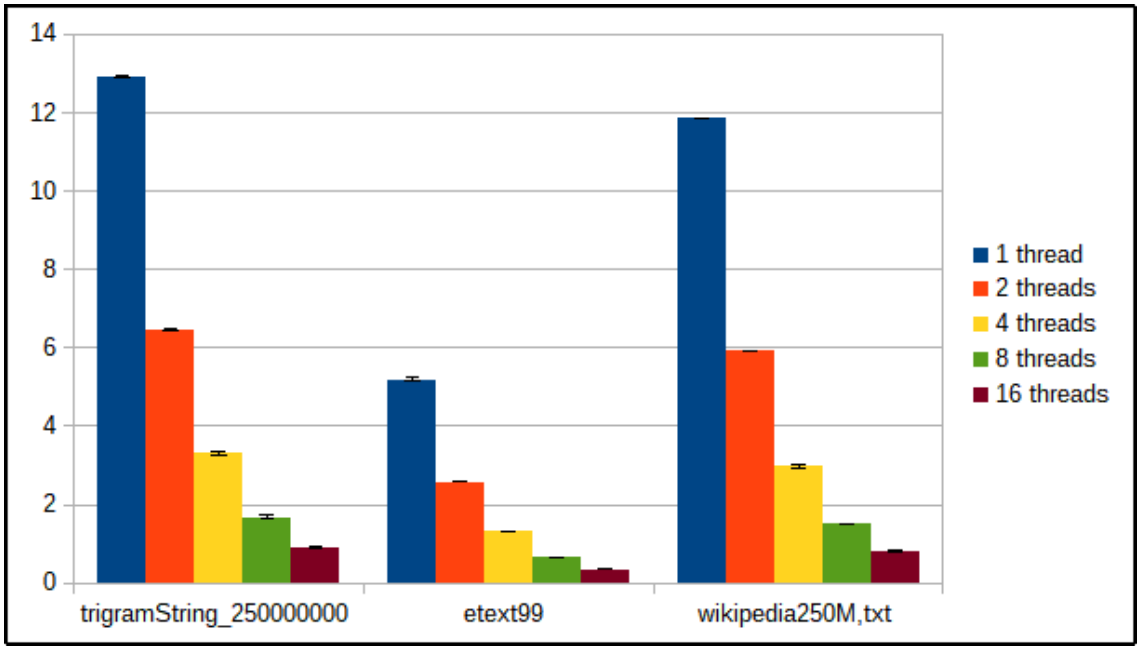


Figure A.101: Execution times (in minutes) of Word Counts ran on Intel 12-24 using SBLCWS with WShare (**second version**).

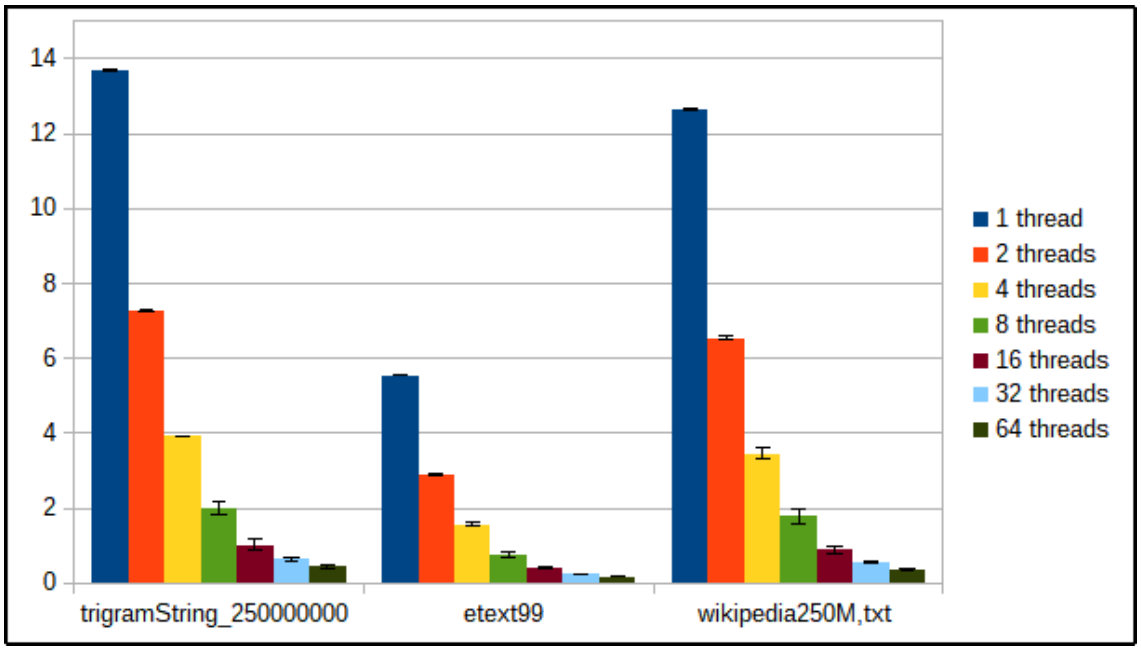


Figure A.102: Execution times (in minutes) of Word Counts ran on AMD 32-64 using WSteal

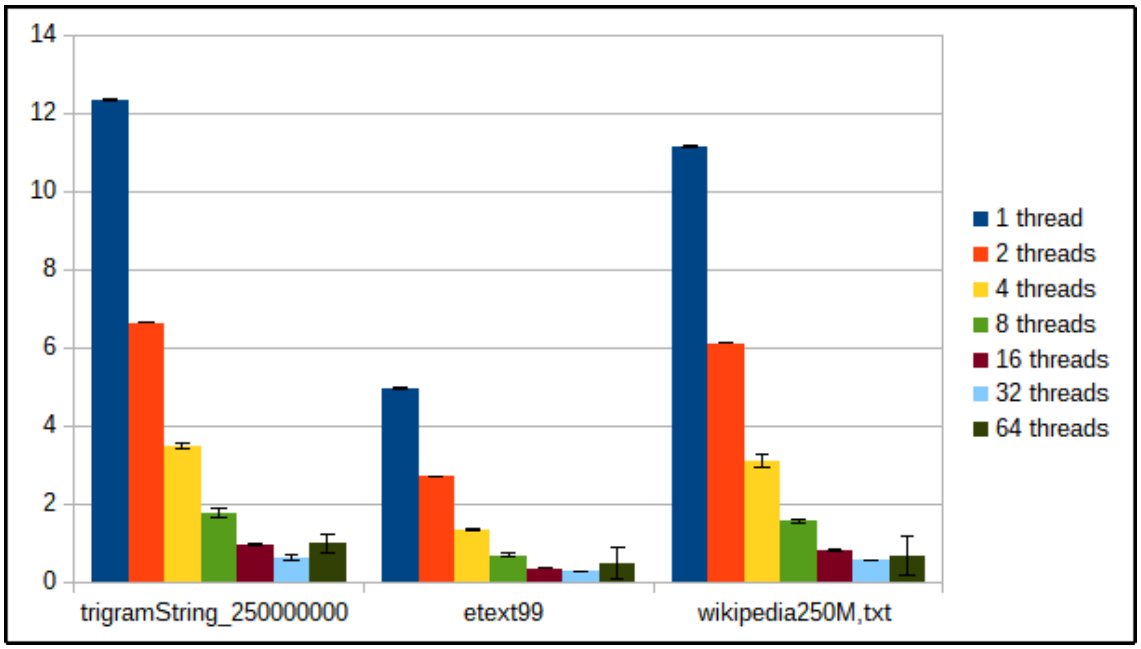


Figure A.103: Execution times (in minutes) of Word Counts ran on AMD 32-64 using LCWS

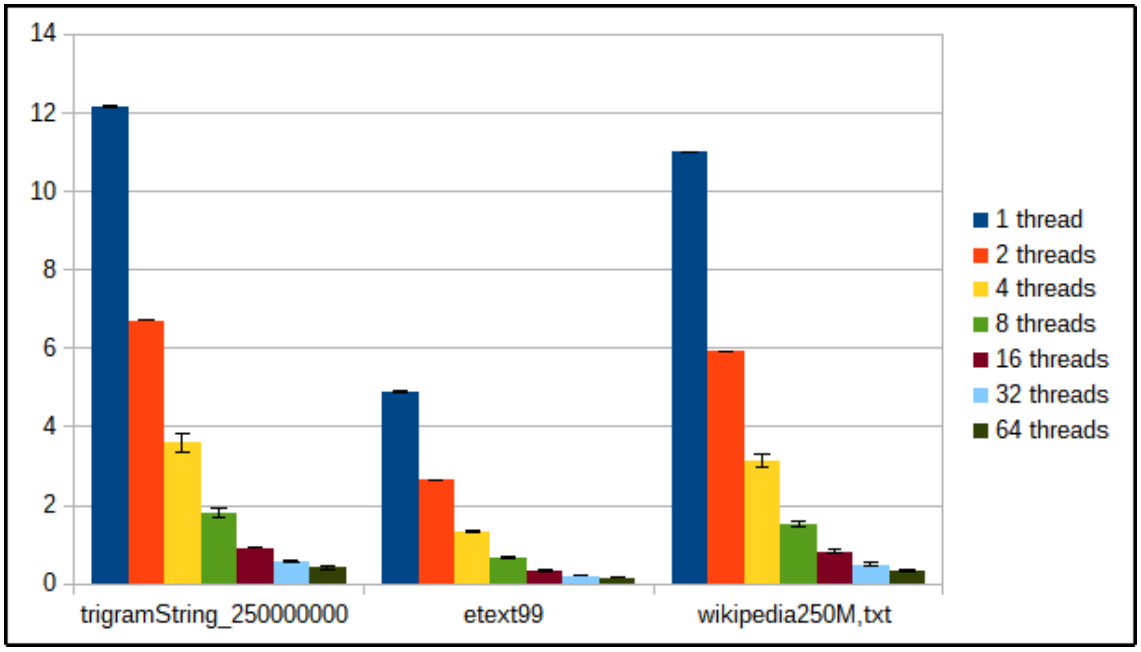


Figure A.104: Execution times (in minutes) of Word Counts ran on AMD 32-64 using SBLCWS (**first version**)

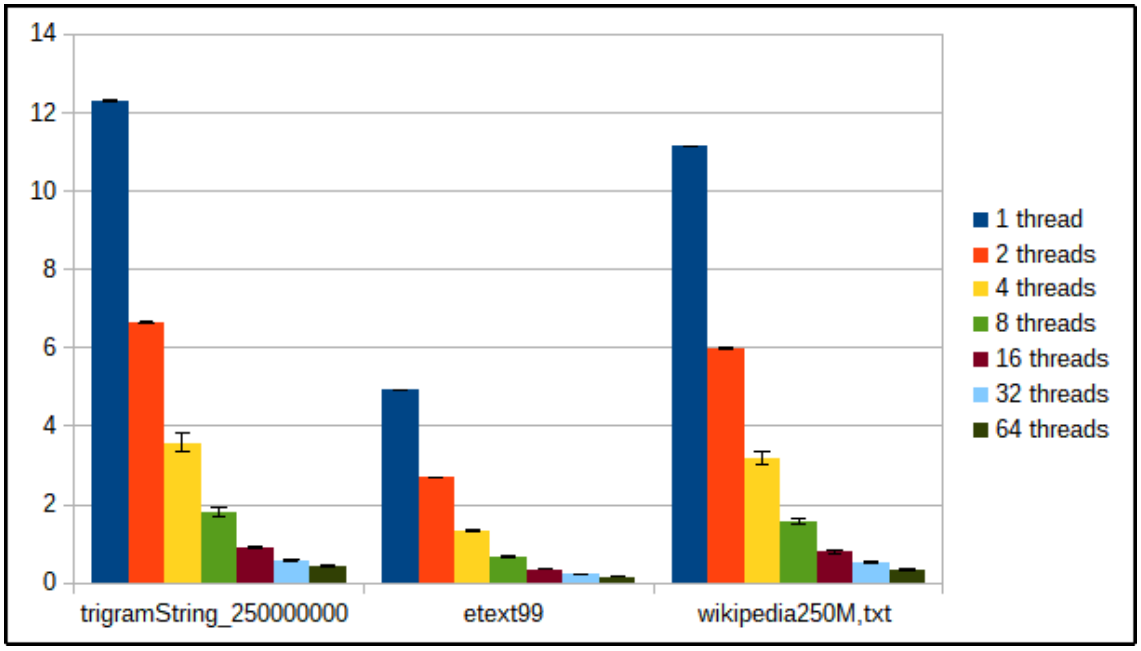


Figure A.105: Execution times (in minutes) of Word Counts ran on AMD 32-64 using SBLCWS (**second version**)

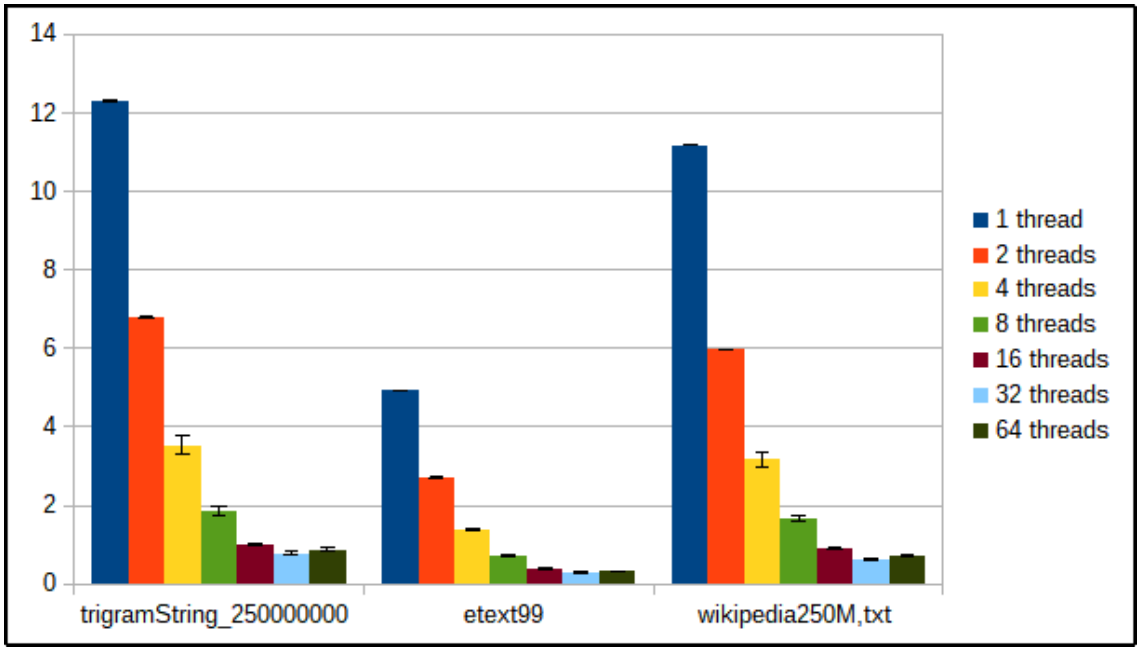


Figure A.106: Execution times (in minutes) of Word Counts ran on AMD 32-64 using LCWS with WShare (**tit-for-tat**).

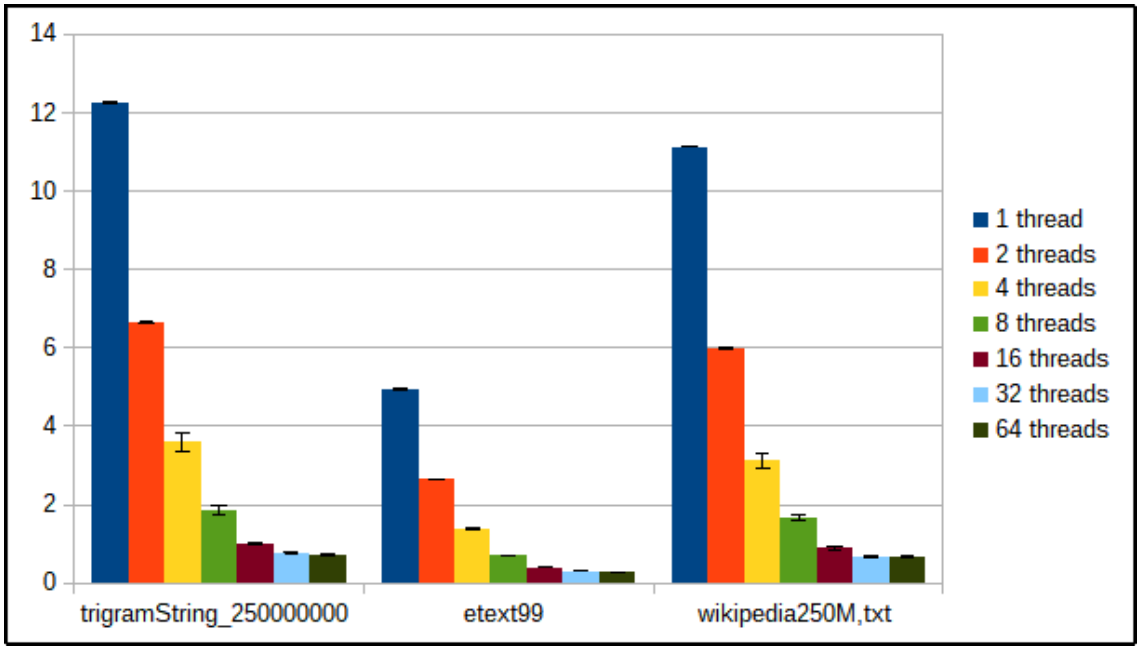


Figure A.107: Execution times (in minutes) of Word Counts ran on AMD 32-64 using LCWS with WShare (**continuous tit-for-tat**).

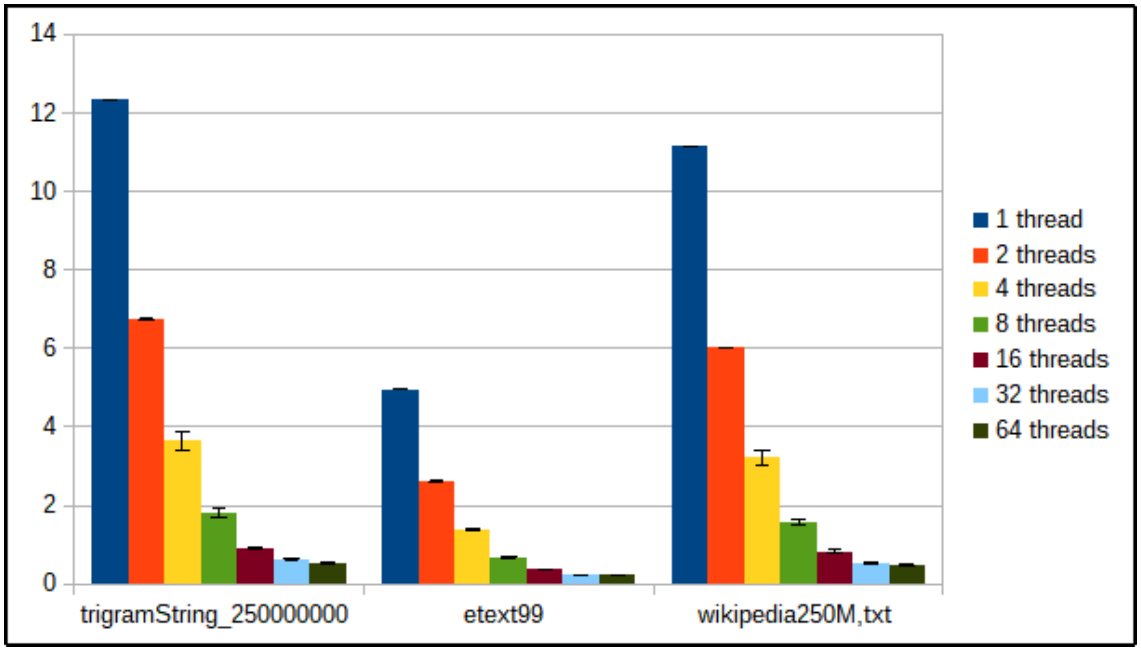


Figure A.108: Execution times (in minutes) of Word Counts ran on AMD 32-64 using SBLCWS with WShare (**first version**).

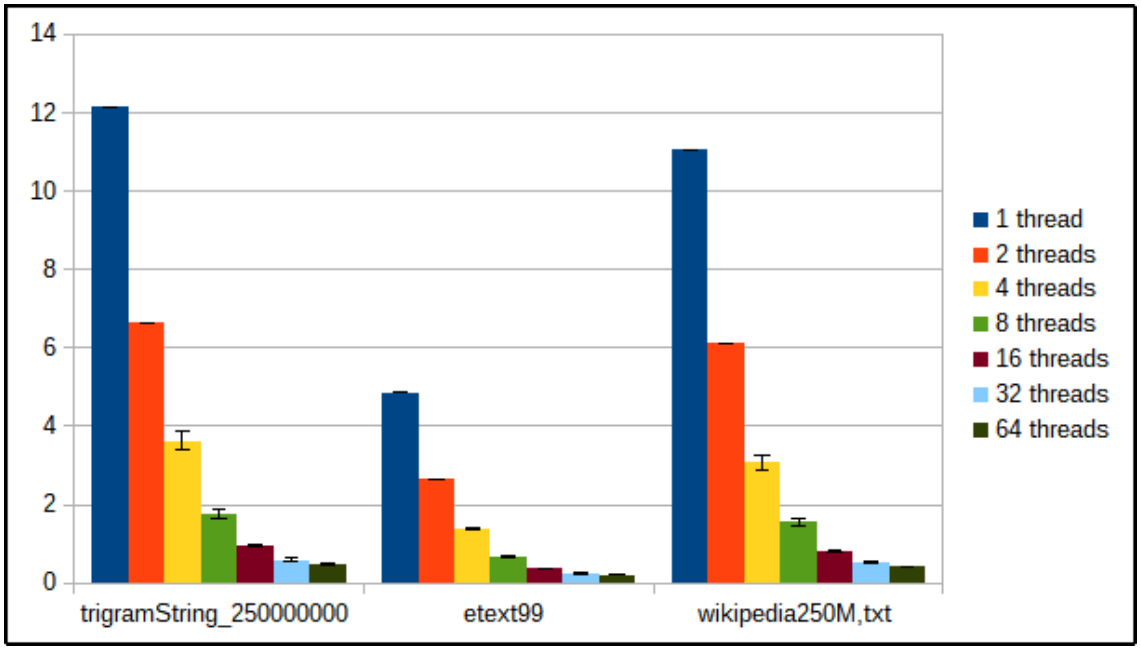


Figure A.109: Execution times (in minutes) of Word Counts ran on AMD 32-64 using SBLCWS with WShare (**second version**).

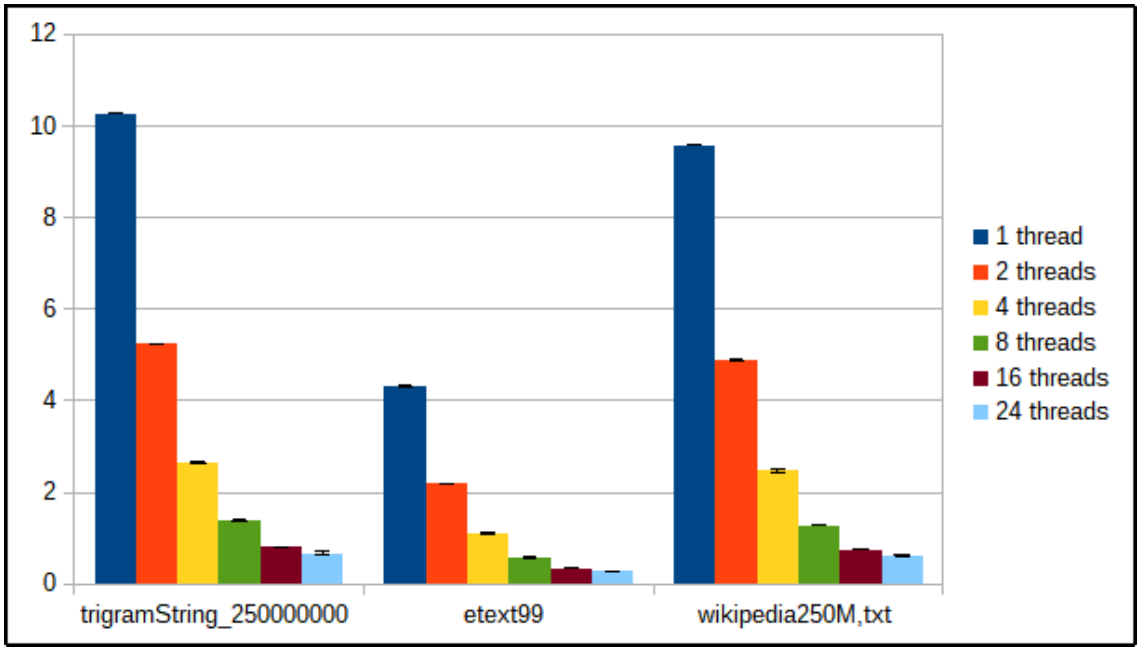


Figure A.110: Execution times (in minutes) of Word Counts ran on Intel 16-16 using WSteal

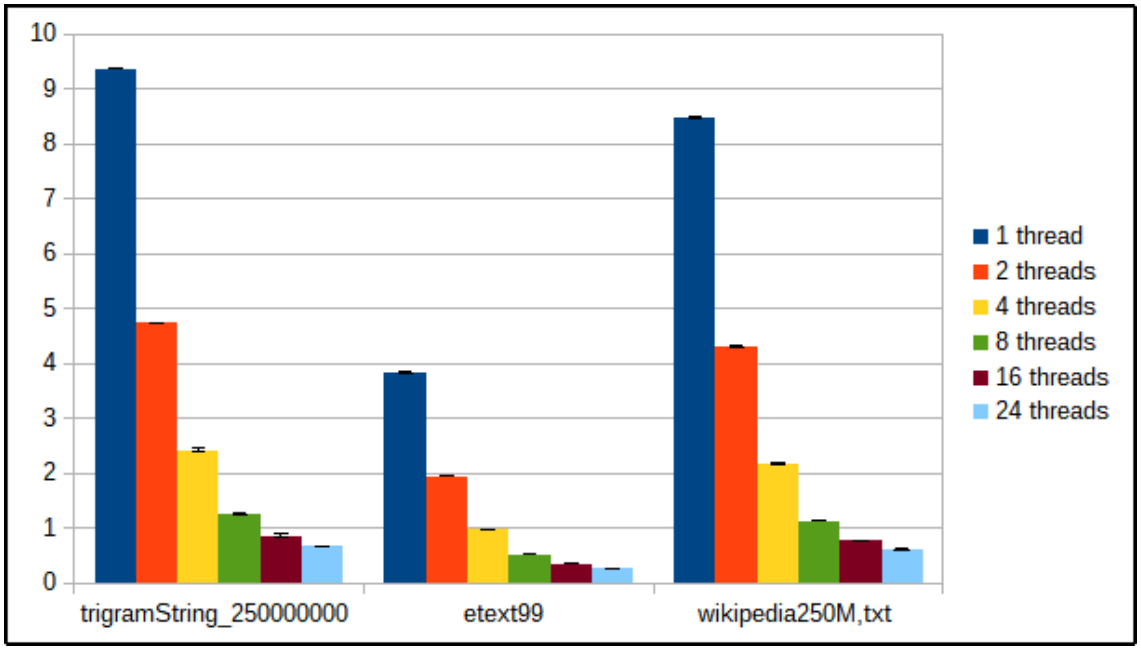


Figure A.111: Execution times (in minutes) of Word Counts ran on Intel 16-16 using LCWS

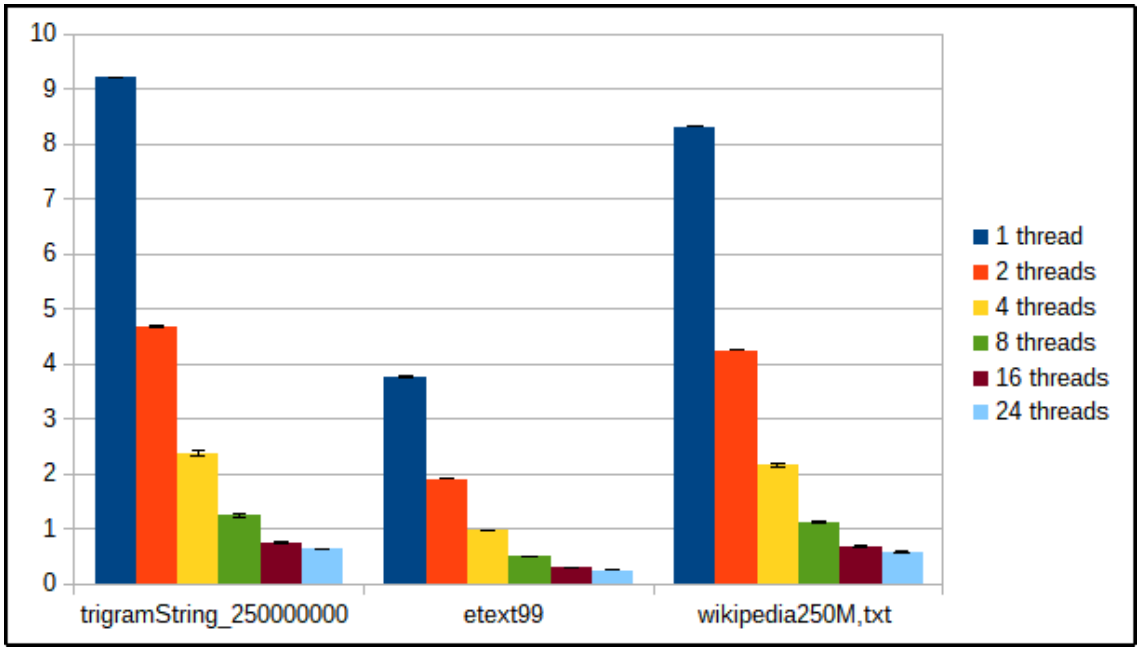


Figure A.112: Execution times (in minutes) of Word Counts ran on Intel 16-16 using SBLCWS (**first version**)

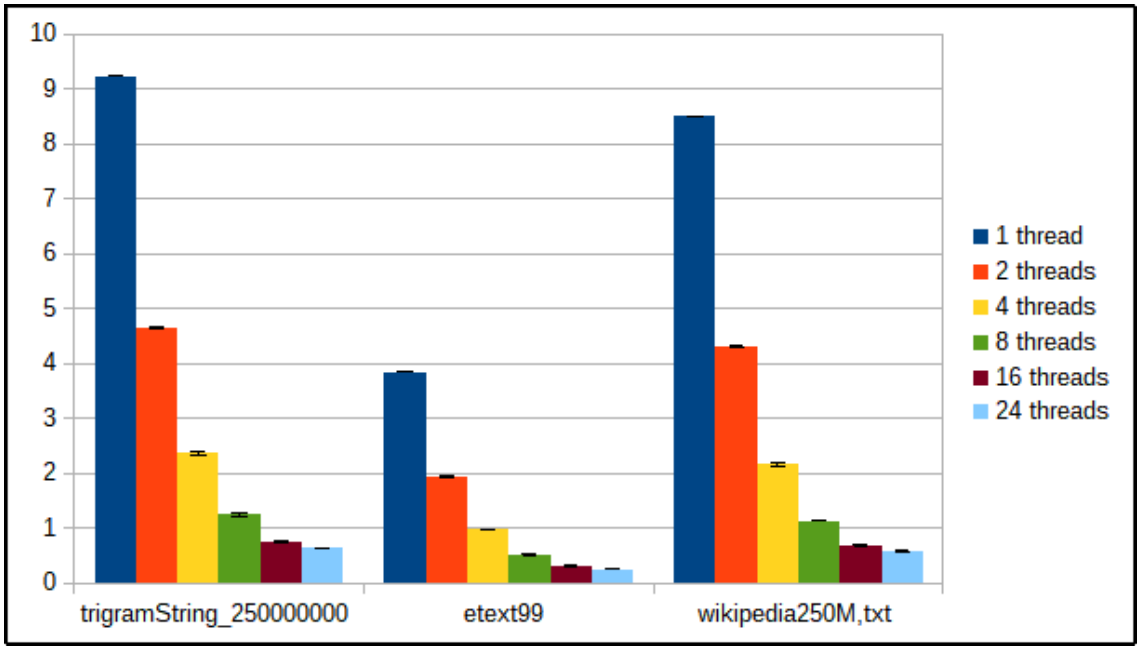


Figure A.113: Execution times (in minutes) of Word Counts ran on Intel 16-16 using SBLCWS (**second version**)

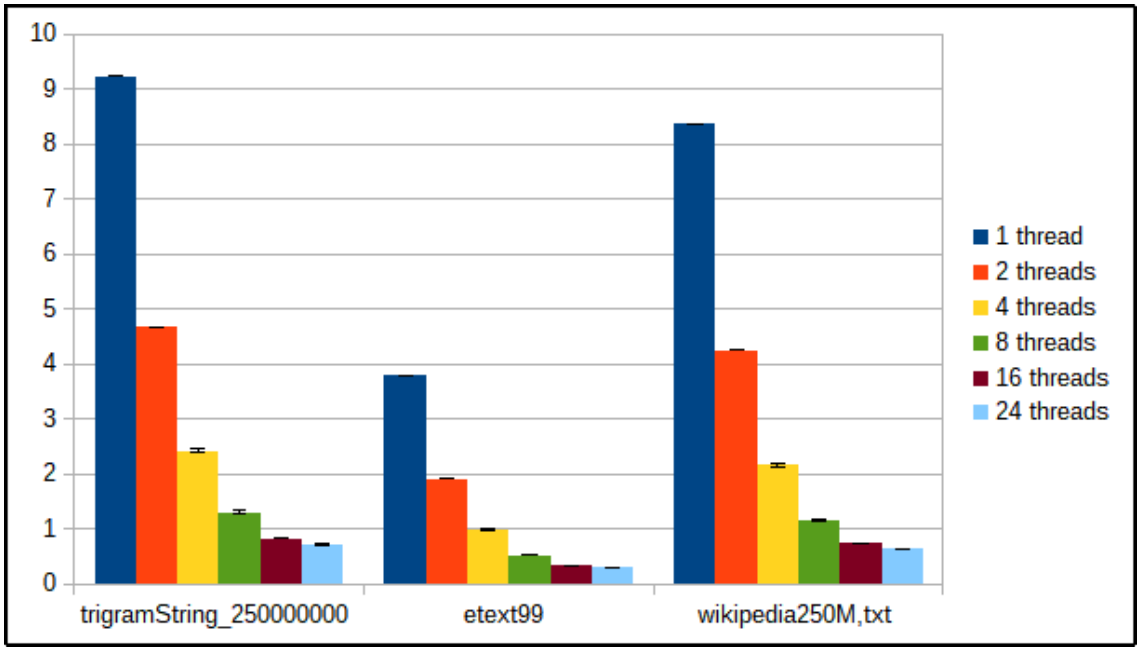


Figure A.114: Execution times (in minutes) of Word Counts ran on Intel 16-16 using LCWS with WShare (**tit-for-tat**).

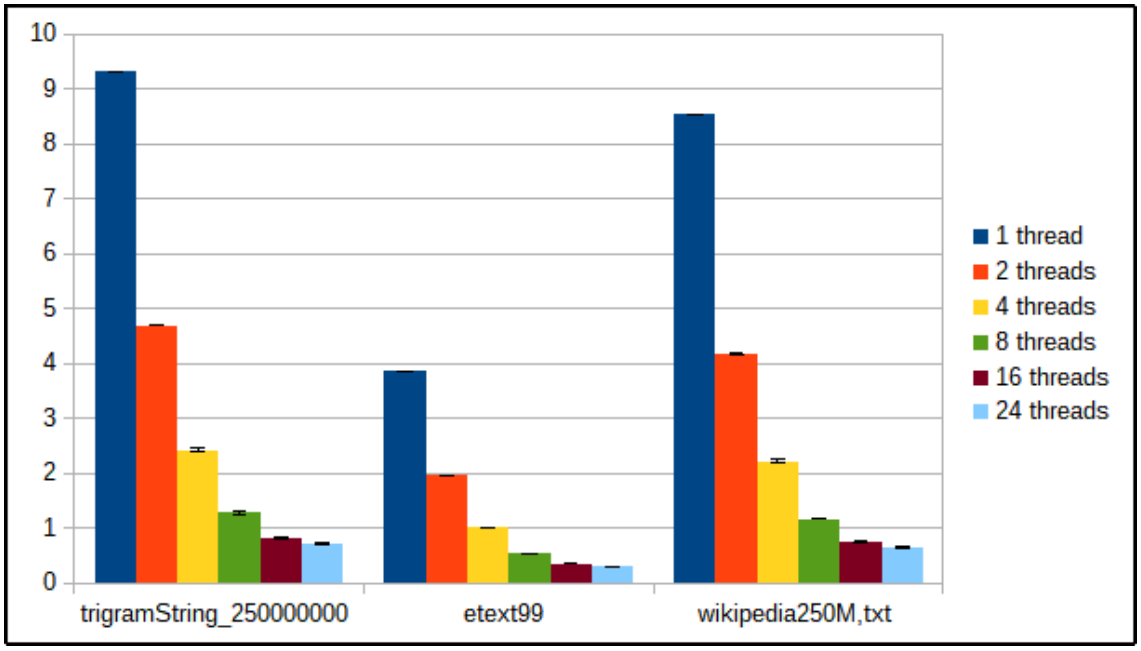


Figure A.115: Execution times (in minutes) of Word Counts ran on Intel 16-16 using LCWS with WShare (**continuous tit-for-tat**).

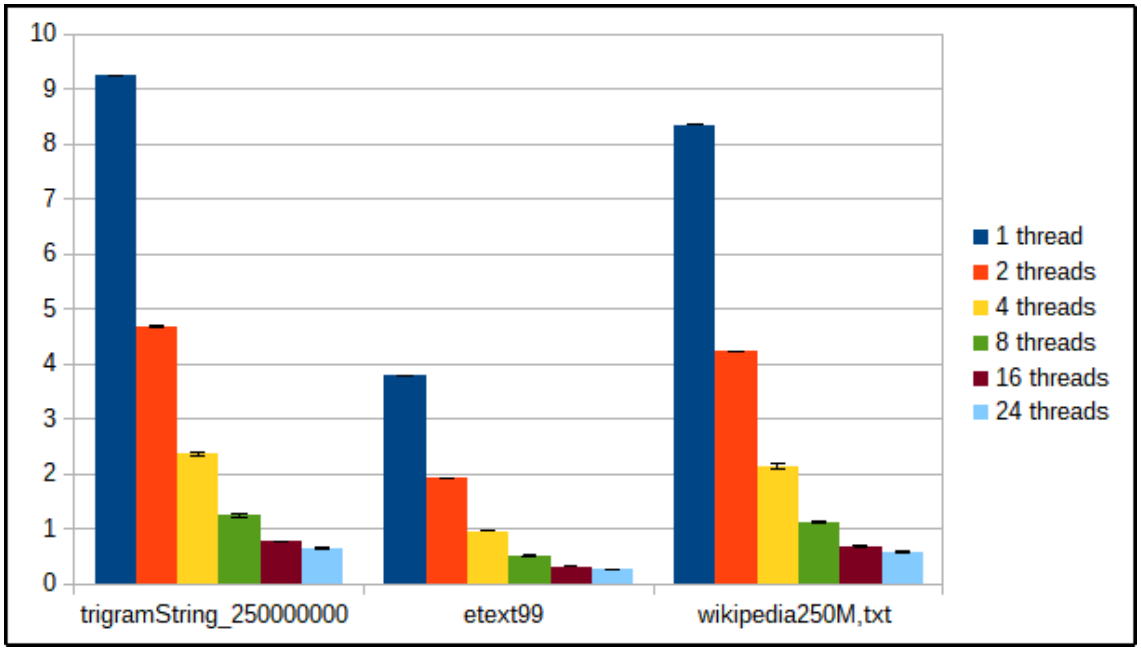


Figure A.116: Execution times (in minutes) of Word Counts ran on Intel 16-16 using SBLCWS with WShare (**first version**).

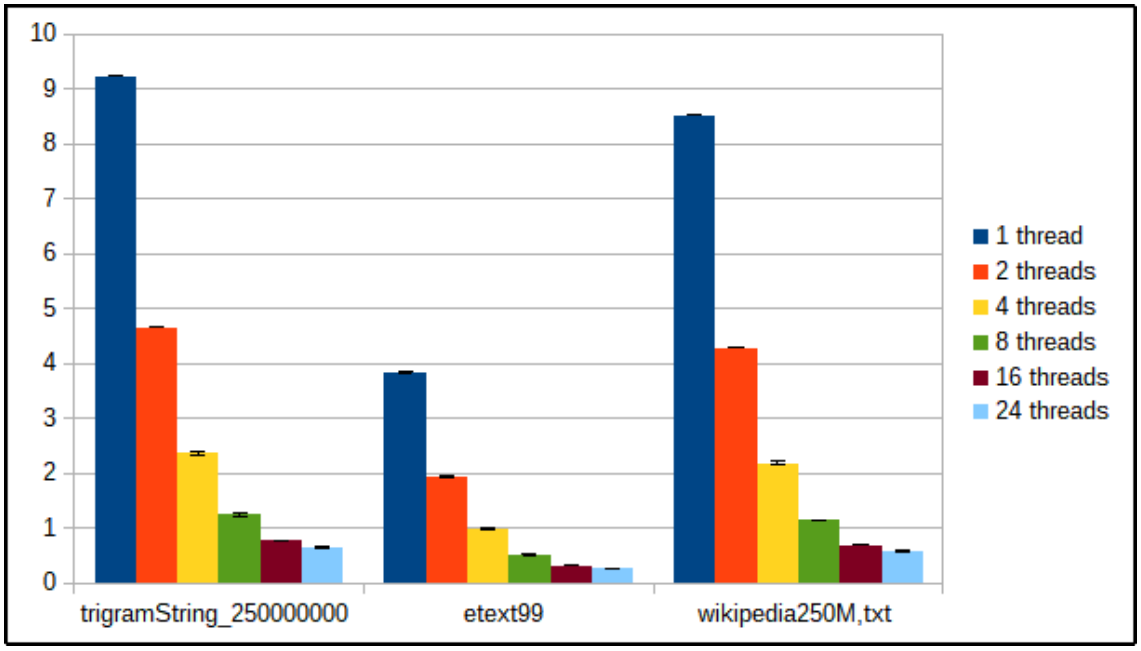


Figure A.117: Execution times (in minutes) of Word Counts ran on Intel 16-16 using SBLCWS with WShare (**second version**).

A.6 Inverted Index

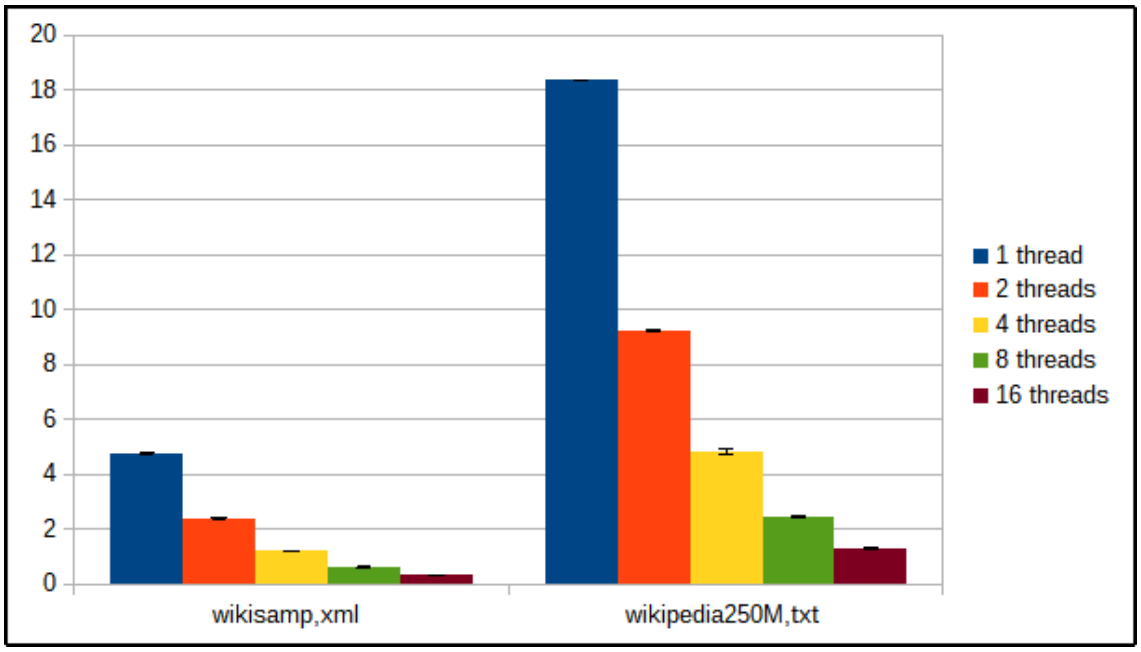


Figure A.118: Execution times (in minutes) of Inverted Index ran on Intel 12-24 using WSteal

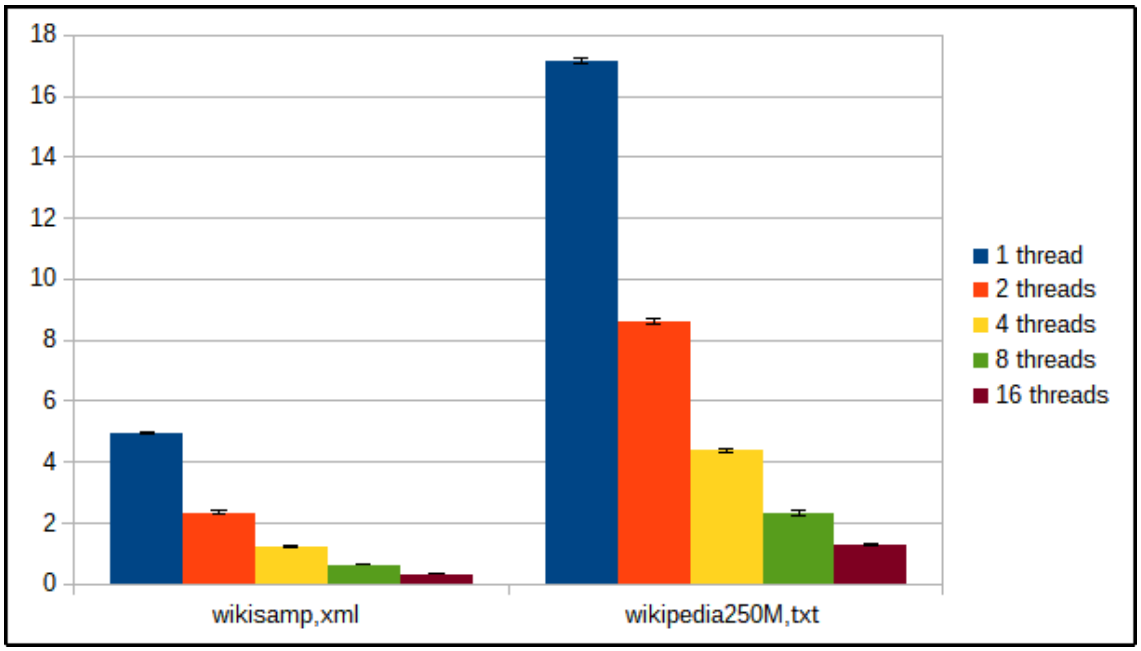


Figure A.119: Execution times (in minutes) of Inverted Index ran on Intel 12-24 using LCWS

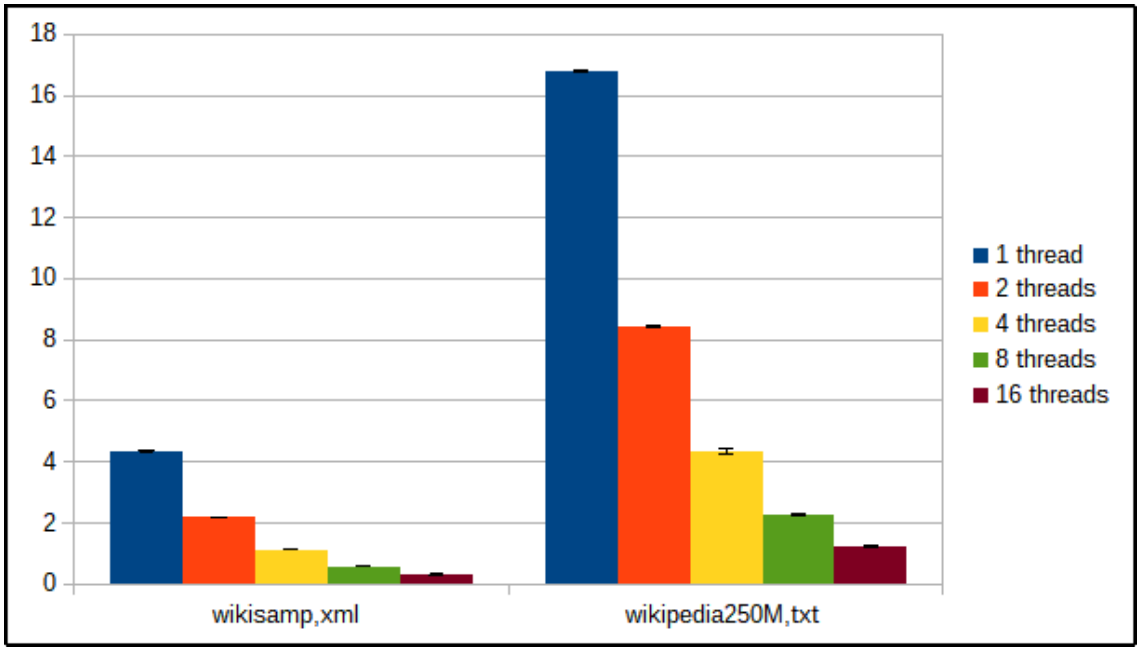


Figure A.120: Execution times (in minutes) of Inverted Index ran on Intel 12-24 using SBLCWS (**first version**)

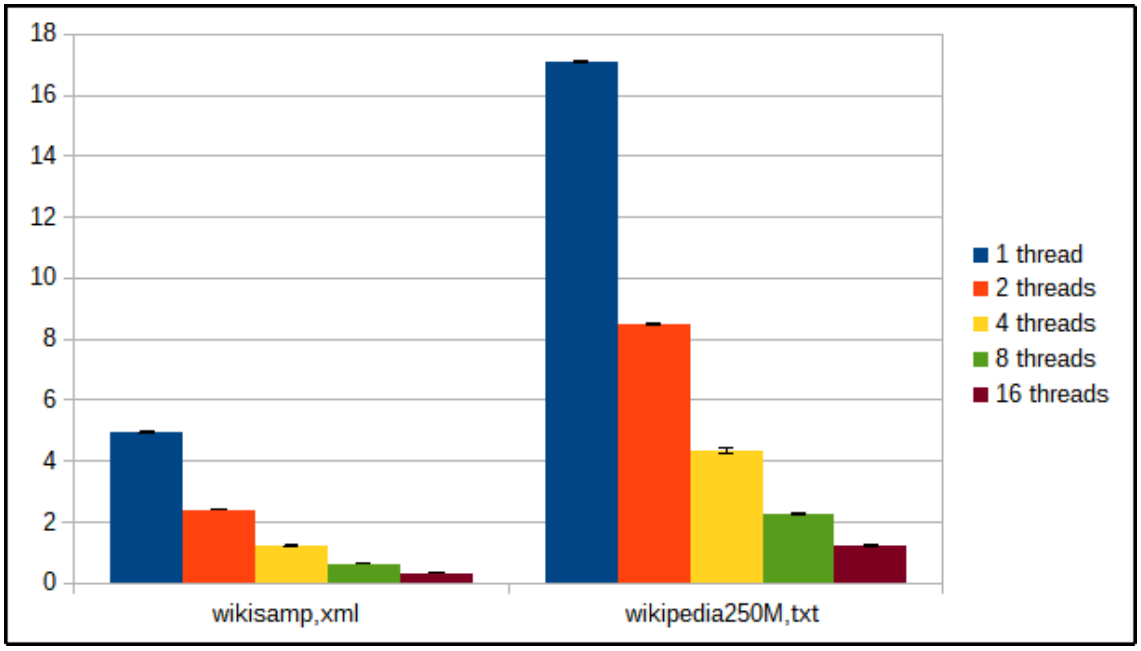


Figure A.121: Execution times (in minutes) of Inverted Index ran on Intel 12-24 using SBLCWS (**second version**)

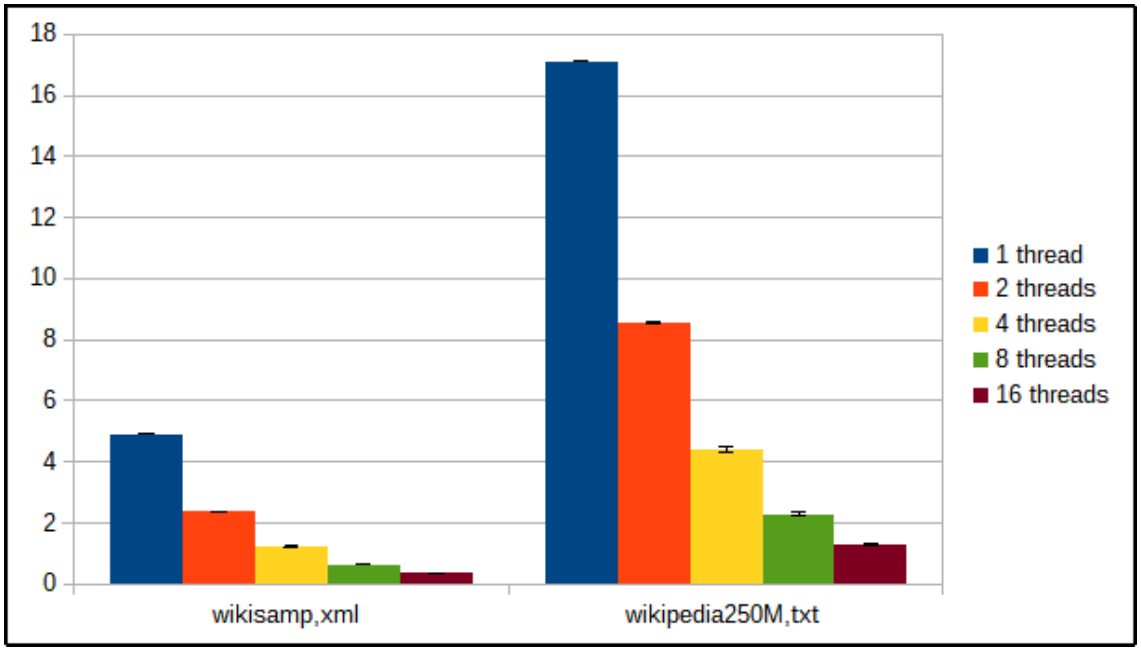


Figure A.122: Execution times (in minutes) of Inverted Index ran on Intel 12-24 using LCWS with WShare (**tit-for-tat**).

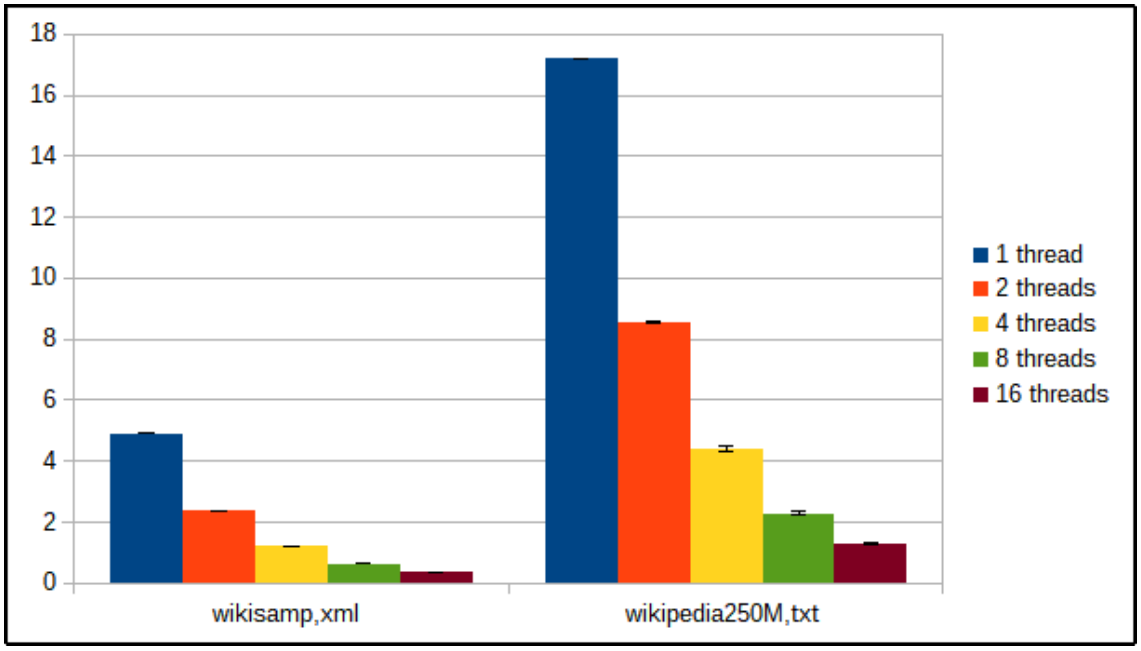


Figure A.123: Execution times (in minutes) of Inverted Index ran on Intel 12-24 using LCWS with WShare (**continuous tit-for-tat**).

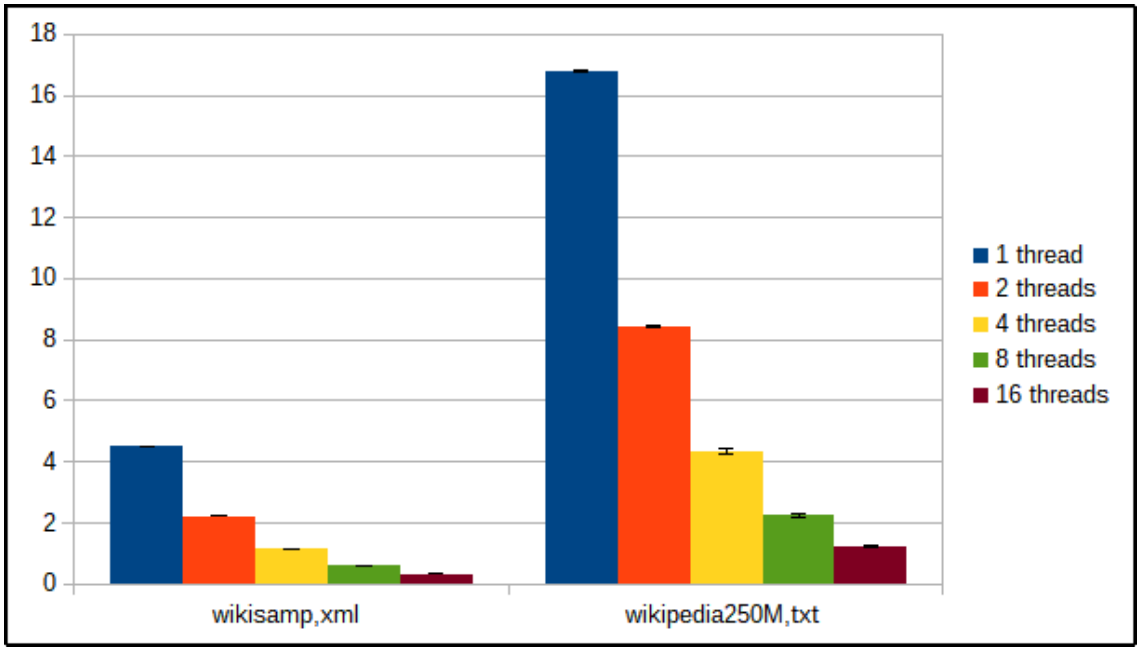


Figure A.124: Execution times (in minutes) of Inverted Index ran on Intel 12-24 using SBLCWS with WShare (**first version**).

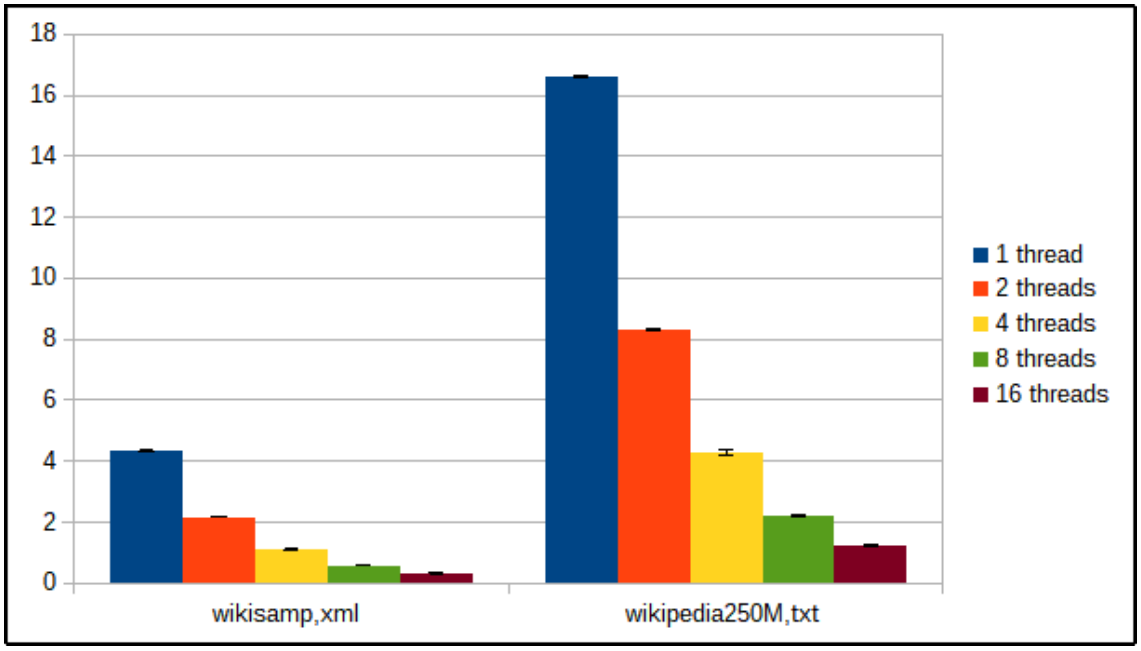


Figure A.125: Execution times (in minutes) of Inverted Index ran on Intel 12-24 using SBLCWS with WShare (**second version**).

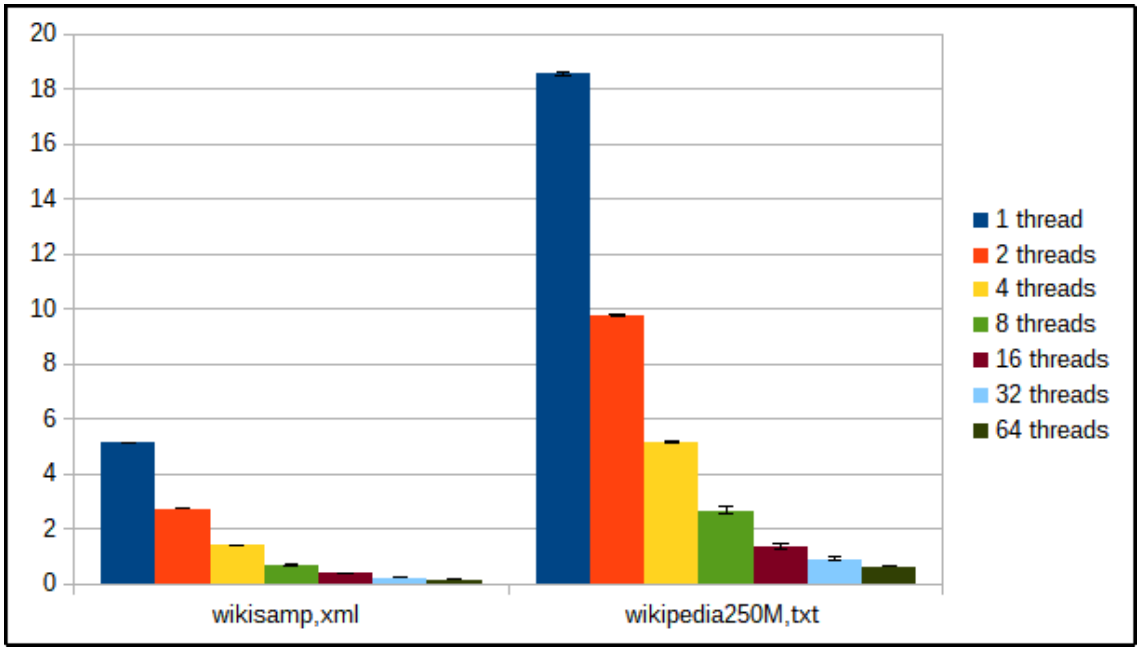


Figure A.126: Execution times (in minutes) of Inverted Index ran on AMD 32-64 using WSteal

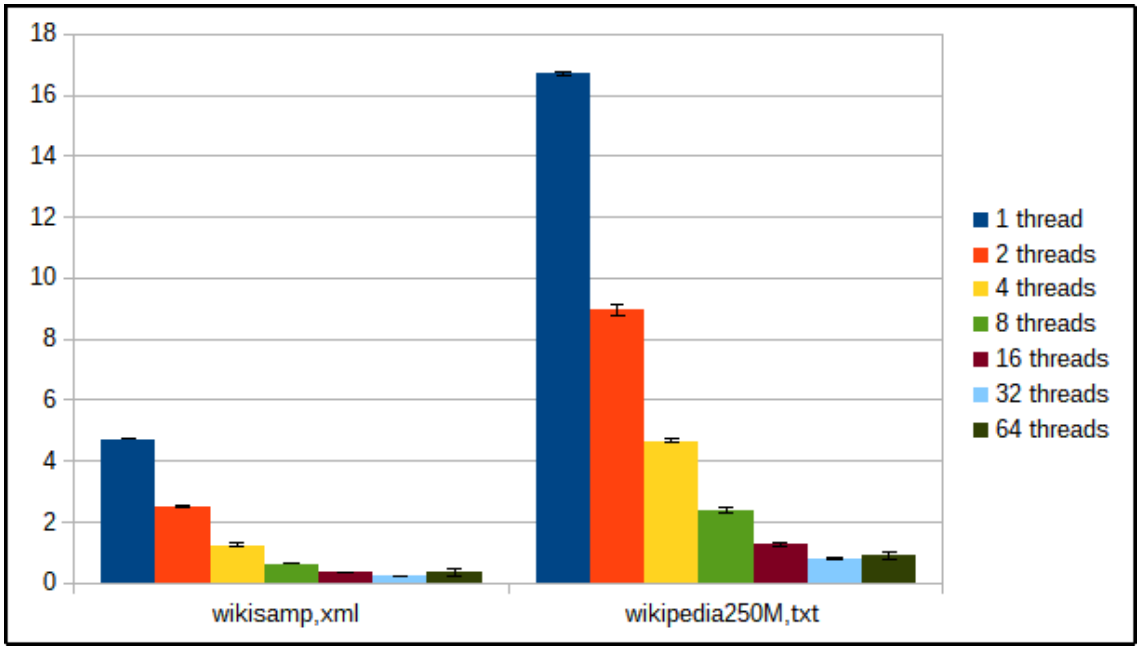


Figure A.127: Execution times (in minutes) of Inverted Index ran on AMD 32-64 using LCWS

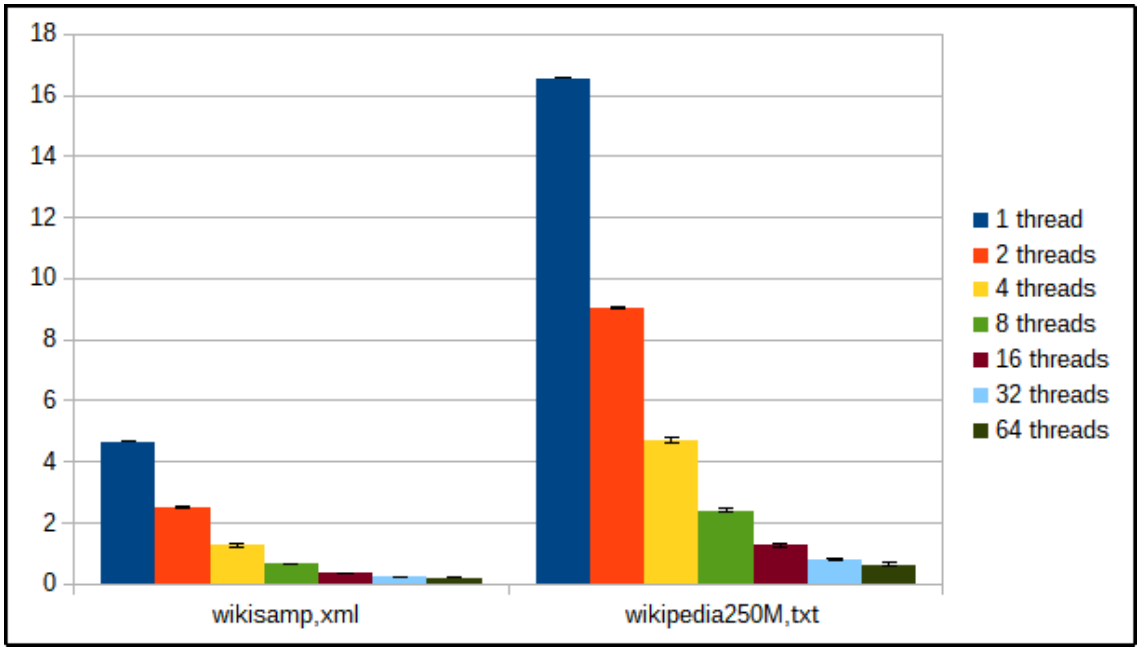


Figure A.128: Execution times (in minutes) of Inverted Index ran on AMD 32-64 using SBLCWS (**first version**)

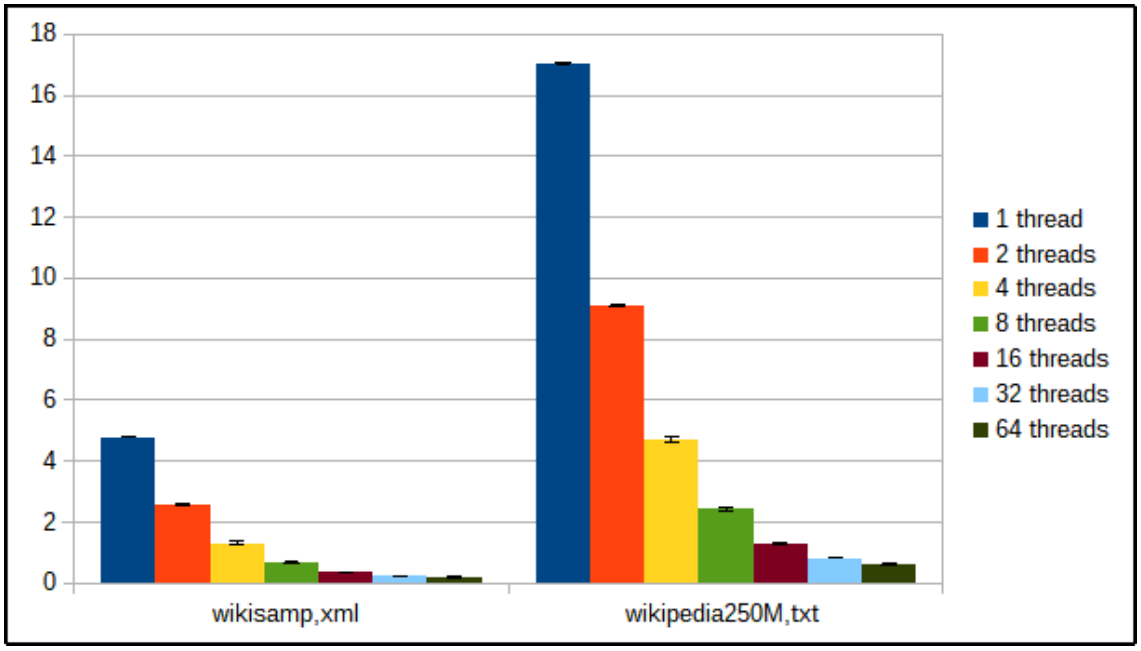


Figure A.129: Execution times (in minutes) of Inverted Index ran on AMD 32-64 using SBLCWS (**second version**)

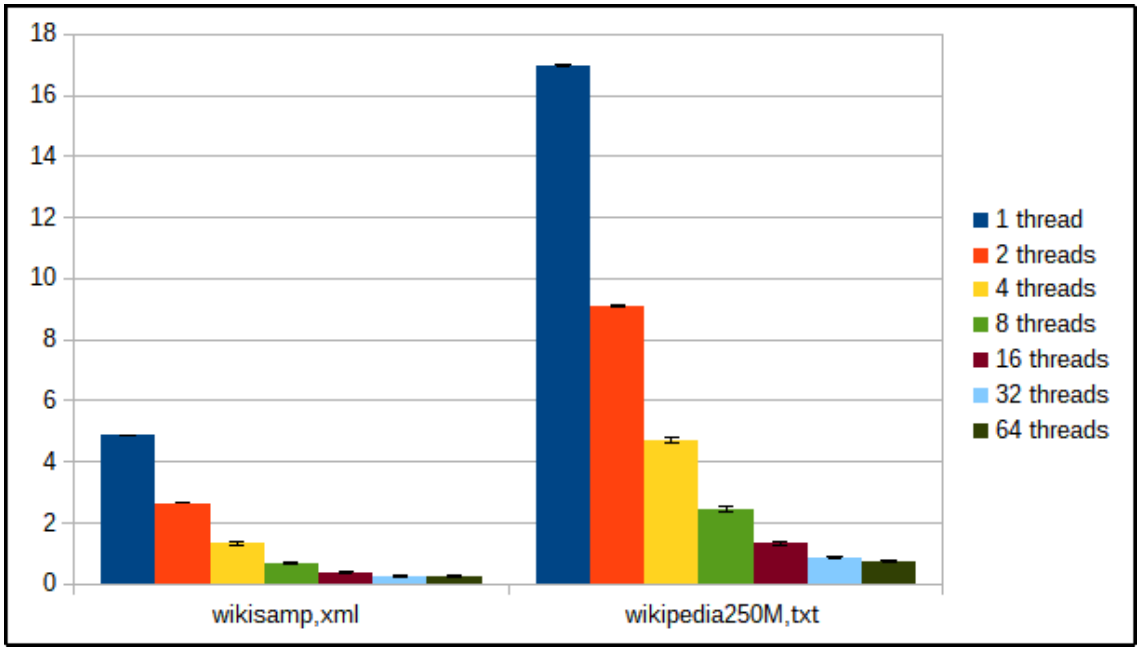


Figure A.130: Execution times (in minutes) of Inverted Index ran on AMD 32-64 using LCWS with WShare (**tit-for-tat**).

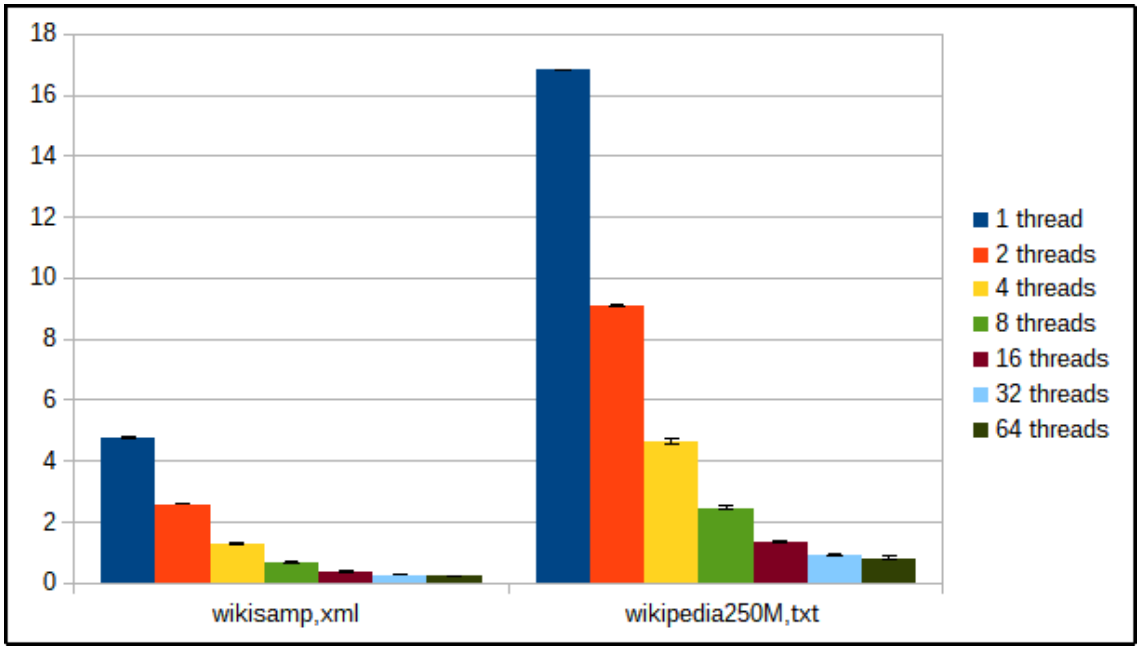


Figure A.131: Execution times (in minutes) of Inverted Index ran on AMD 32-64 using LCWS with WShare (**continuous tit-for-tat**).

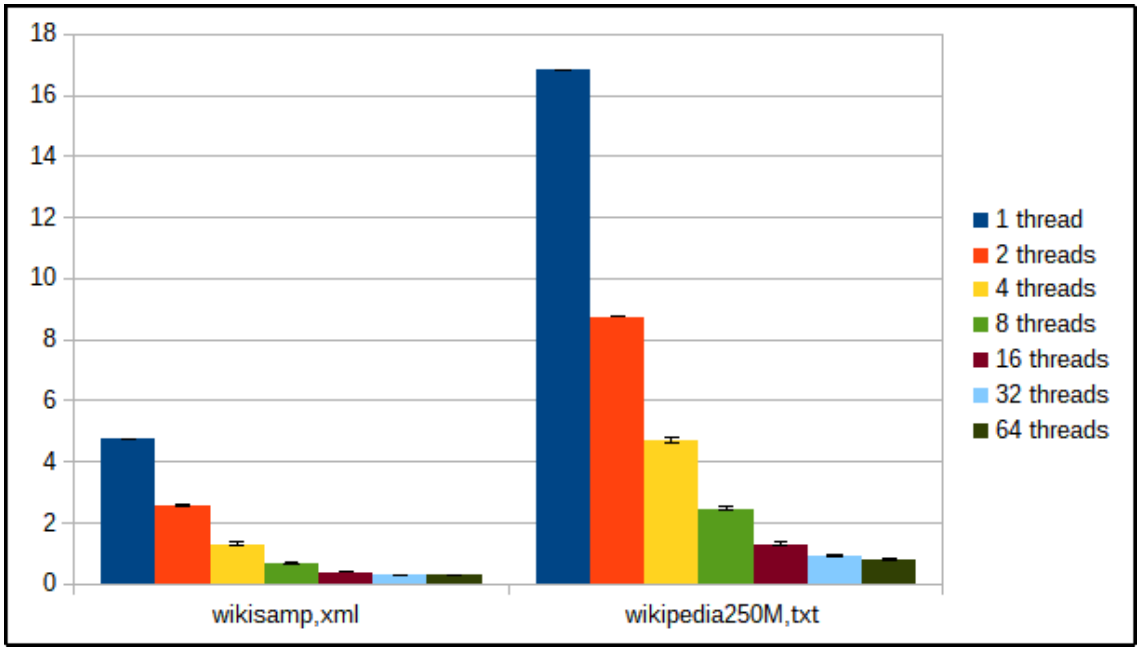


Figure A.132: Execution times (in minutes) of Inverted Index ran on AMD 32-64 using SBLCWS with WShare (**first version**).

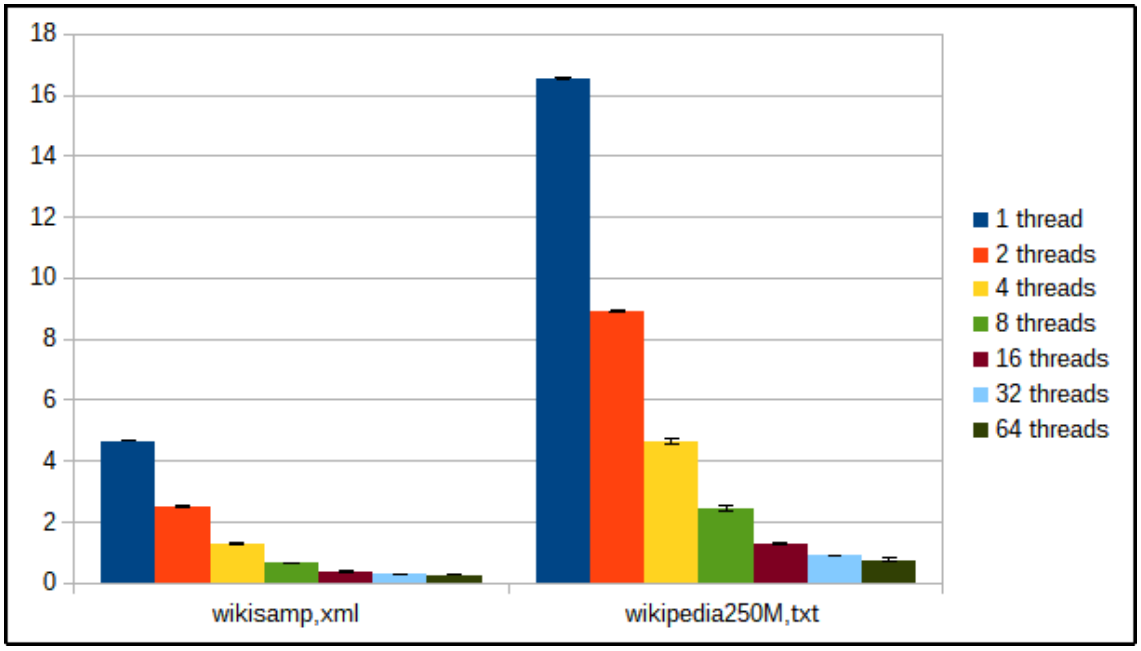


Figure A.133: Execution times (in minutes) of Inverted Index ran on AMD 32-64 using SBLCWS with WShare (**second version**).

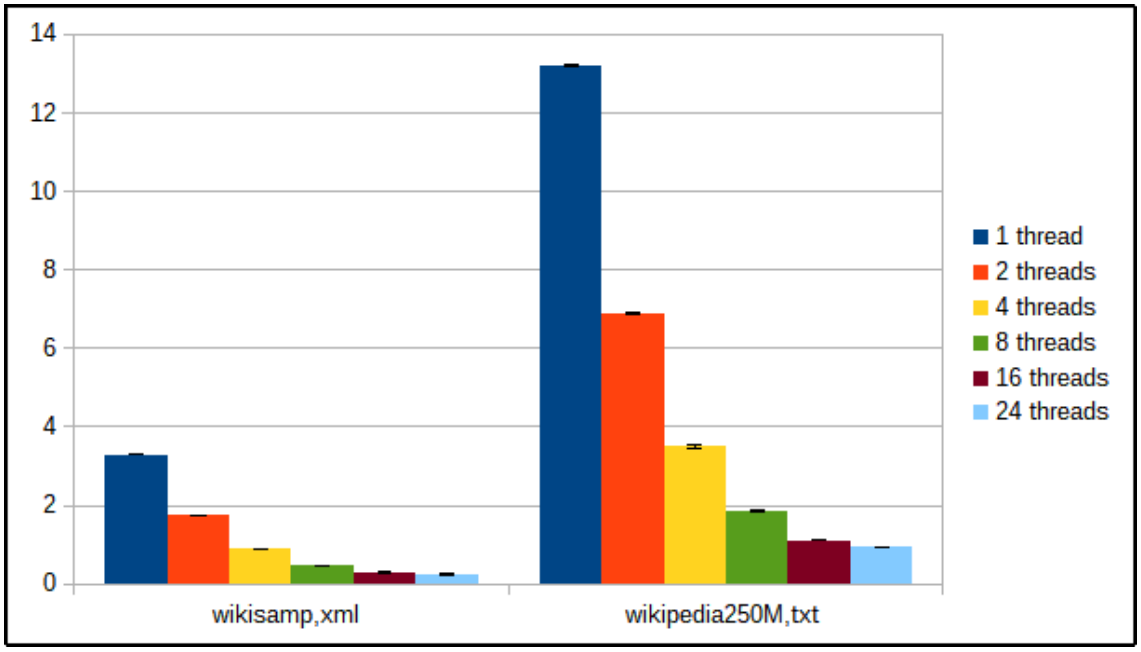


Figure A.134: Execution times (in minutes) of Inverted Index ran on Intel 16-16 using WSteal

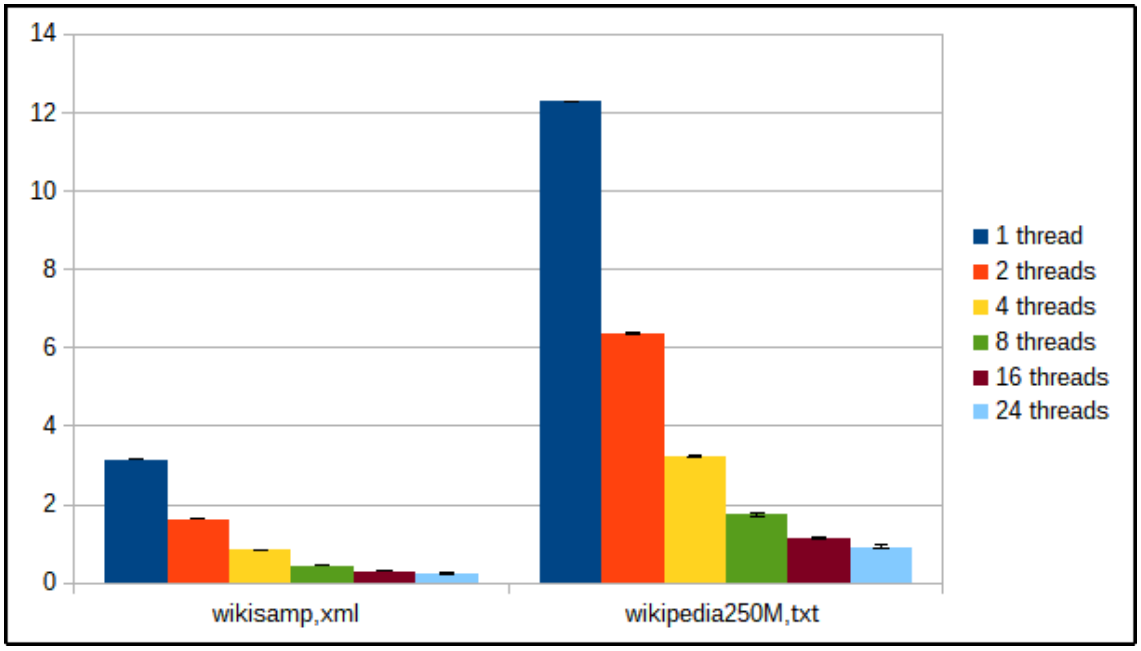


Figure A.135: Execution times (in minutes) of Inverted Index ran on Intel 16-16 using LCWS

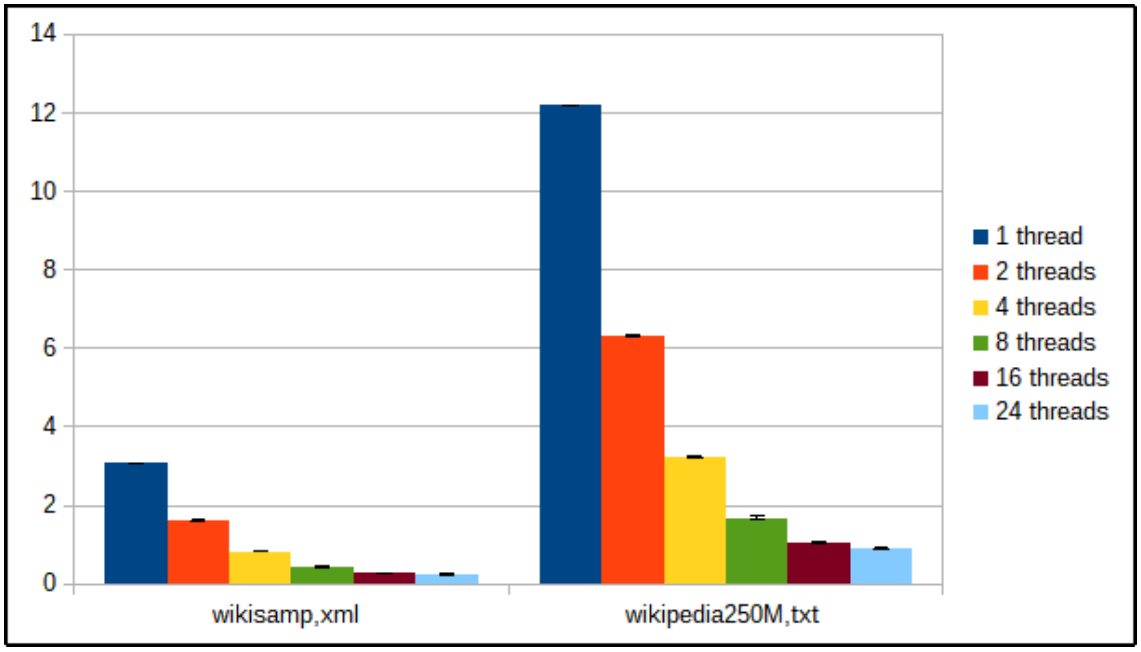


Figure A.136: Execution times (in minutes) of Inverted Index ran on Intel 16-16 using SBLCWS (**first version**)

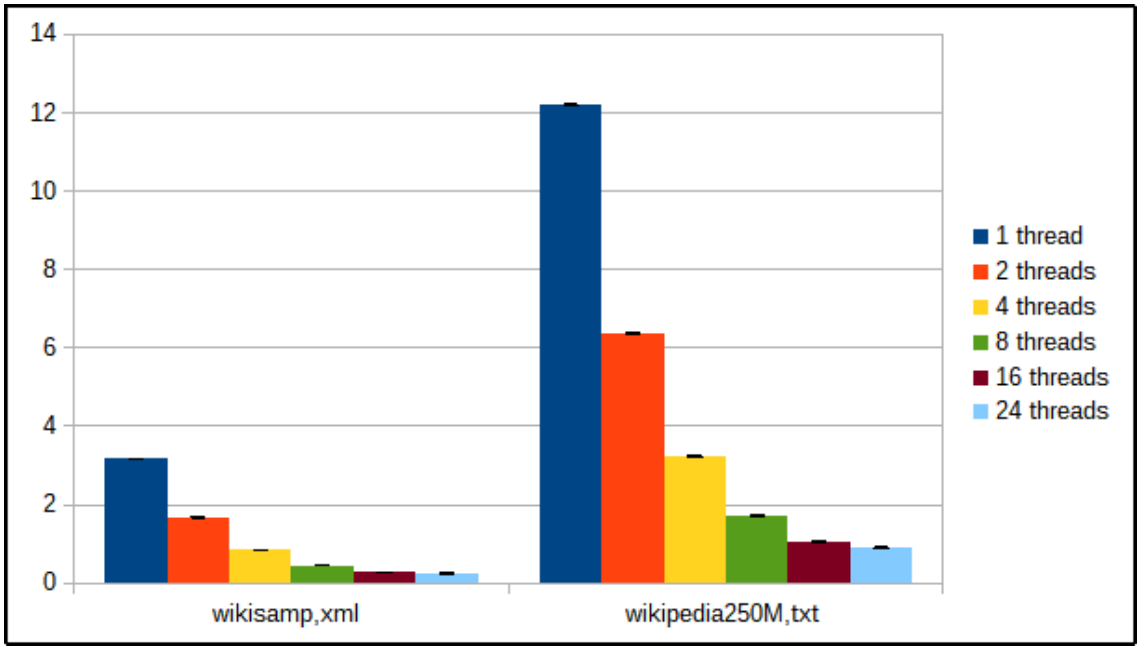


Figure A.137: Execution times (in minutes) of Inverted Index ran on Intel 16-16 using SBLCWS (**second version**)

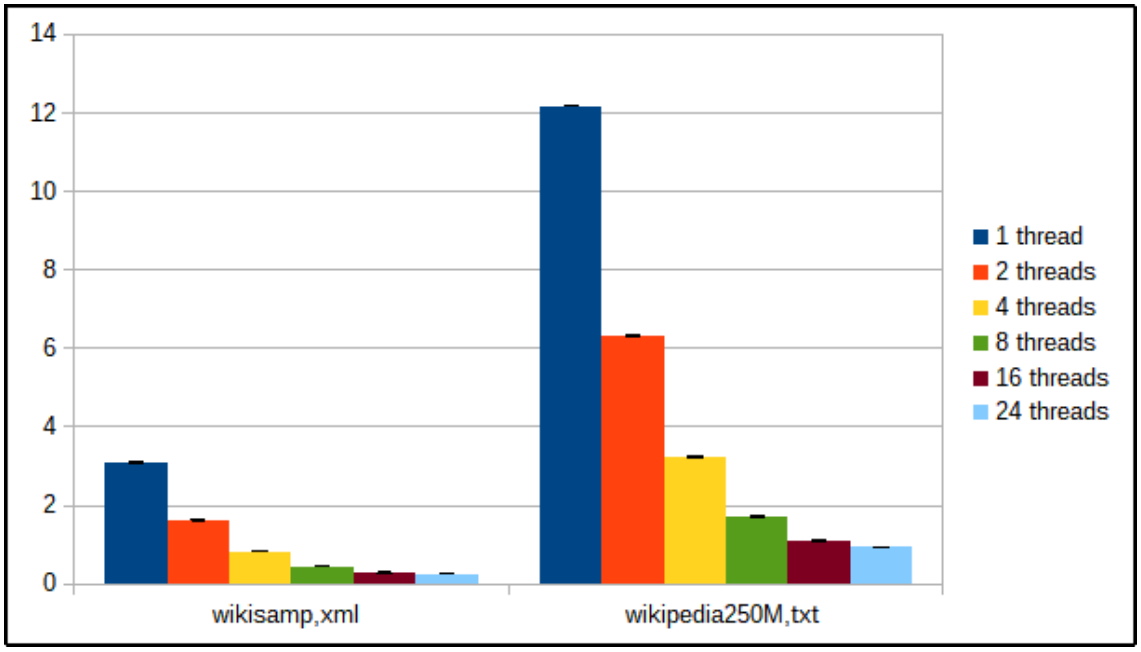


Figure A.138: Execution times (in minutes) of Inverted Index ran on Intel 16-16 using LCWS with WShare (**tit-for-tat**).

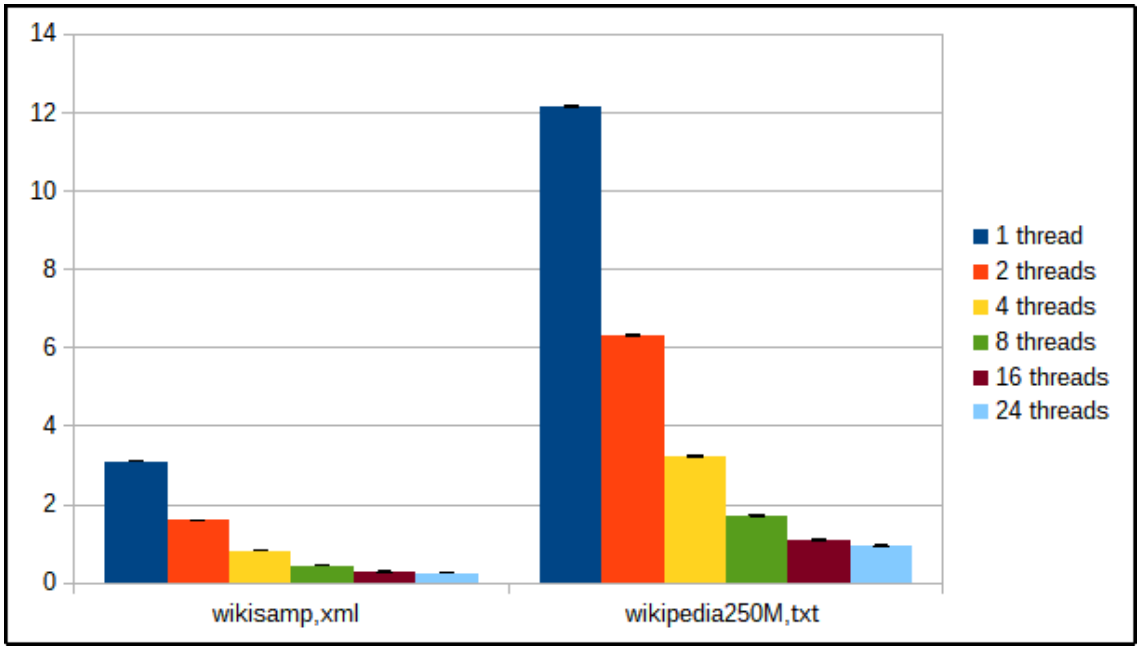


Figure A.139: Execution times (in minutes) of Inverted Index ran on Intel 16-16 using LCWS with WShare (**continuous tit-for-tat**).

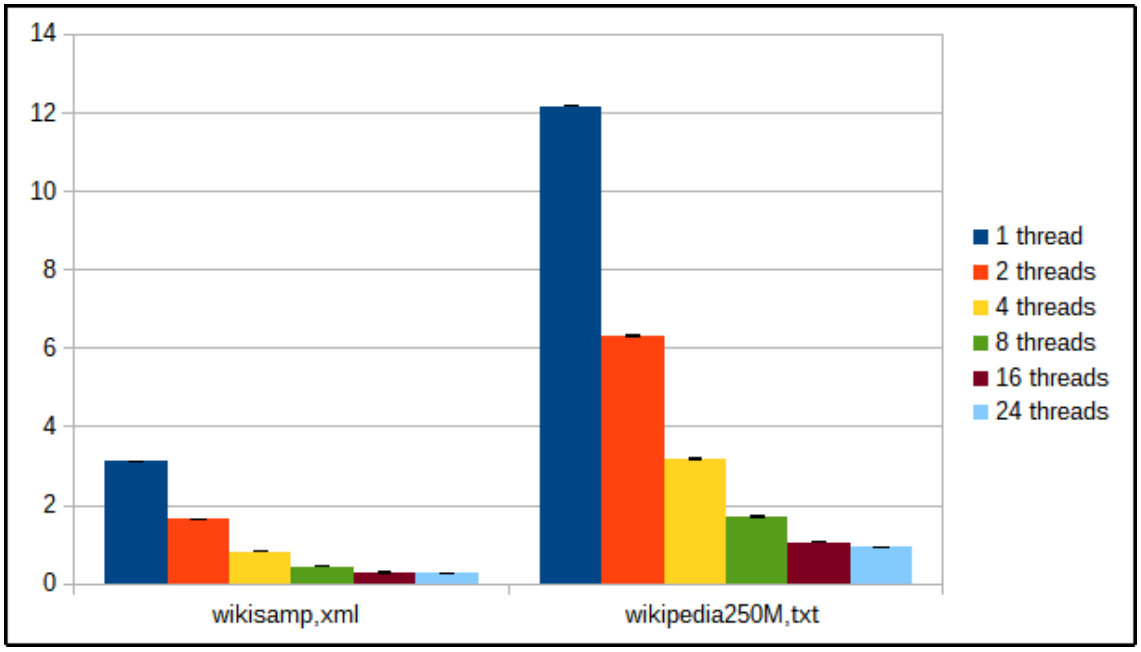


Figure A.140: Execution times (in minutes) of Inverted Index ran on Intel 16-16 using SBLCWS with WShare (**first version**).

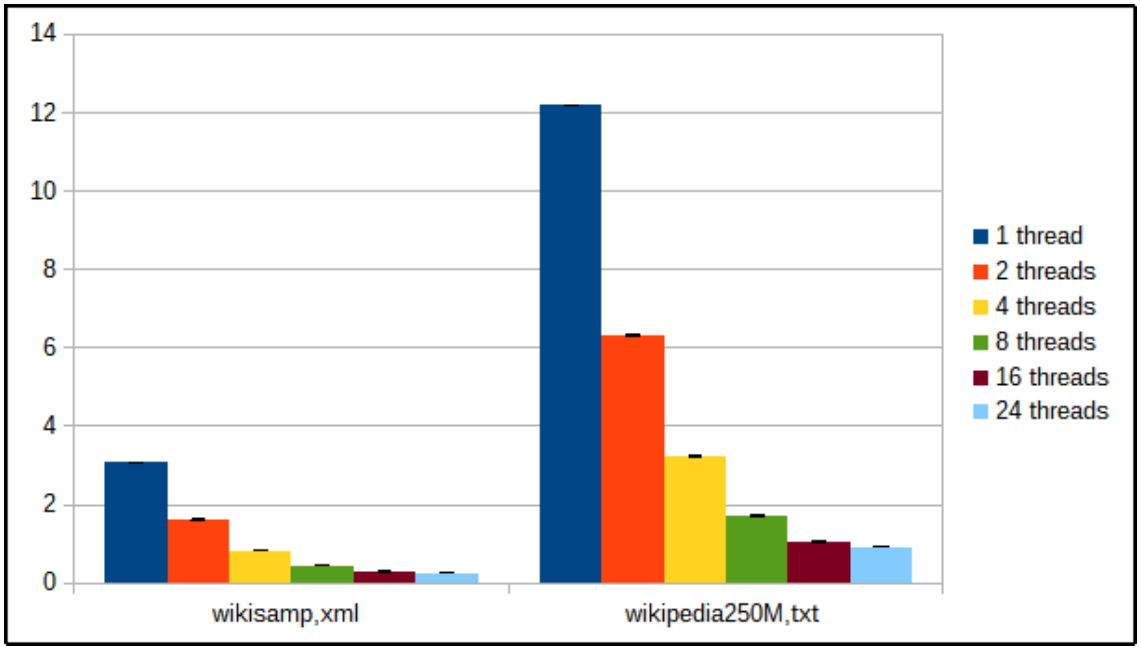


Figure A.141: Execution times (in minutes) of Inverted Index ran on Intel 16-16 using SBLCWS with WShare (**second version**).

A.7 Classify

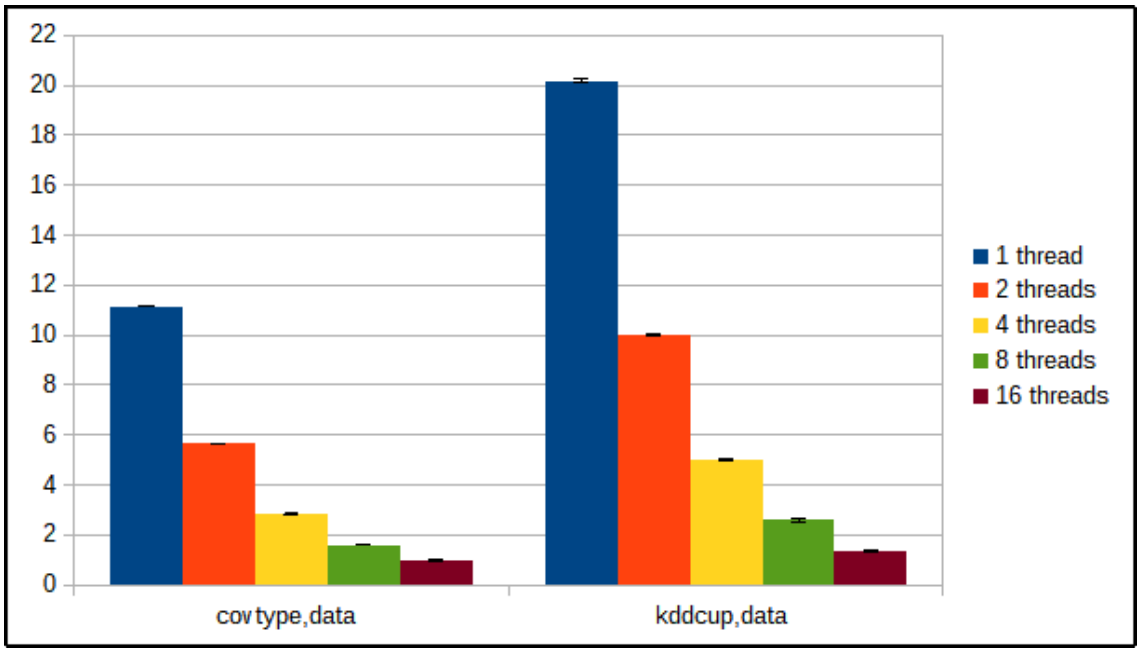


Figure A.142: Execution times (in minutes) of Classify ran on Intel 12-24 using WSteal

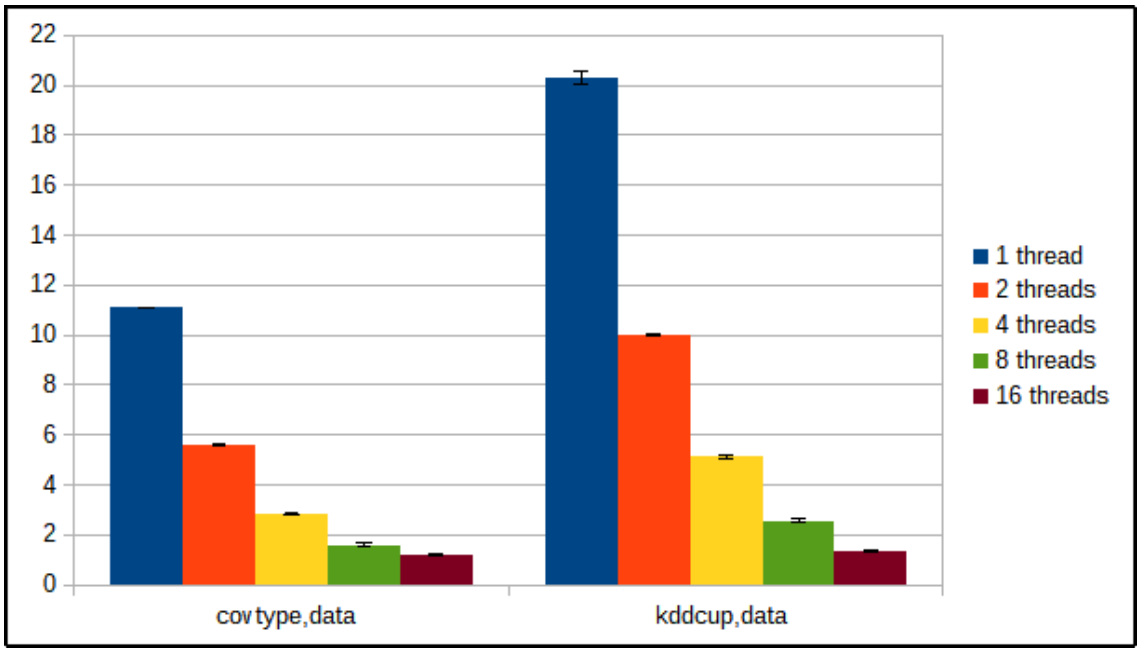


Figure A.143: Execution times (in minutes) of Classify ran on Intel 12-24 using LCWS

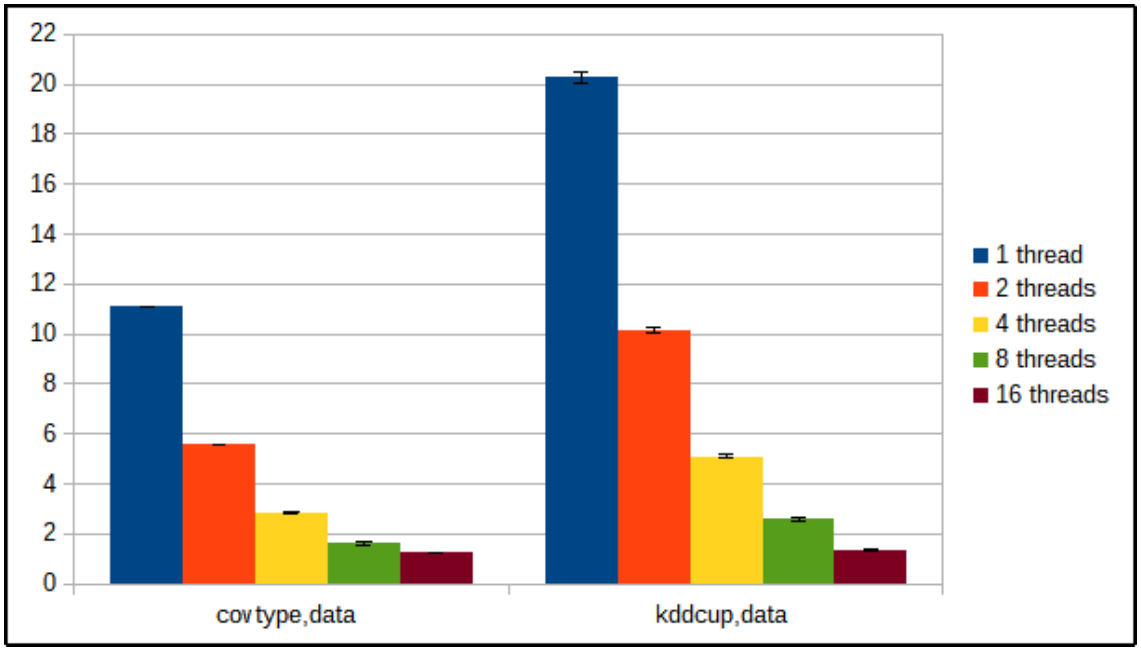


Figure A.144: Execution times (in minutes) of Classify ran on Intel 12-24 using SBLCWS (**first version**)

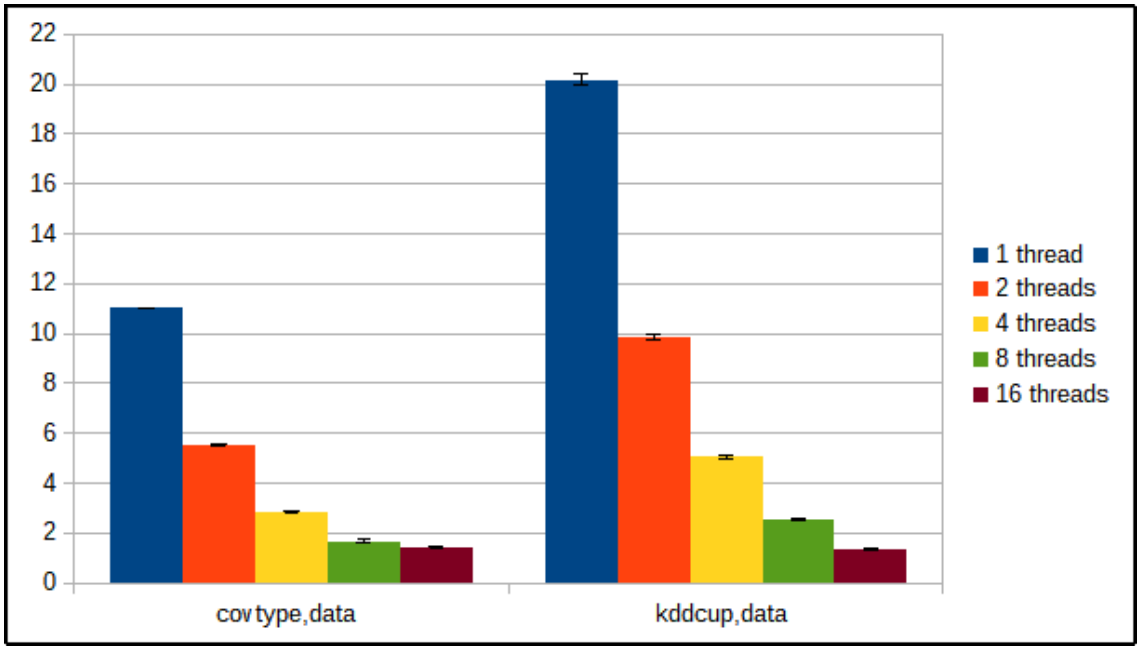


Figure A.145: Execution times (in minutes) of Classify ran on Intel 12-24 using SBLCWS (**second version**)

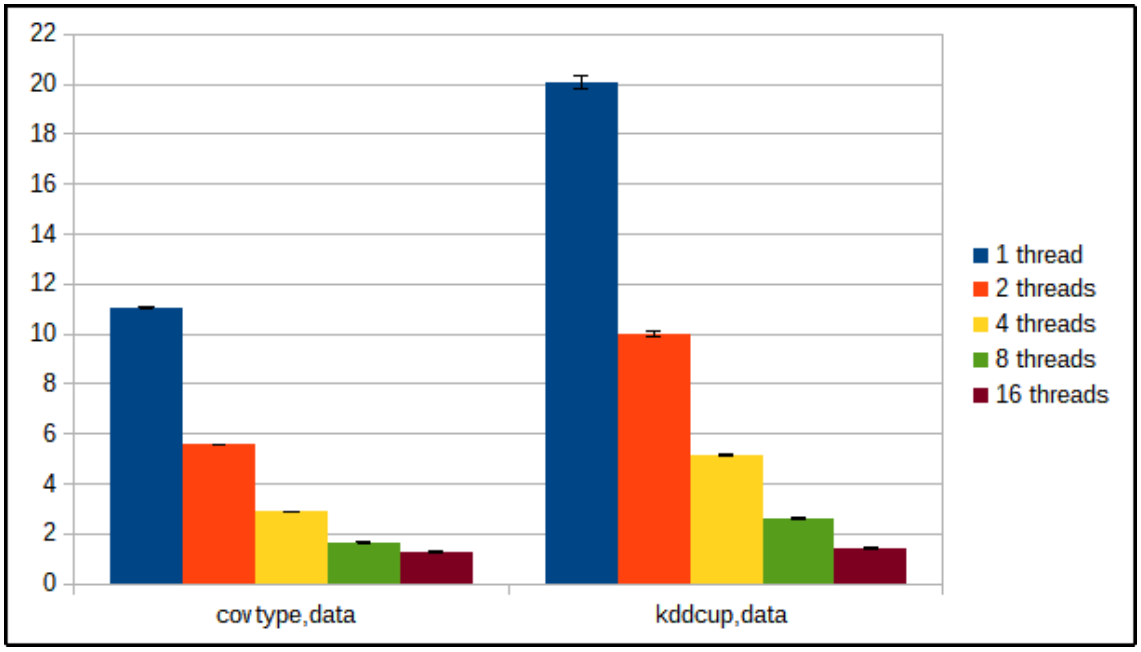


Figure A.146: Execution times (in minutes) of Classify ran on Intel 12-24 using LCWS with WShare (**tit-for-tat**).

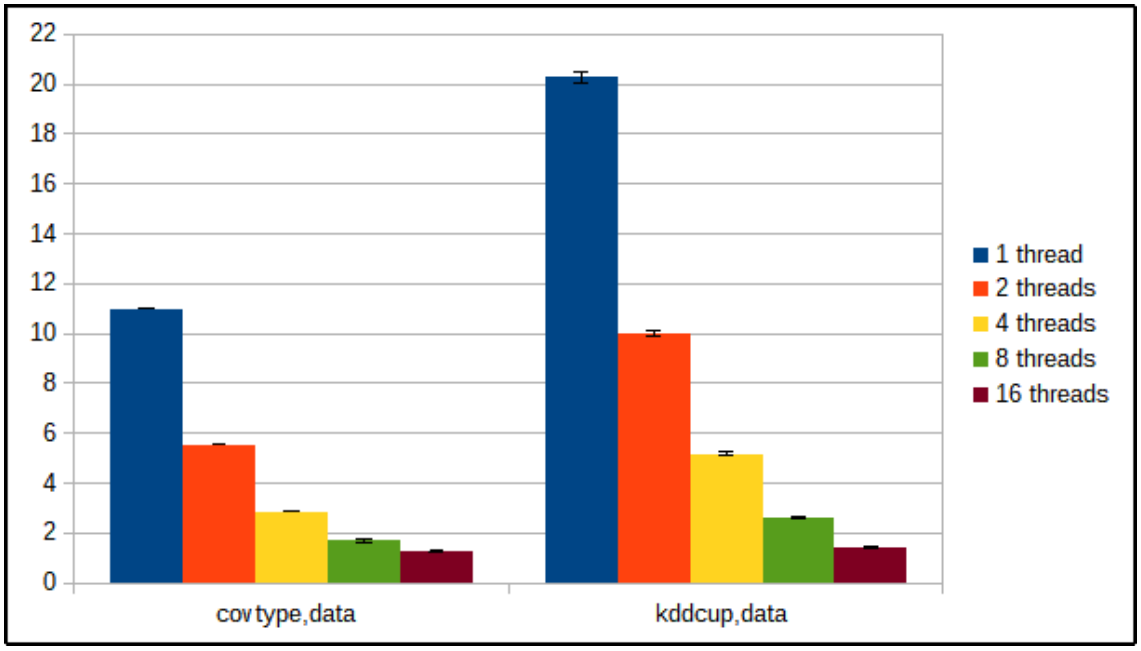


Figure A.147: Execution times (in minutes) of Classify ran on Intel 12-24 using LCWS with WShare (**continuous tit-for-tat**).

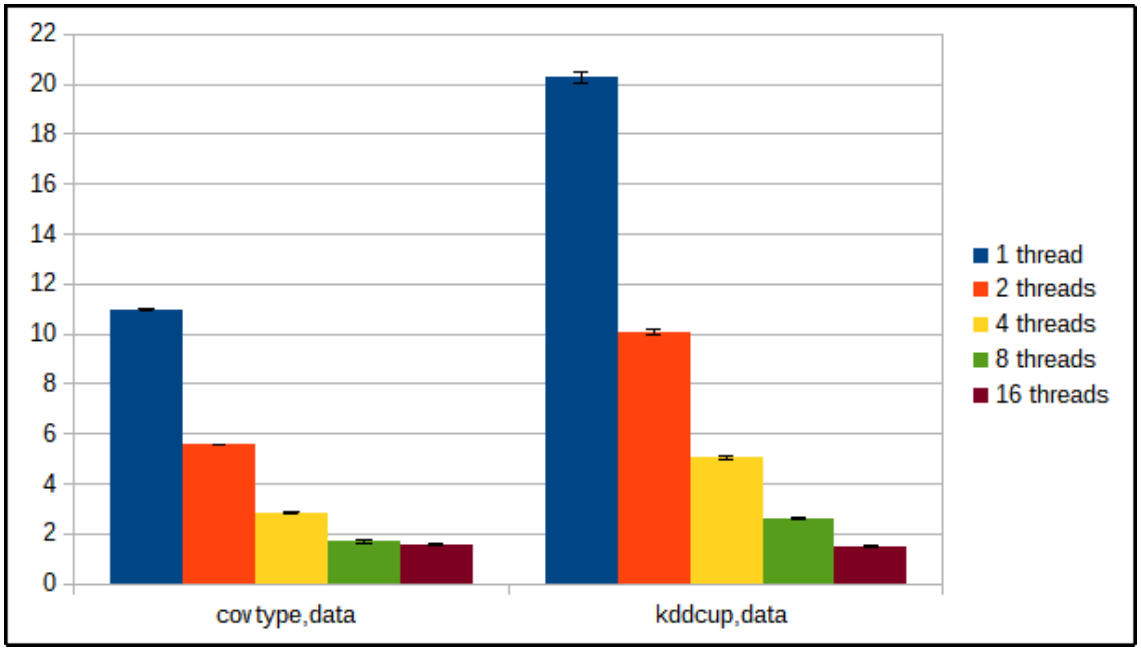


Figure A.148: Execution times (in minutes) of Classify ran on Intel 12-24 using SBLCWS with WShare (**first version**).

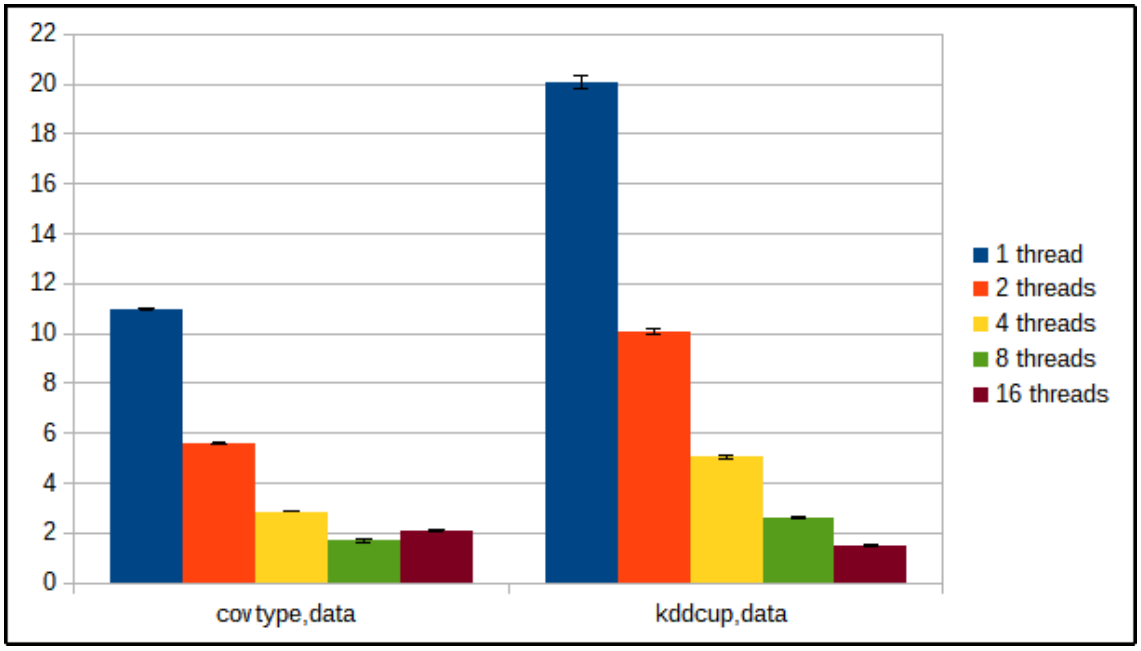


Figure A.149: Execution times (in minutes) of Classify ran on Intel 12-24 using SBLCWS with WShare (**second version**).

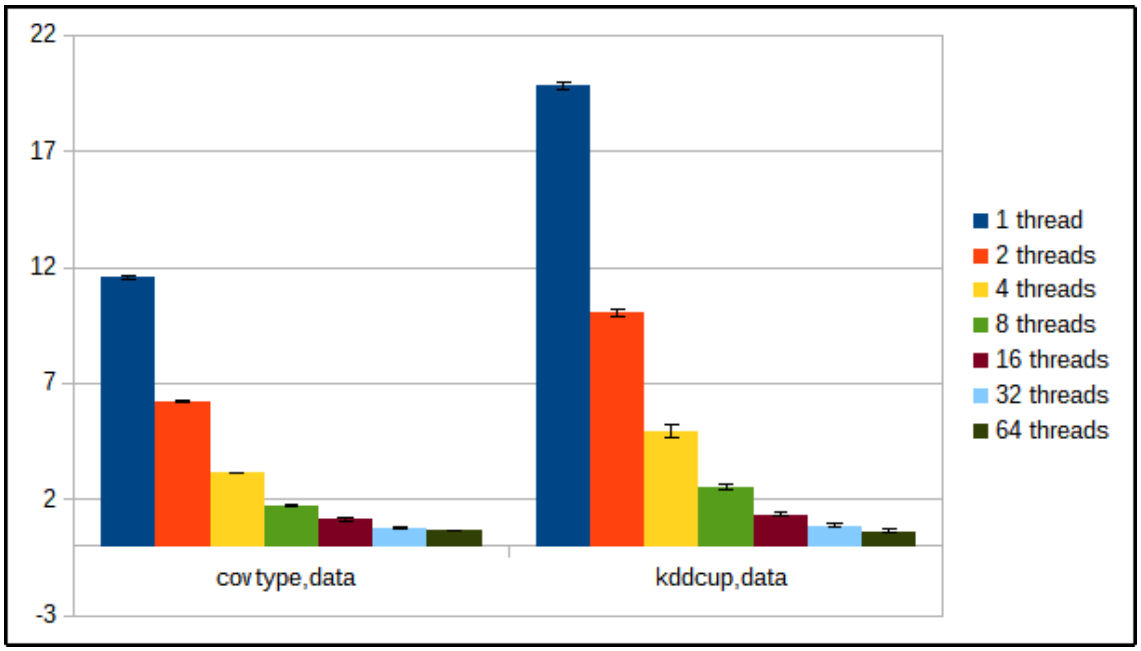


Figure A.150: Execution times (in minutes) of Classify ran on AMD 32-64 using WSteal

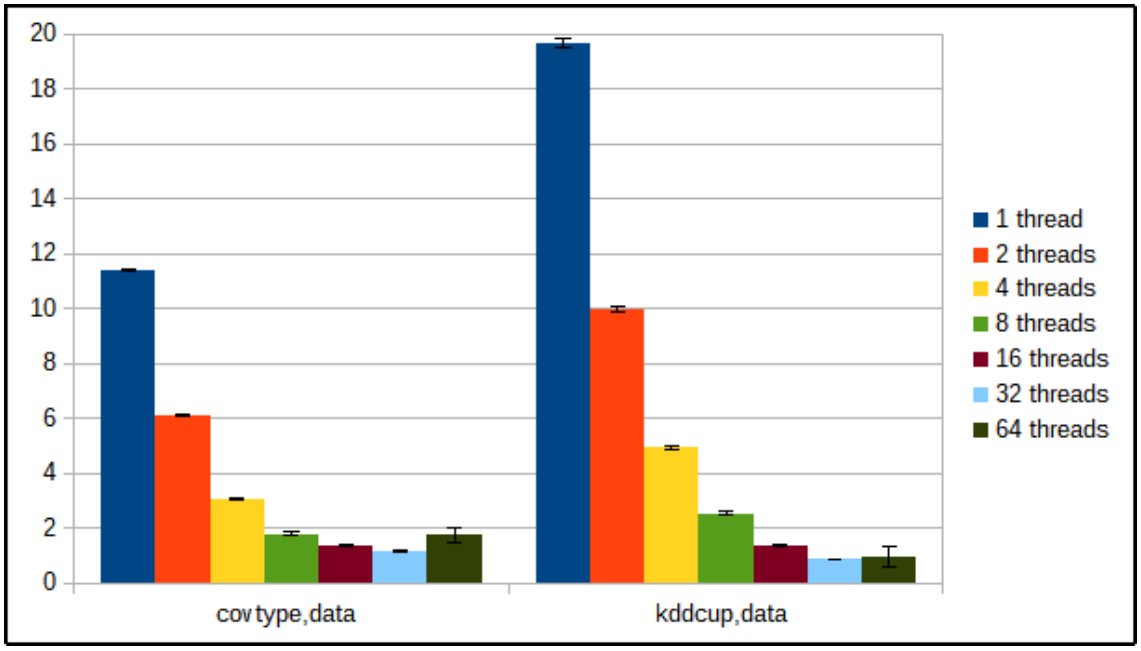


Figure A.151: Execution times (in minutes) of Classify ran on AMD 32-64 using LCWS

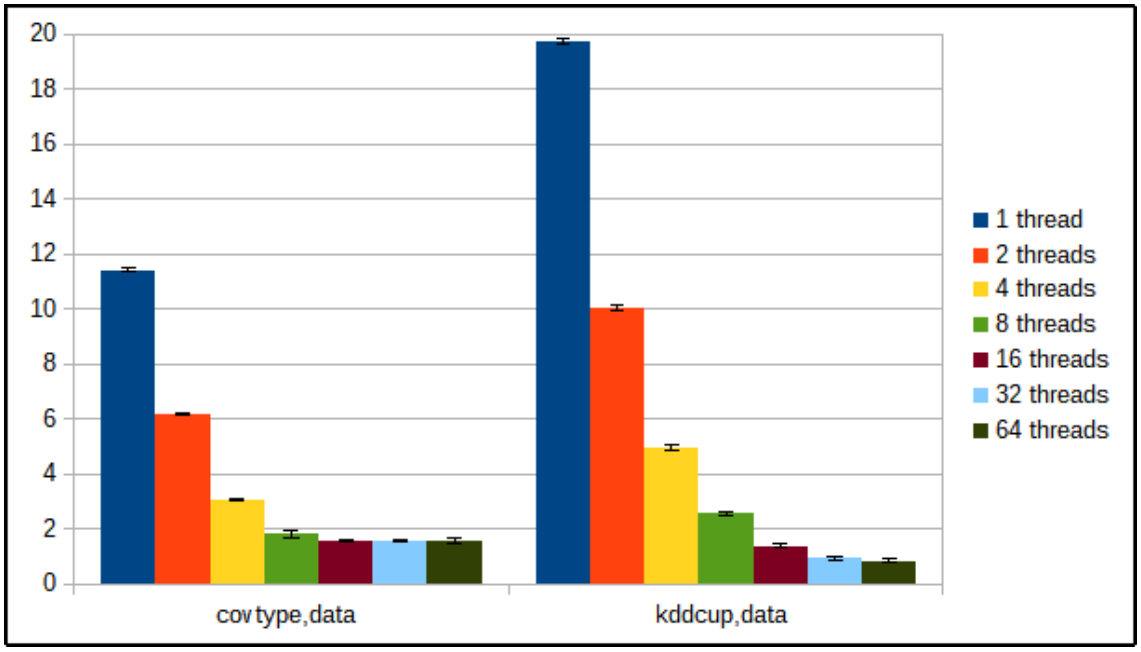


Figure A.152: Execution times (in minutes) of Classify ran on AMD 32-64 using SBLCWS (**first version**)

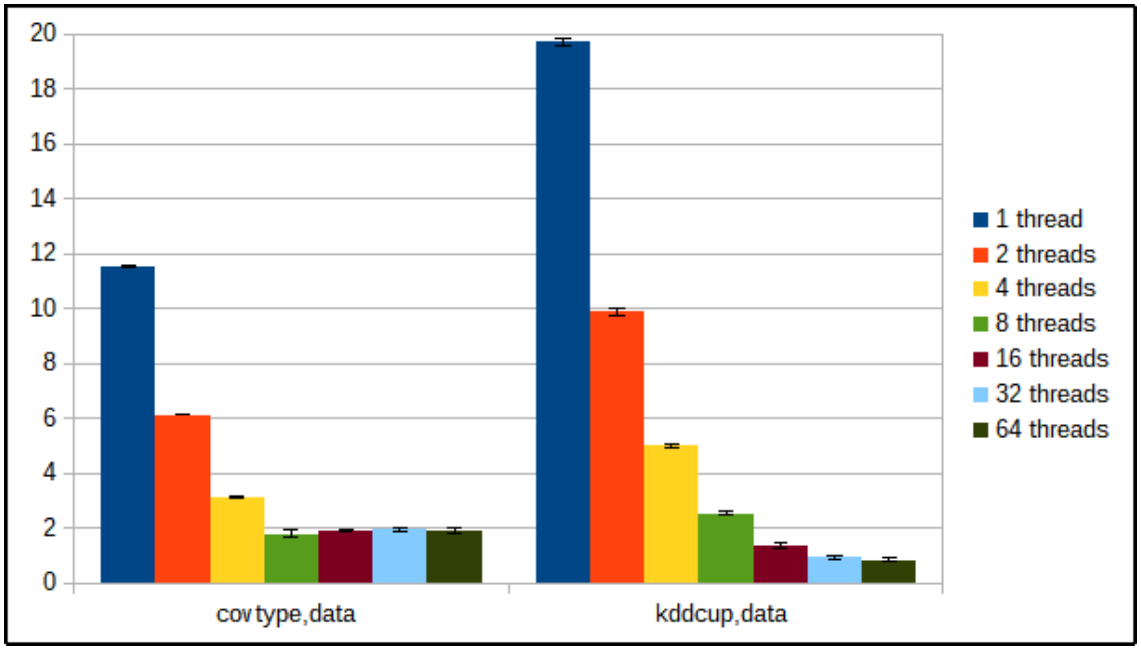


Figure A.153: Execution times (in minutes) of Classify ran on AMD 32-64 using SBLCWS (**second version**)

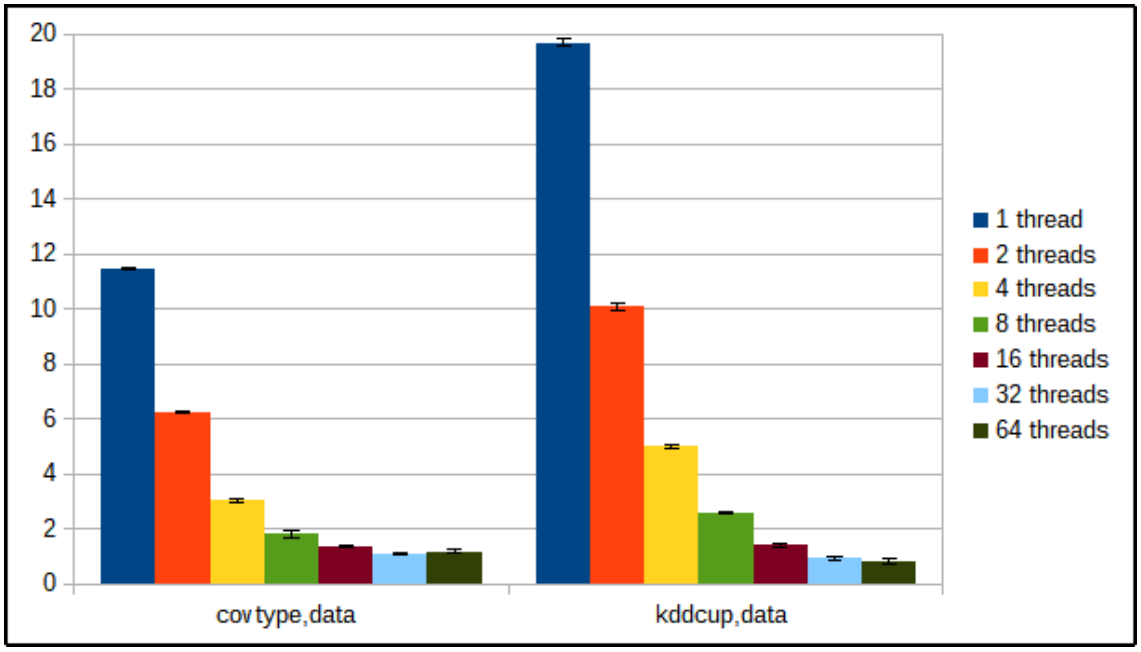


Figure A.154: Execution times (in minutes) of Classify ran on AMD 32-64 using LCWS with WShare (**tit-for-tat**).

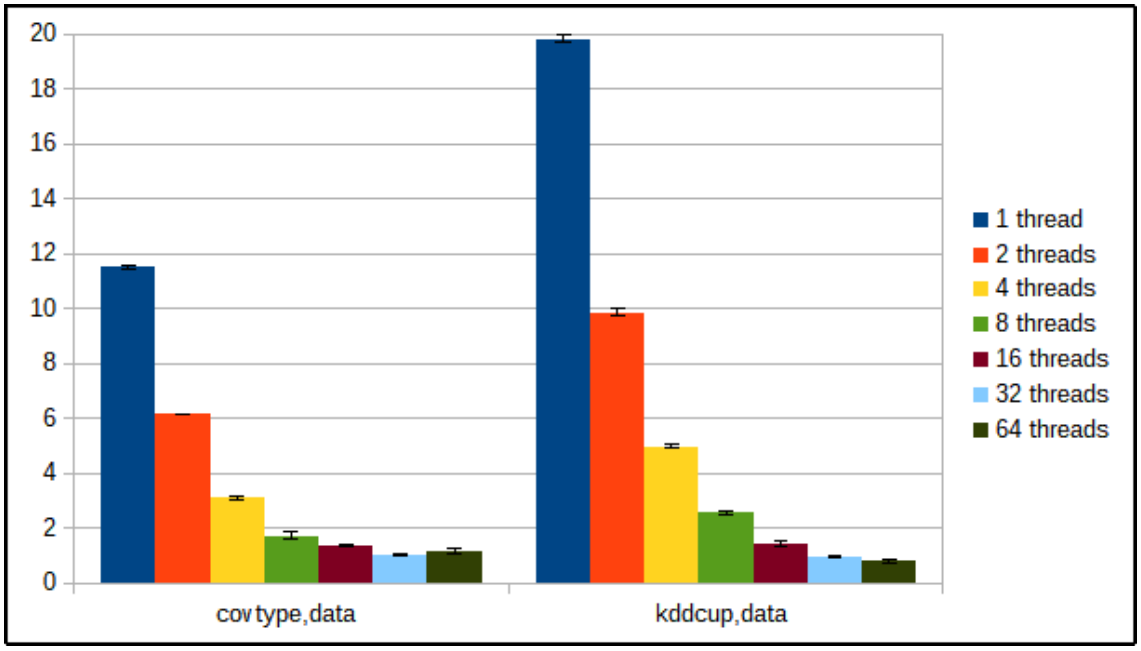


Figure A.155: Execution times (in minutes) of Classify ran on AMD 32-64 using LCWS with WShare (**continuous tit-for-tat**).

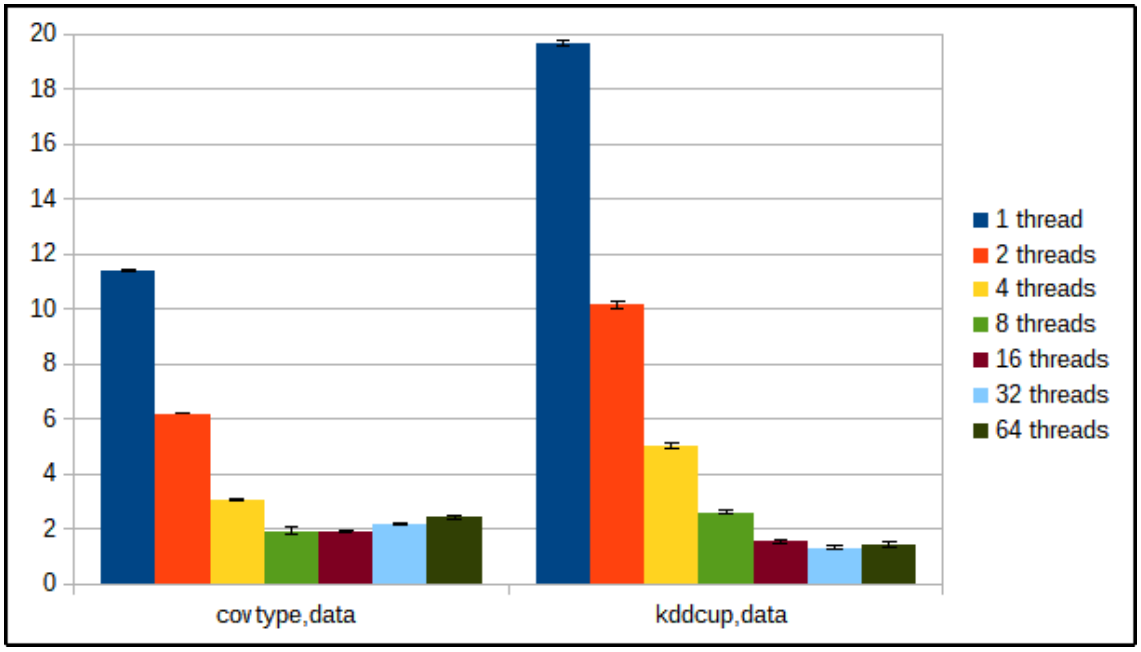


Figure A.156: Execution times (in minutes) of Classify ran on AMD 32-64 using SBLCWS with WShare (**first version**).

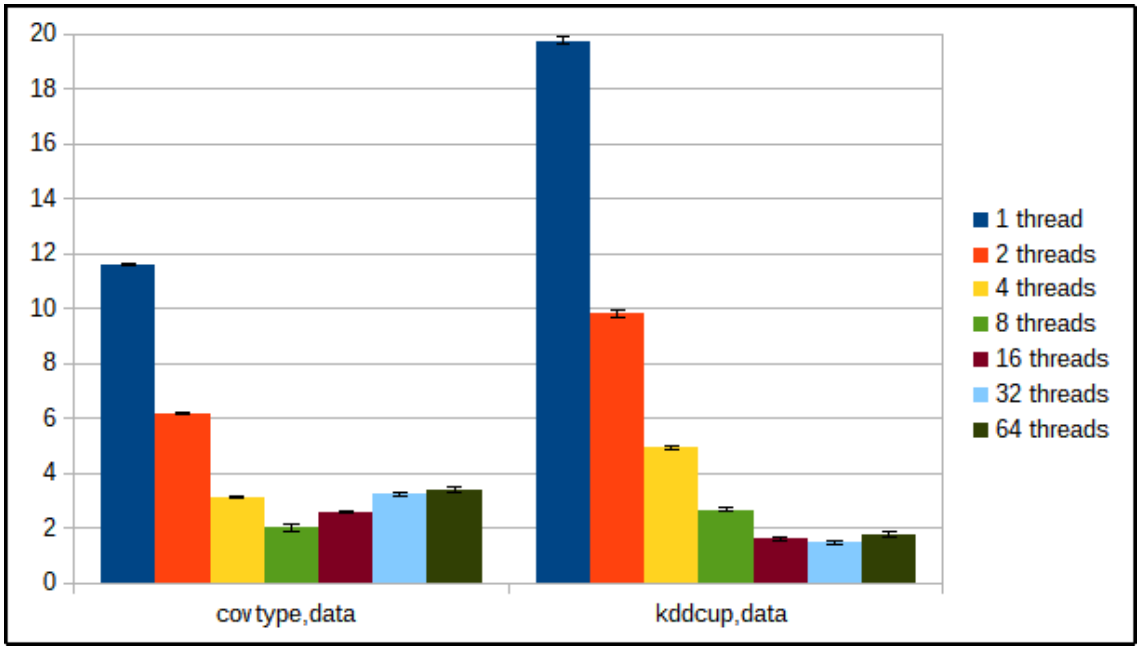


Figure A.157: Execution times (in minutes) of Classify ran on AMD 32-64 using SBLCWS with WShare (**second version**).

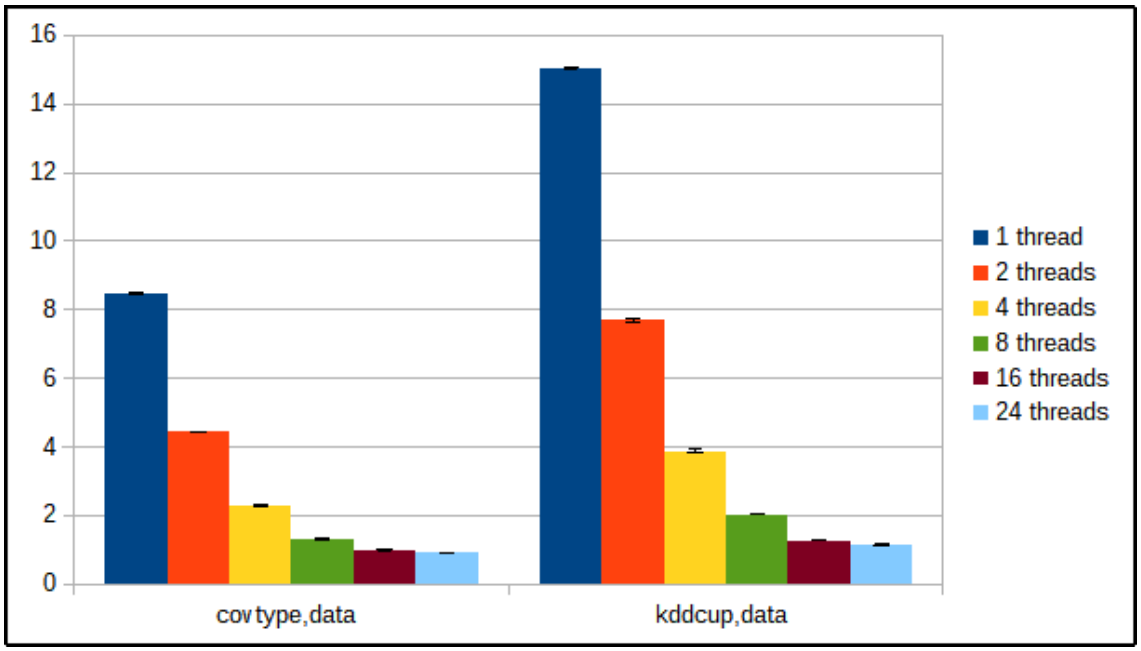


Figure A.158: Execution times (in minutes) of Classify ran on Intel 16-16 using WSteal

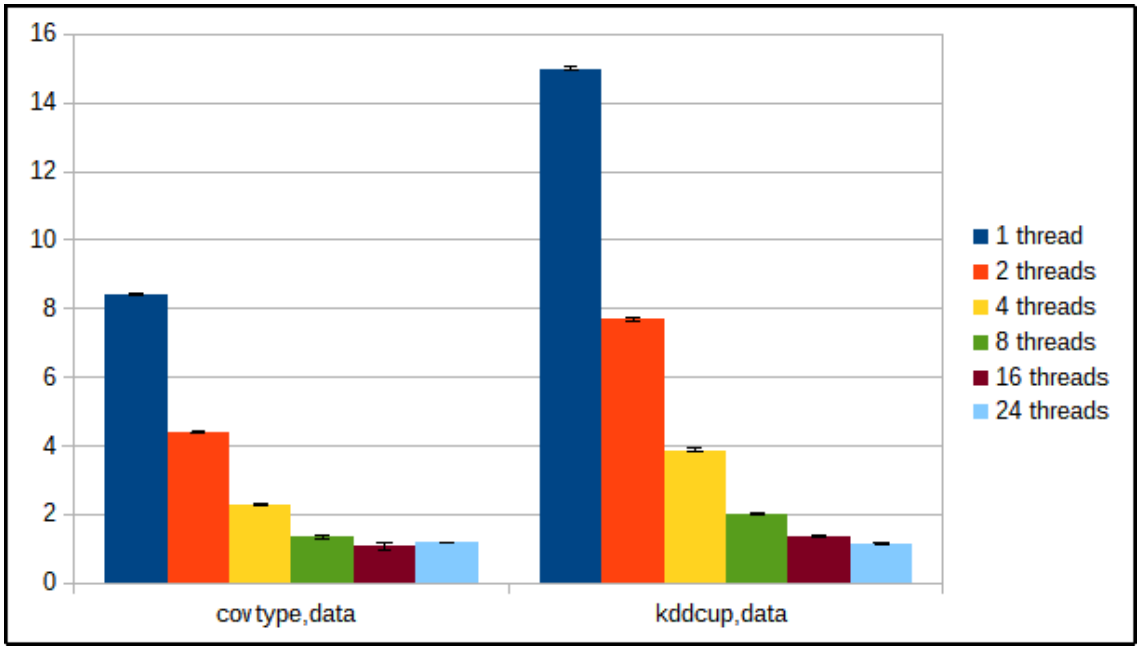


Figure A.159: Execution times (in minutes) of Classify ran on Intel 16-16 using LCWS

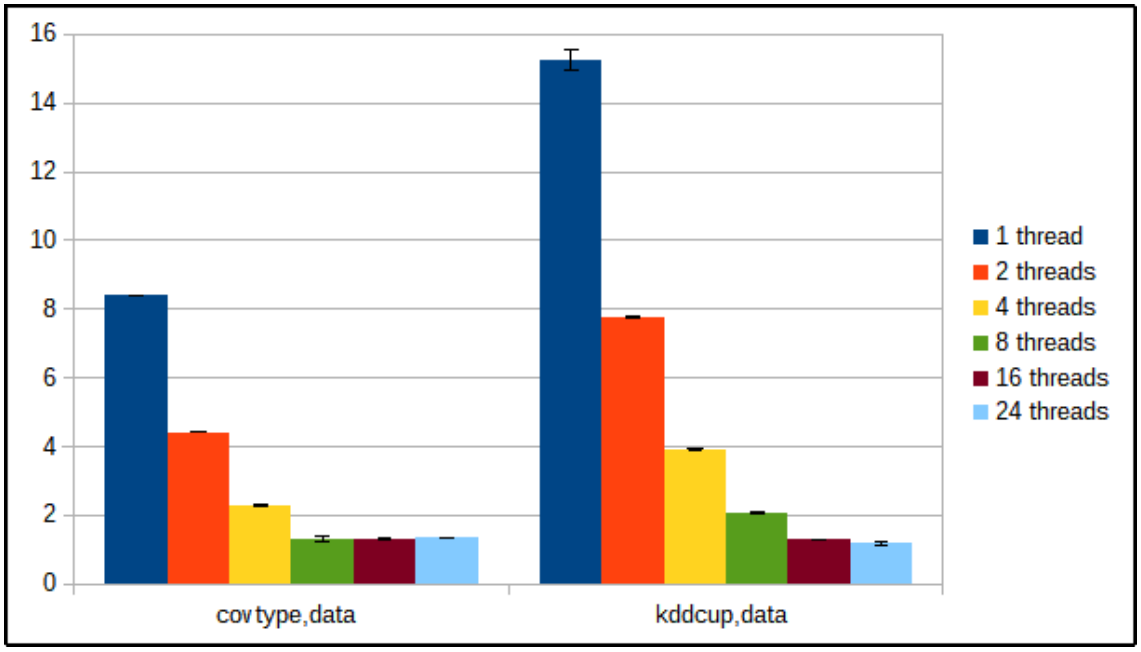


Figure A.160: Execution times (in minutes) of Classify ran on Intel 16-16 using SBLCWS (**first version**)

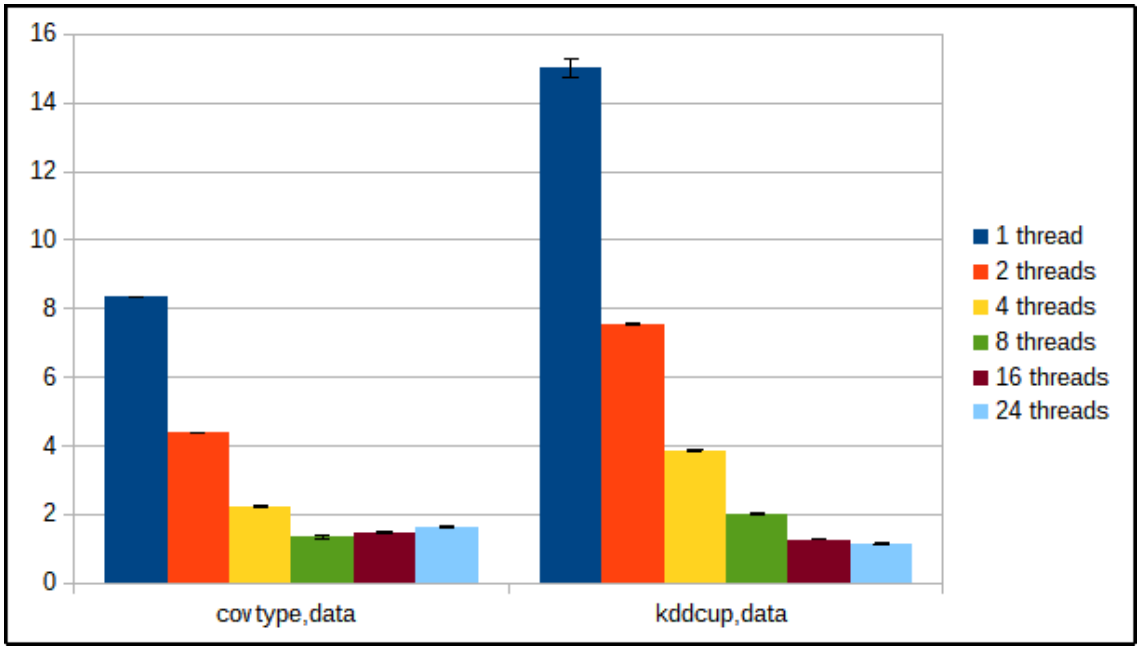


Figure A.161: Execution times (in minutes) of Classify ran on Intel 16-16 using SBLCWS (**second version**)

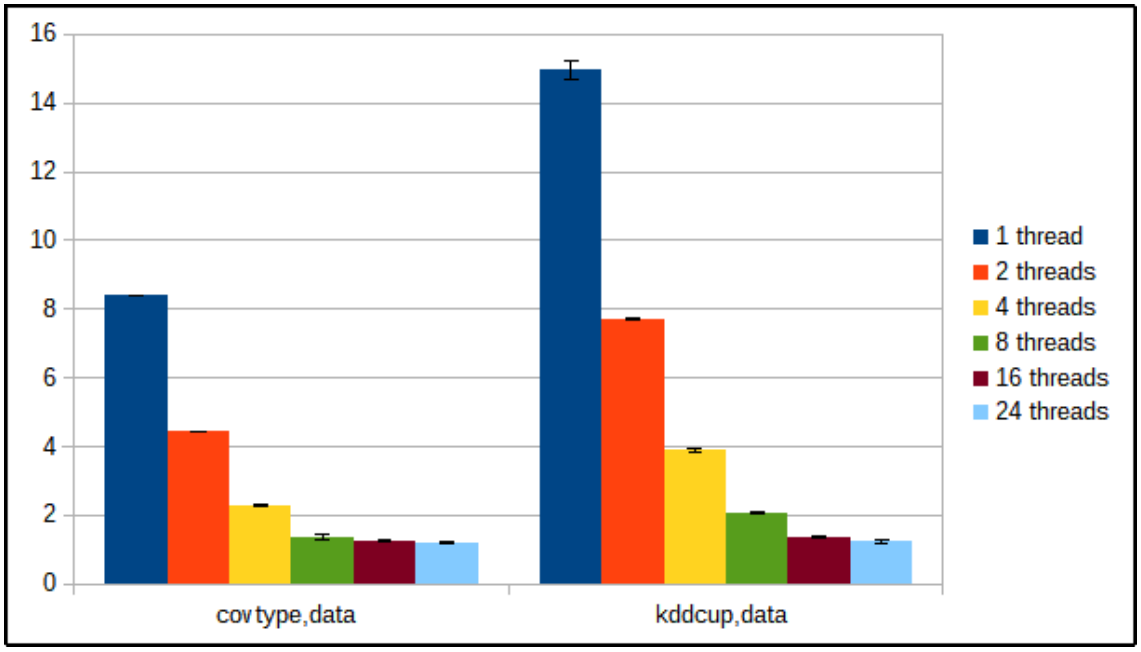


Figure A.162: Execution times (in minutes) of Classify ran on Intel 16-16 using LCWS with WShare (**tit-for-tat**).

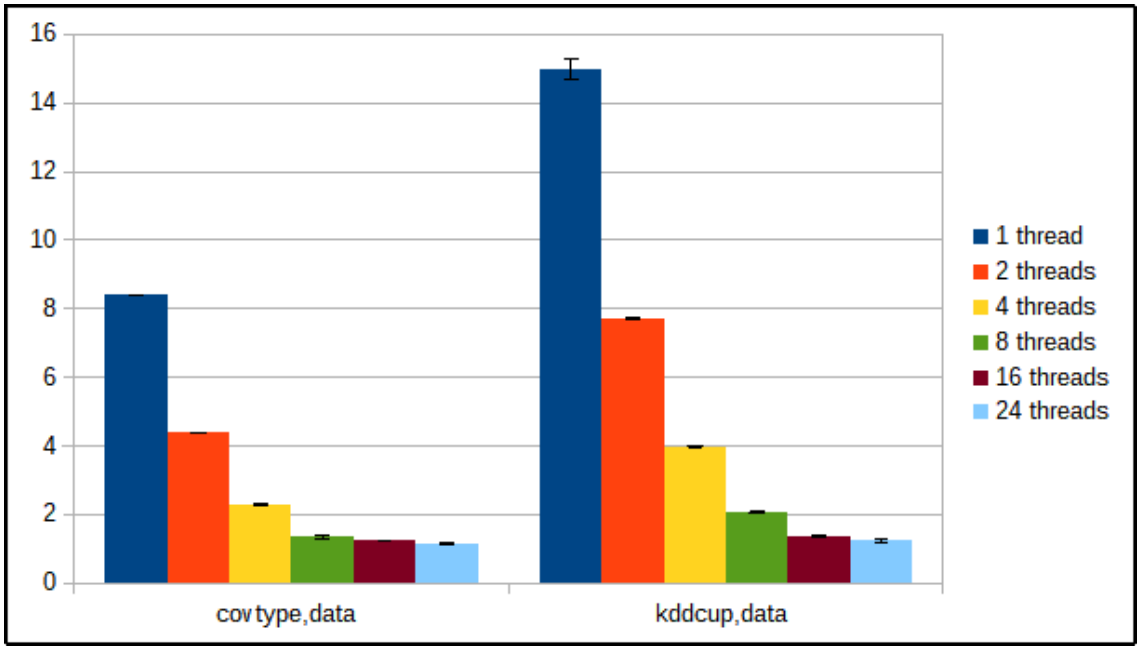


Figure A.163: Execution times (in minutes) of Classify ran on Intel 16-16 using LCWS with WShare (**continuous tit-for-tat**).

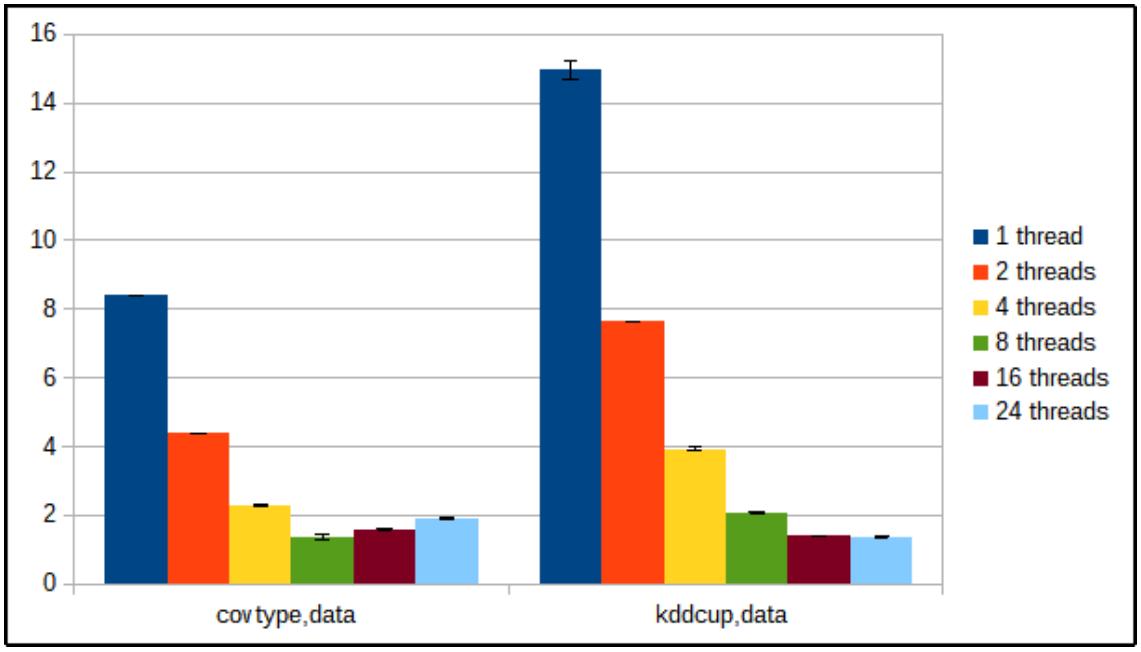


Figure A.164: Execution times (in minutes) of Classify ran on Intel 16-16 using SBLCWS with WShare (**first version**).

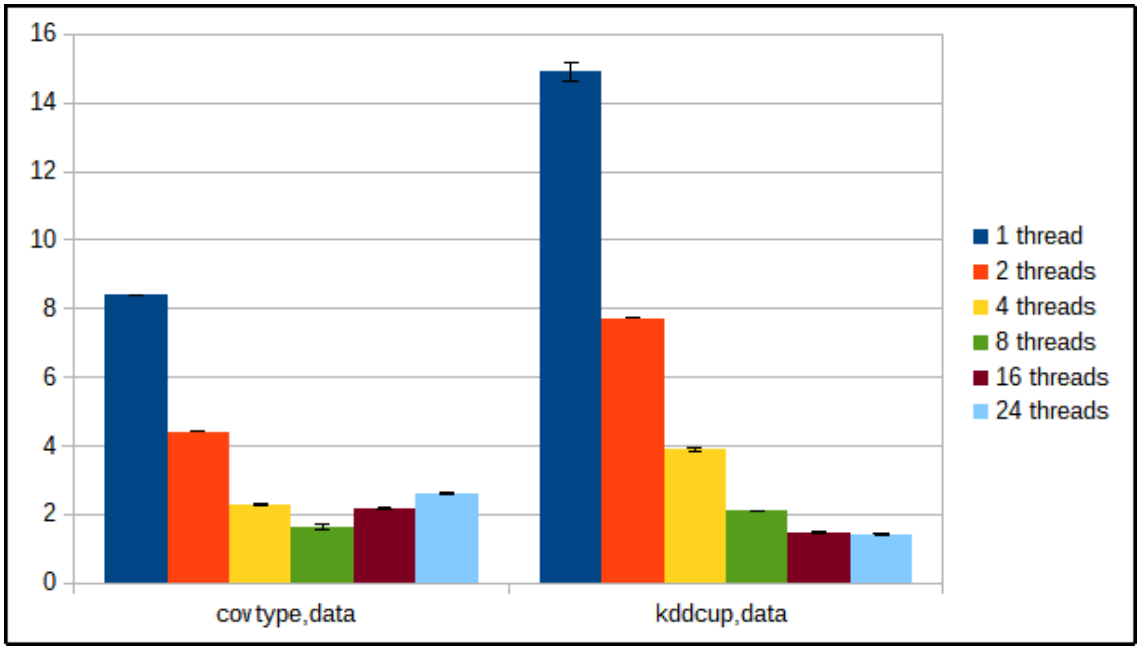


Figure A.165: Execution times (in minutes) of Classify ran on Intel 16-16 using SBLCWS with WShare (**second version**).

A.8 Spanning Forest

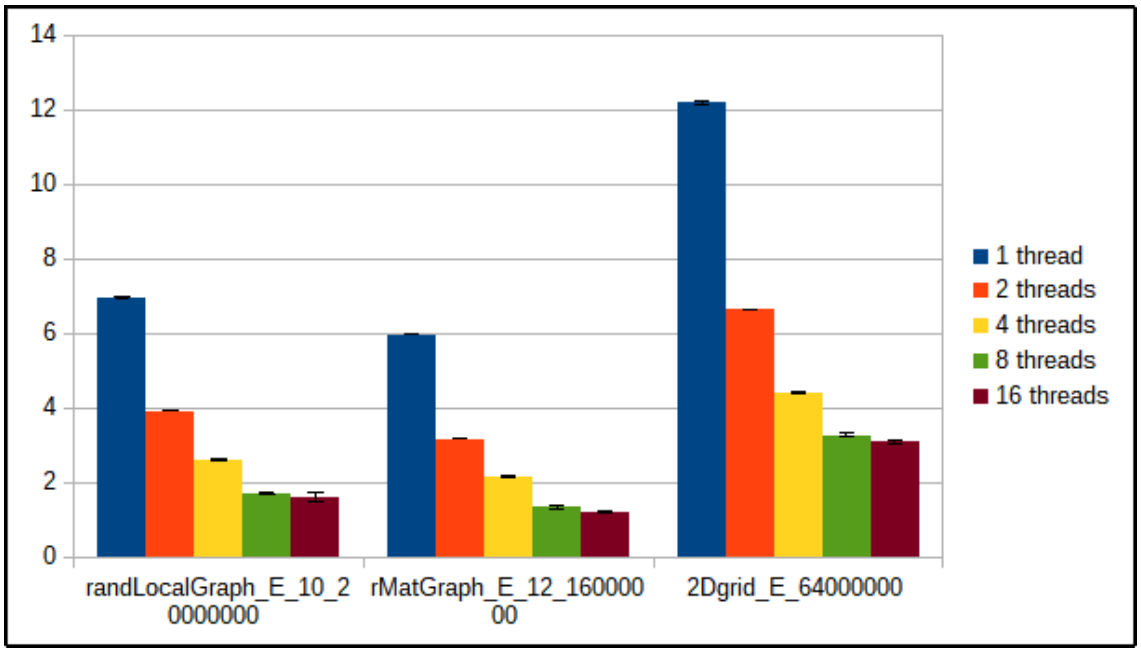


Figure A.166: Execution times (in minutes) of Spanning Forest ran on Intel 12-24 using WSteal

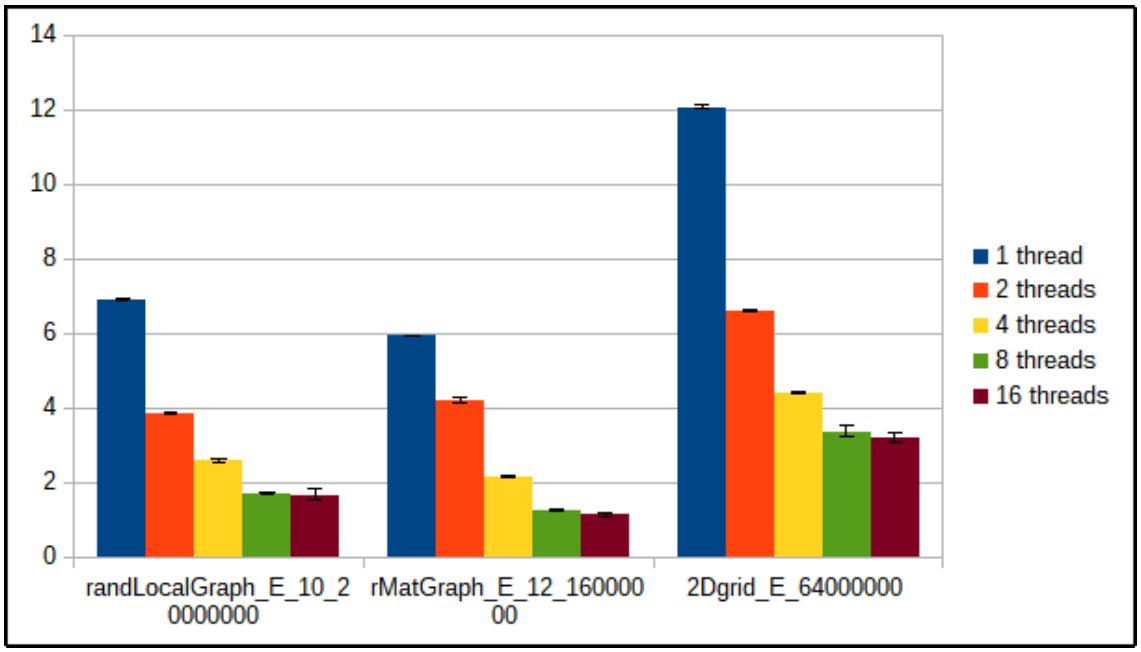


Figure A.167: Execution times (in minutes) of Spanning Forest ran on Intel 12-24 using LCWS

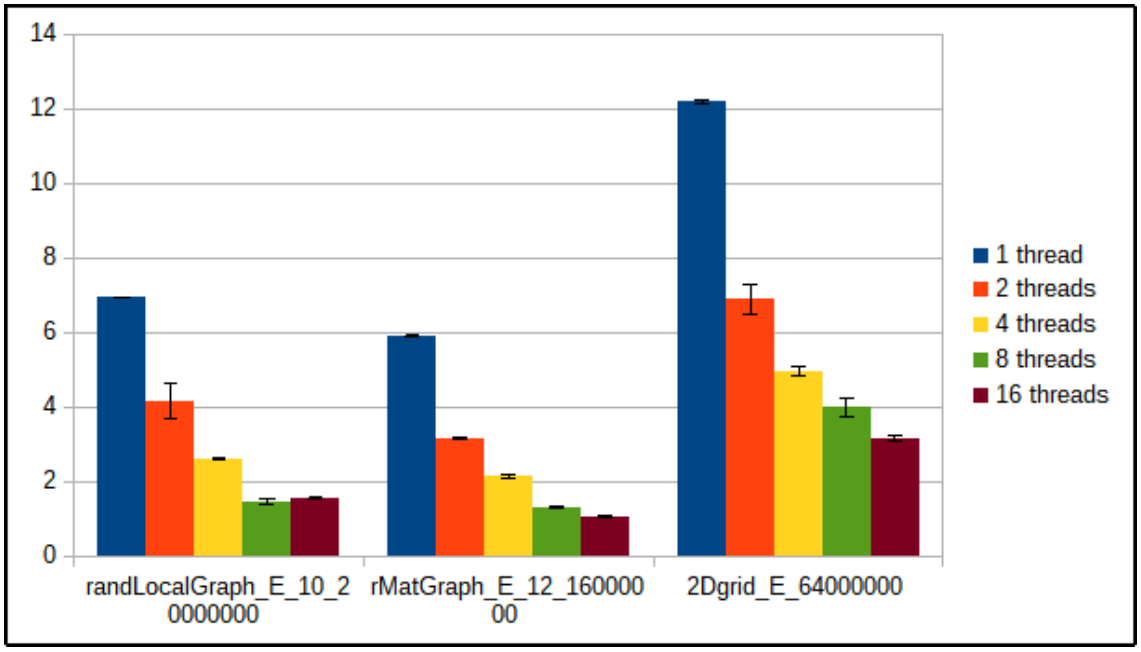


Figure A.168: Execution times (in minutes) of Spanning Forest ran on Intel 12-24 using SBLCWS (**first version**)

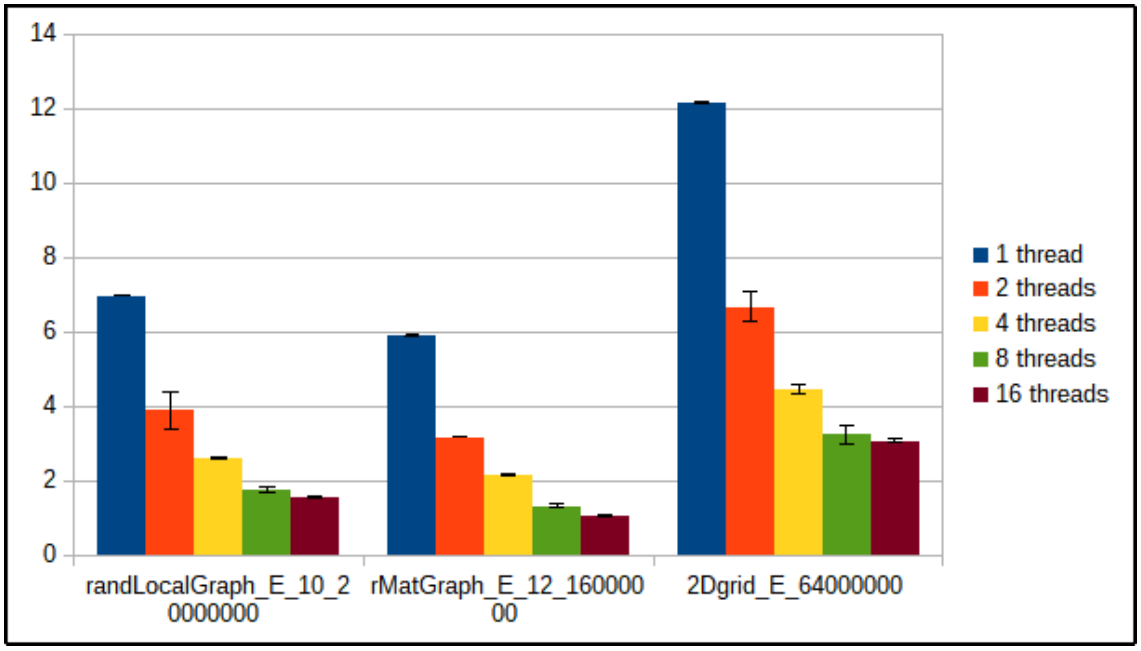


Figure A.169: Execution times (in minutes) of Spanning Forest ran on Intel 12-24 using SBLCWS (**second version**)

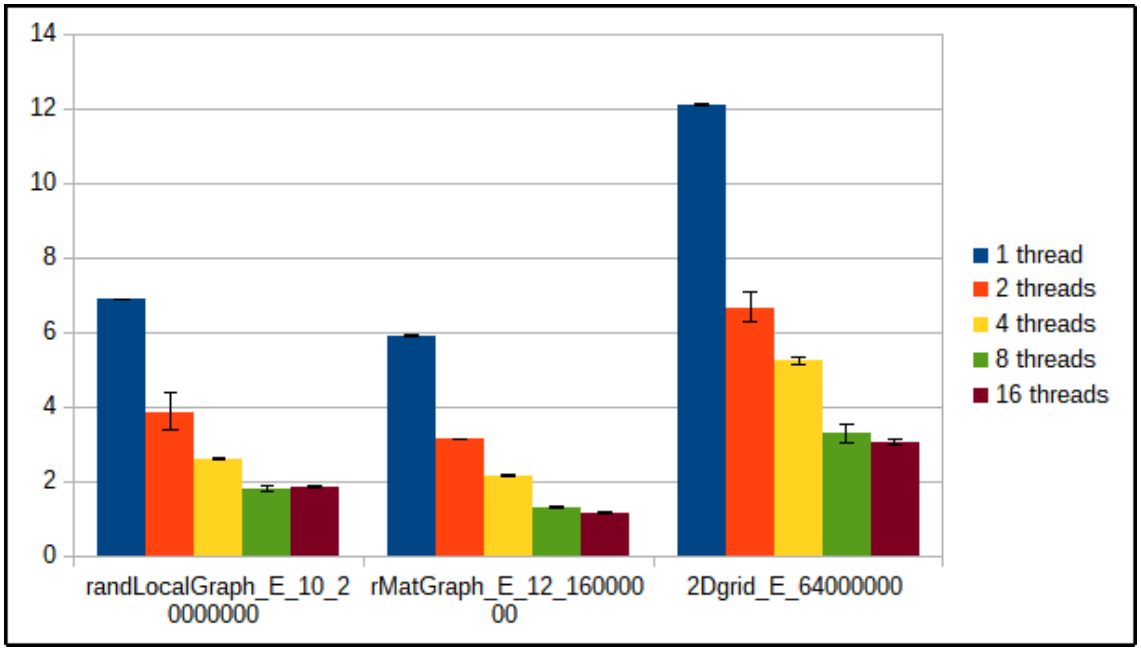


Figure A.170: Execution times (in minutes) of Spanning Forest ran on Intel 12-24 using LCWS with WShare (**tit-for-tat**).

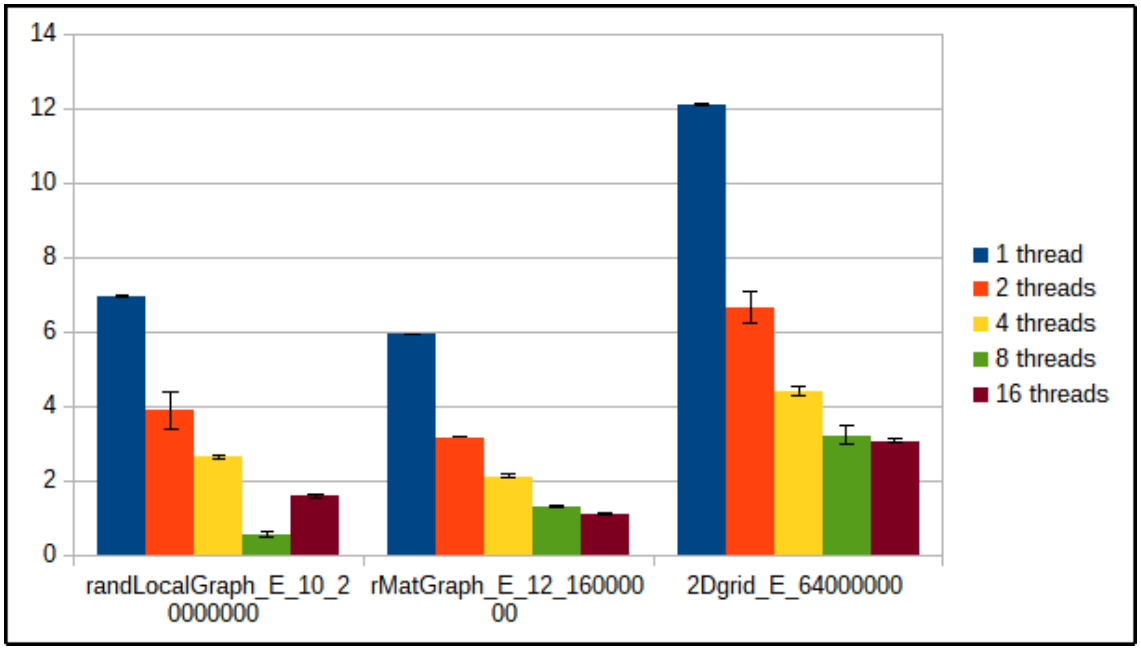


Figure A.171: Execution times (in minutes) of Spanning Forest ran on Intel 12-24 using LCWS with WShare (**continuous tit-for-tat**).

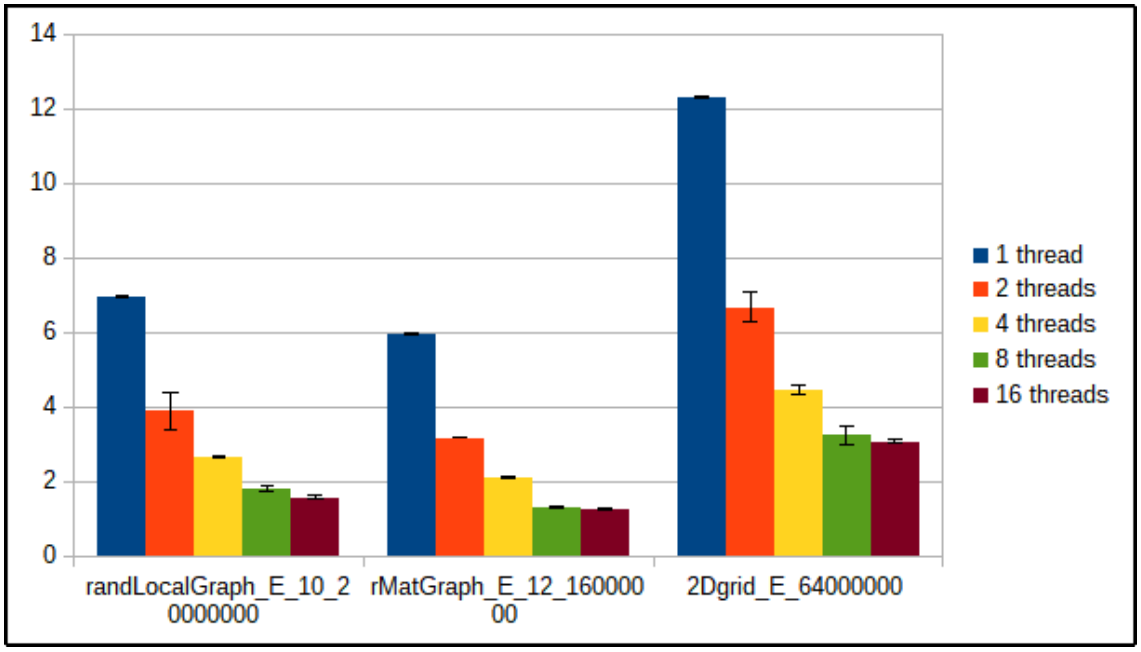


Figure A.172: Execution times (in minutes) of Spanning Forest ran on Intel 12-24 using SBLCWS with WShare (**first version**).

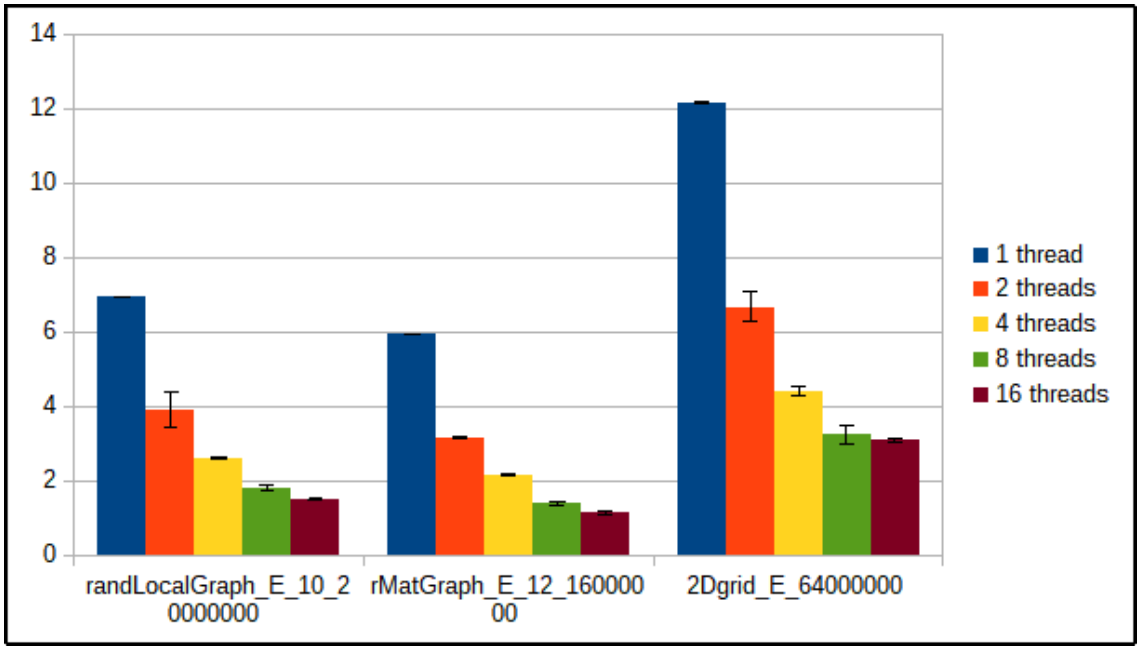


Figure A.173: Execution times (in minutes) of Spanning Forest ran on Intel 12-24 using SBLCWS with WShare (**second version**).

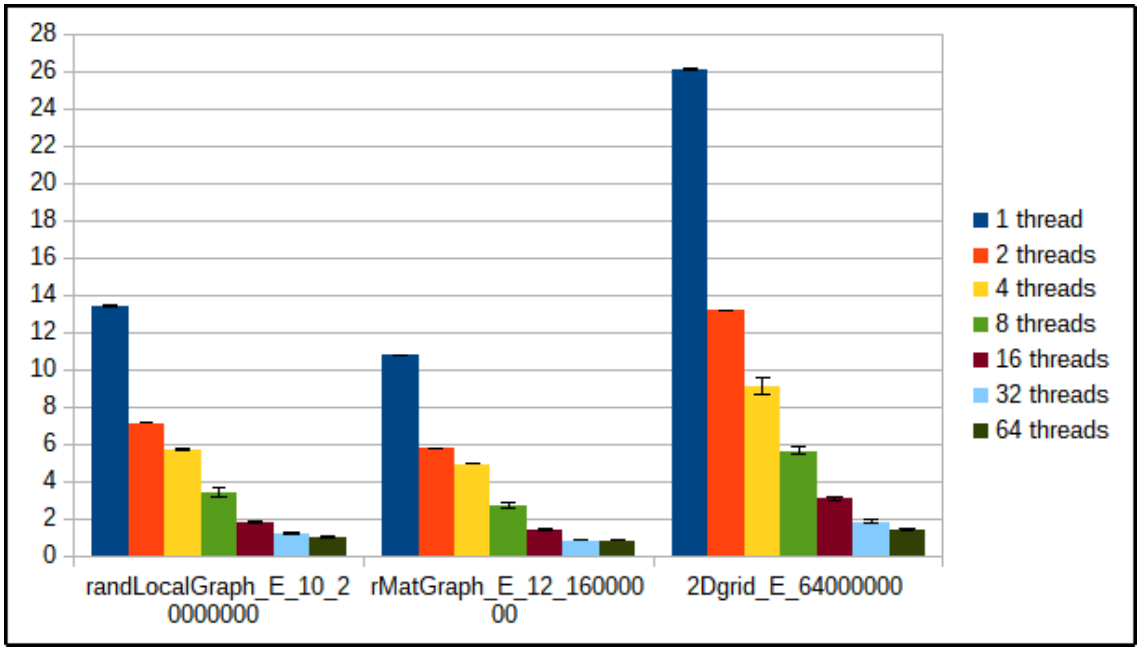


Figure A.174: Execution times (in minutes) of Spanning Forest ran on AMD 32-64 using WSteal

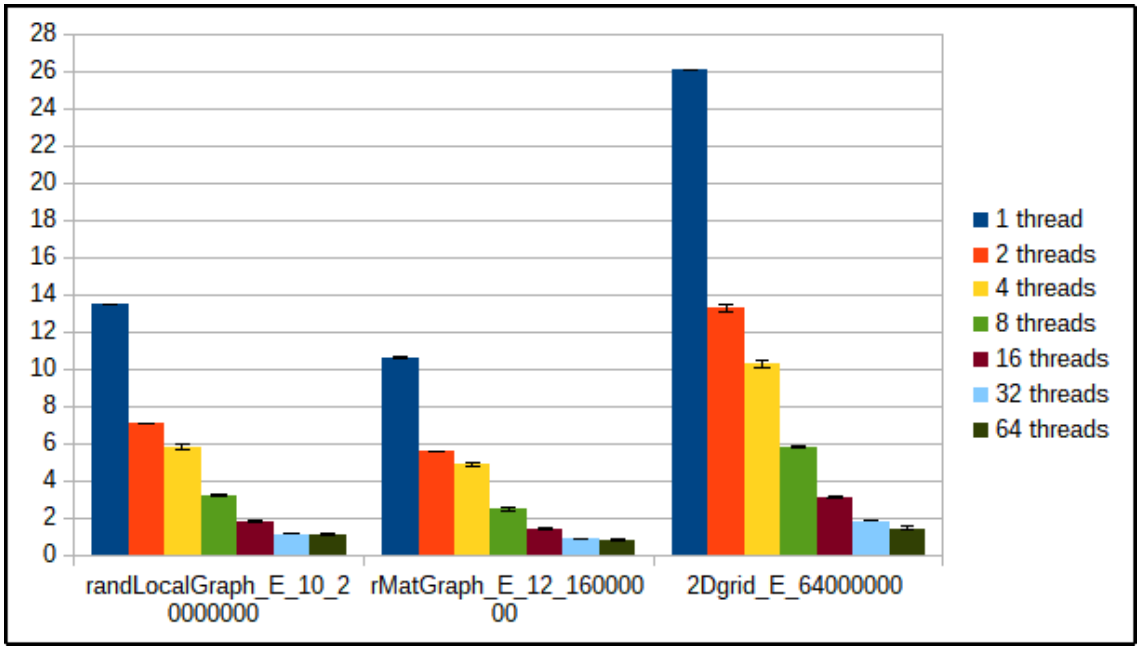


Figure A.175: Execution times (in minutes) of Spanning Forest ran on AMD 32-64 using LCWS

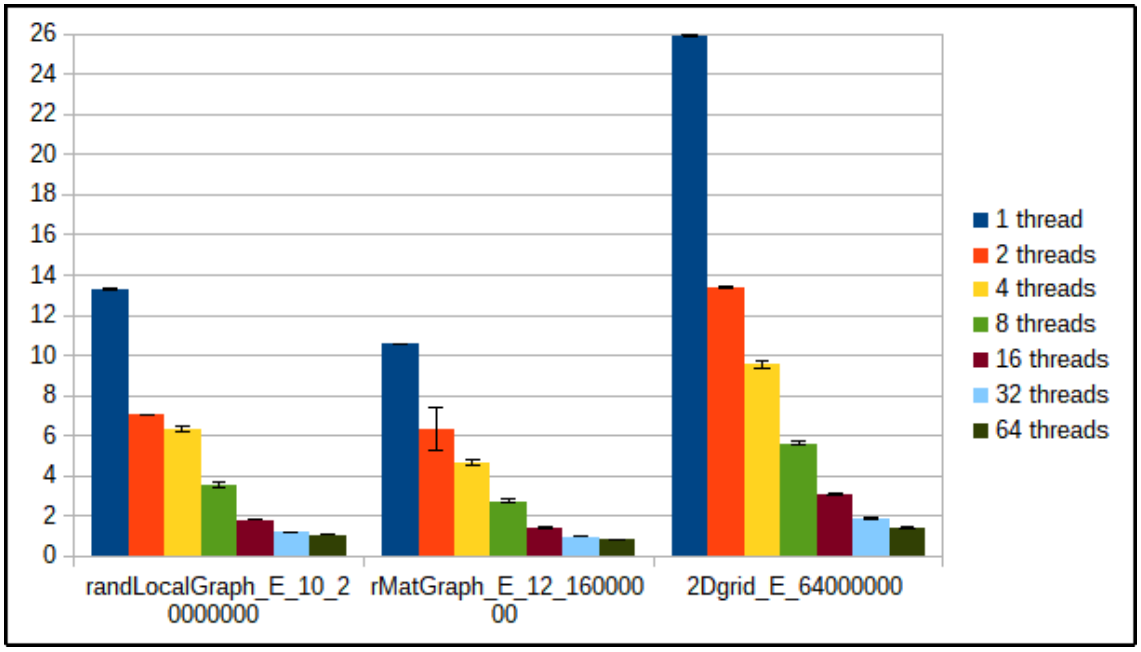


Figure A.176: Execution times (in minutes) of Spanning Forest ran on AMD 32-64 using SBLCWS (**first version**)

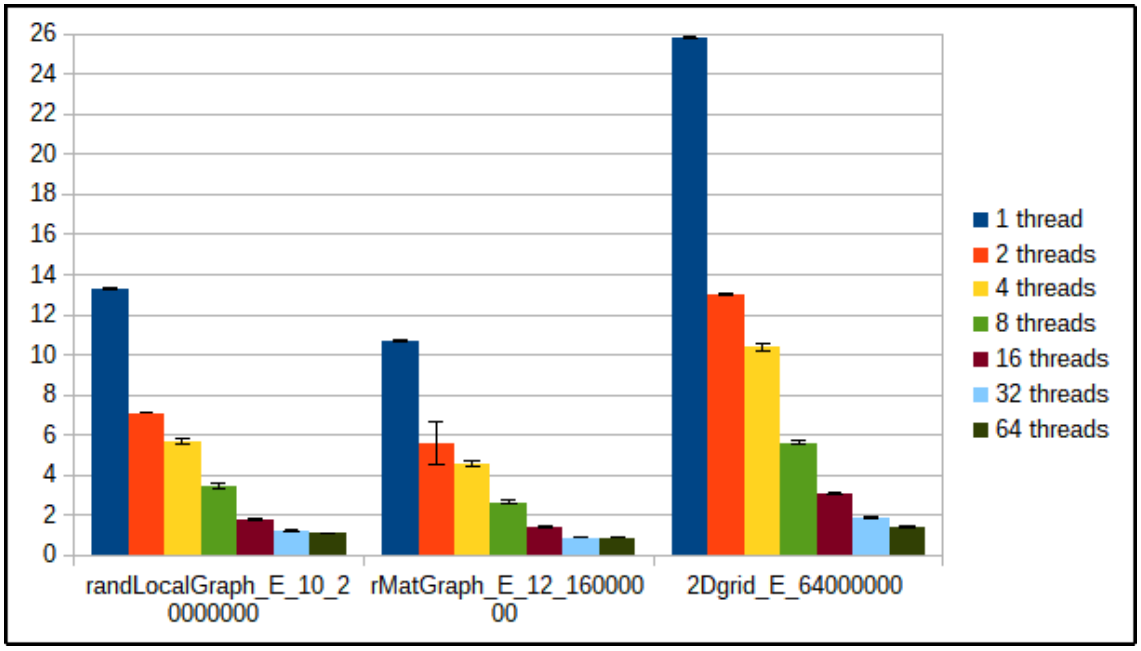


Figure A.177: Execution times (in minutes) of Spanning Forest ran on AMD 32-64 using SBLCWS (**second version**)

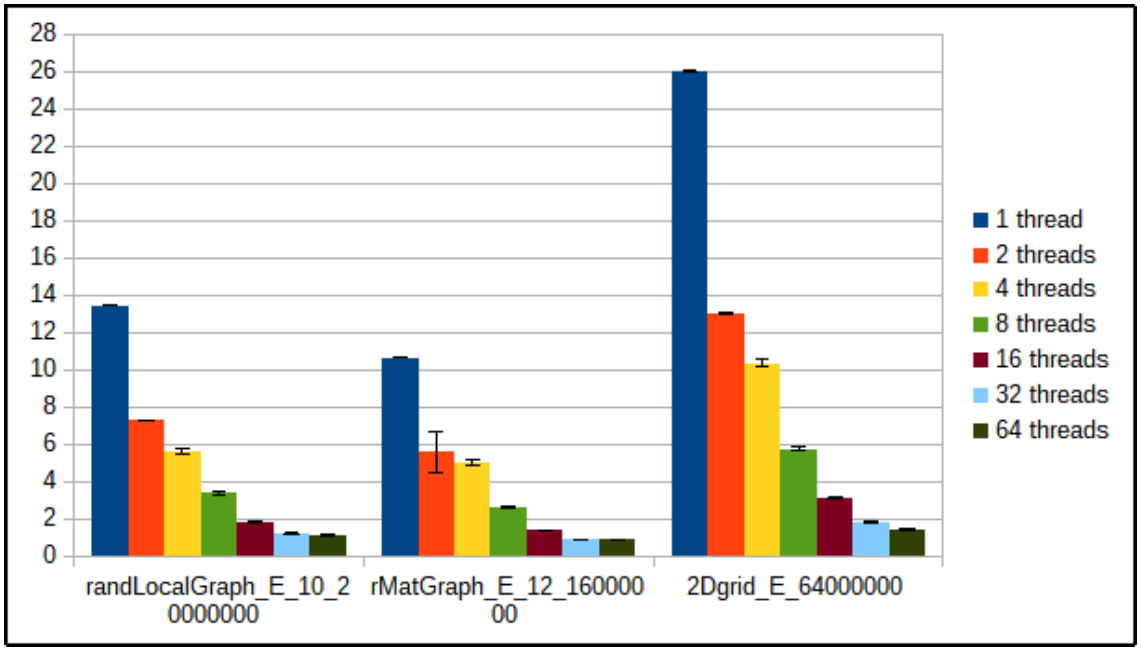


Figure A.178: Execution times (in minutes) of Spanning Forest ran on AMD 32-64 using LCWS with WShare (**tit-for-tat**).

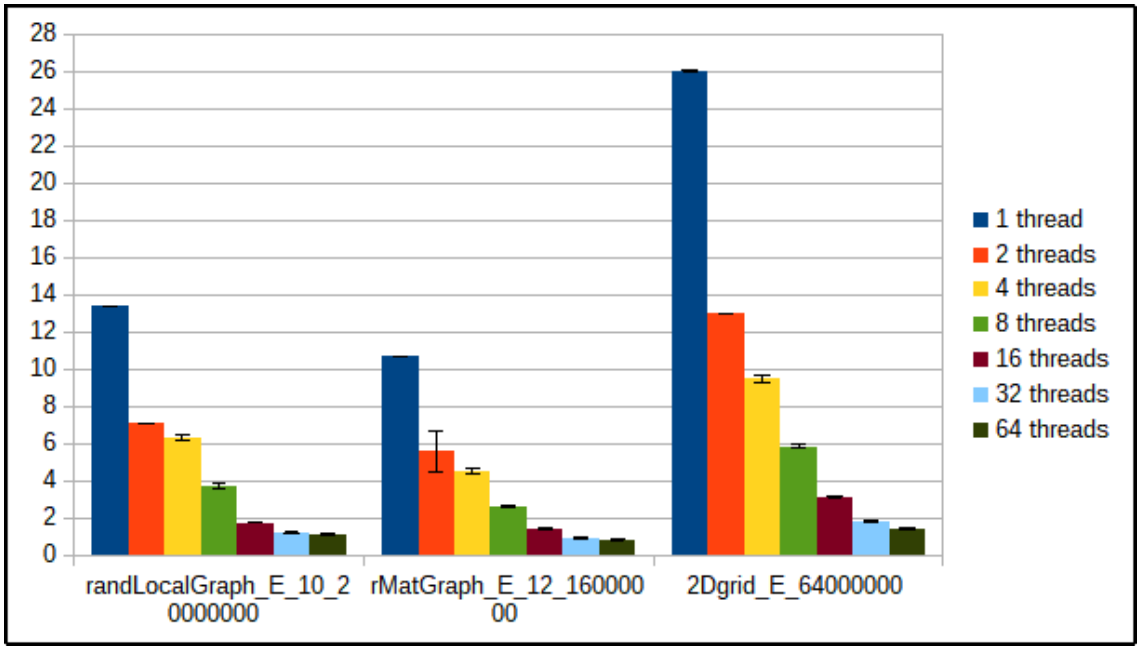


Figure A.179: Execution times (in minutes) of Spanning Forest ran on AMD 32-64 using LCWS with WShare (**continuous tit-for-tat**).

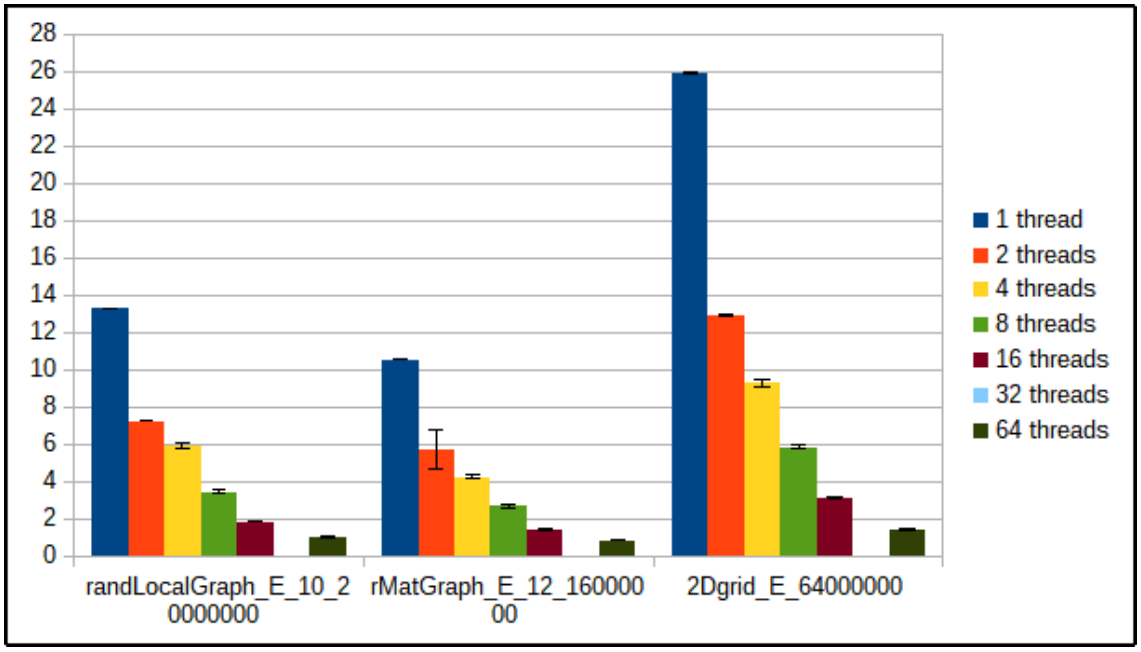


Figure A.180: Execution times (in minutes) of Spanning Forest ran on AMD 32-64 using SBLCWS with WShare (**first version**).

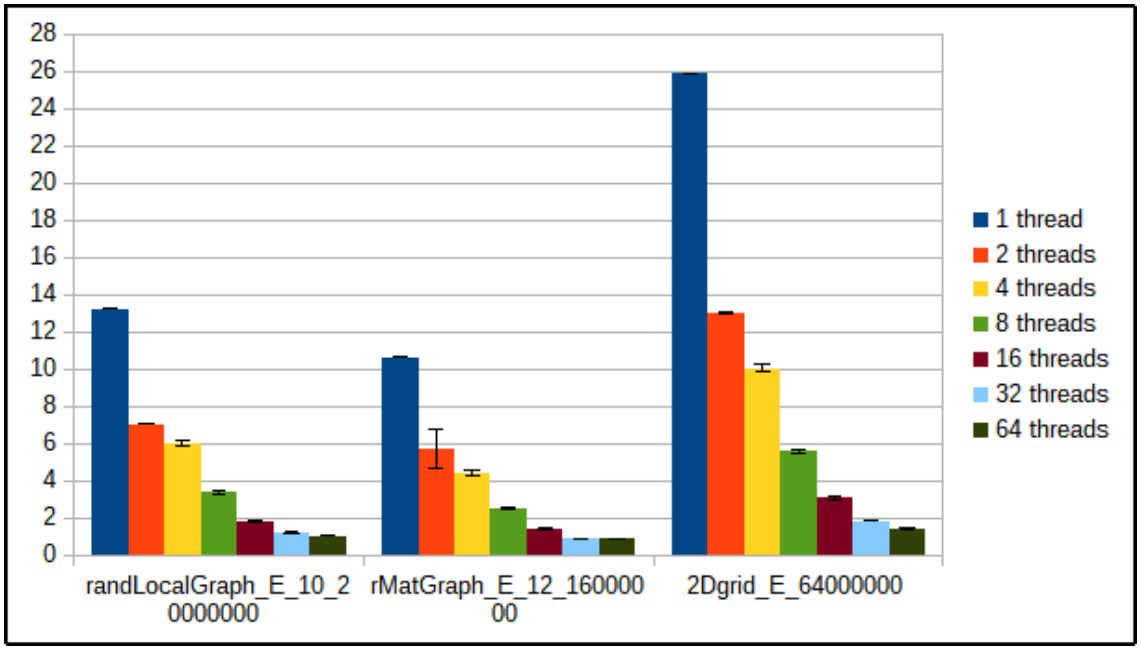


Figure A.181: Execution times (in minutes) of Spanning Forest ran on AMD 32-64 using SBLCWS with WShare (**second version**).

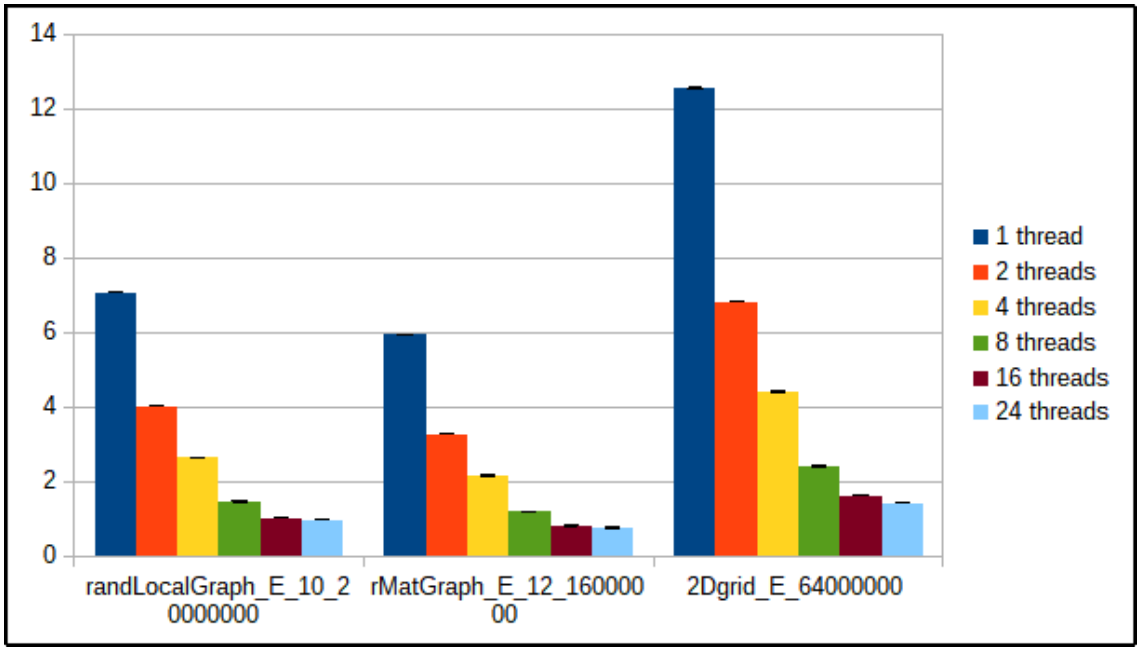


Figure A.182: Execution times (in minutes) of Spanning Forest ran on Intel 16-16 using WSteal

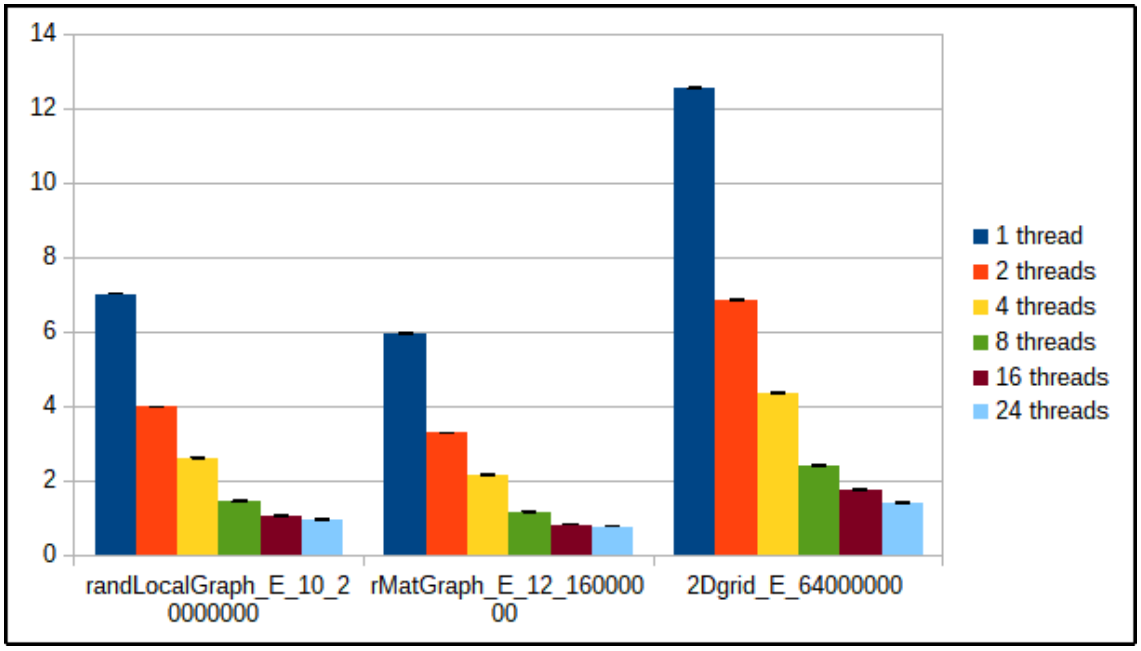


Figure A.183: Execution times (in minutes) of Spanning Forest ran on Intel 16-16 using LCWS

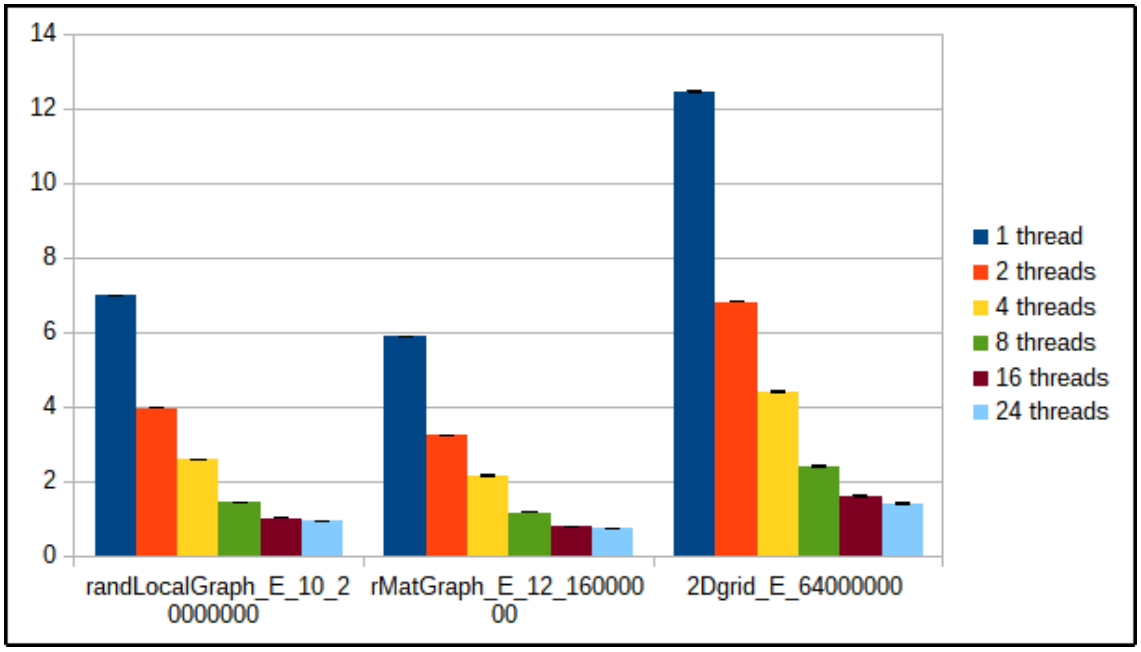


Figure A.184: Execution times (in minutes) of Spanning Forest ran on Intel 16-16 using SBLCWS (**first version**)

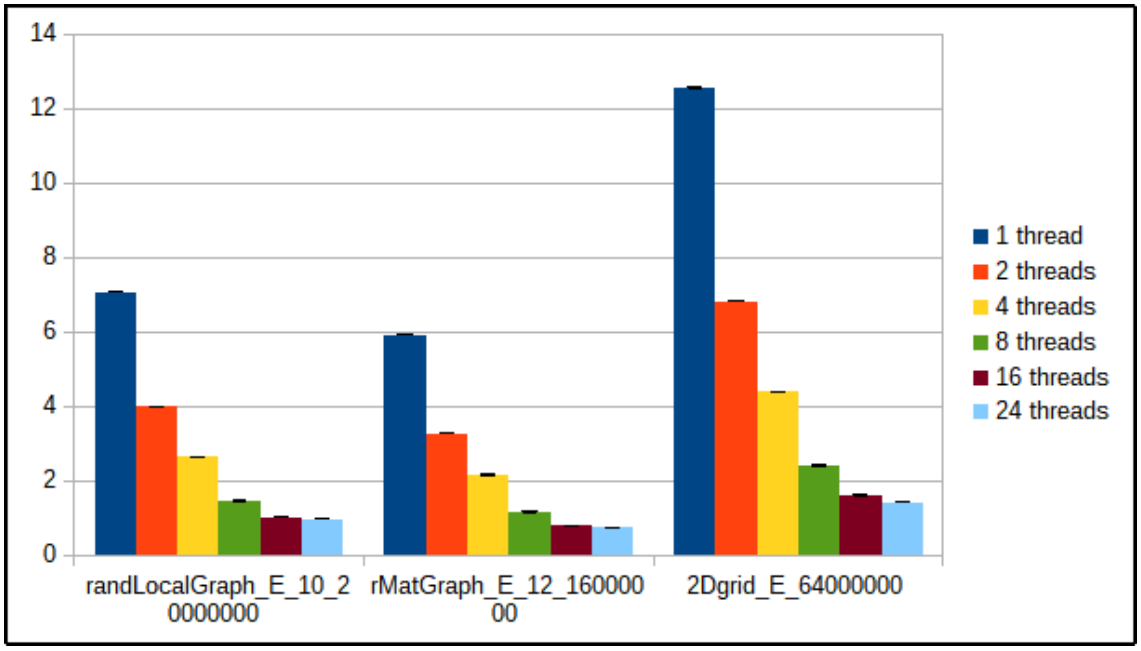


Figure A.185: Execution times (in minutes) of Spanning Forest ran on Intel 16-16 using SBLCWS (**second version**)

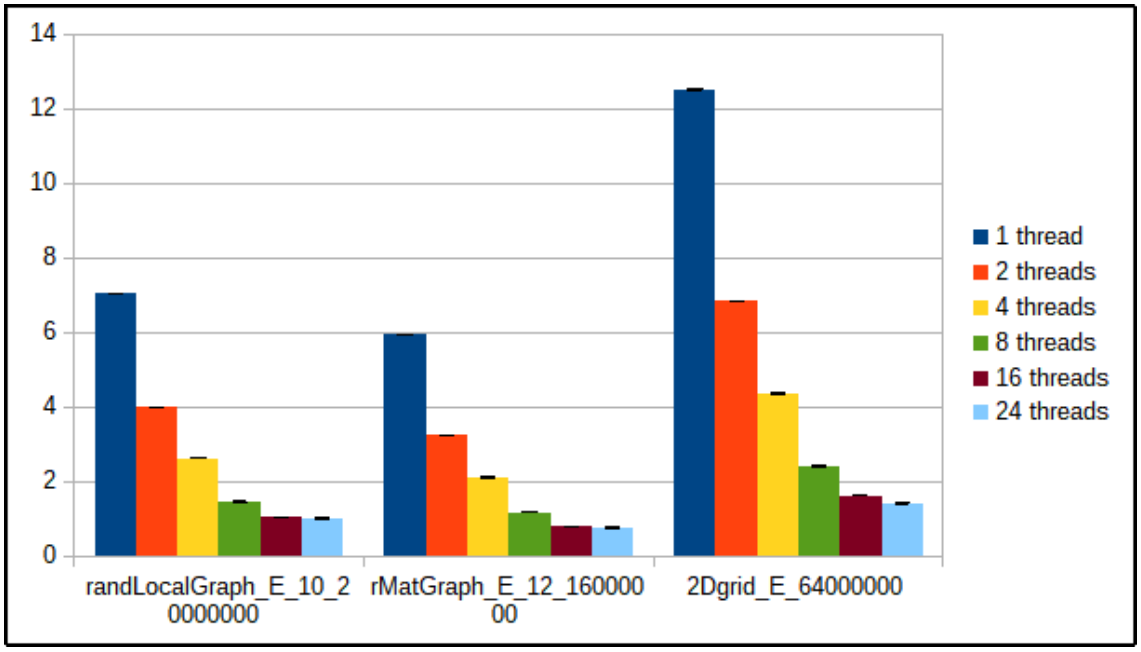


Figure A.186: Execution times (in minutes) of Spanning Forest ran on Intel 16-16 using LCWS with WShare (**tit-for-tat**).

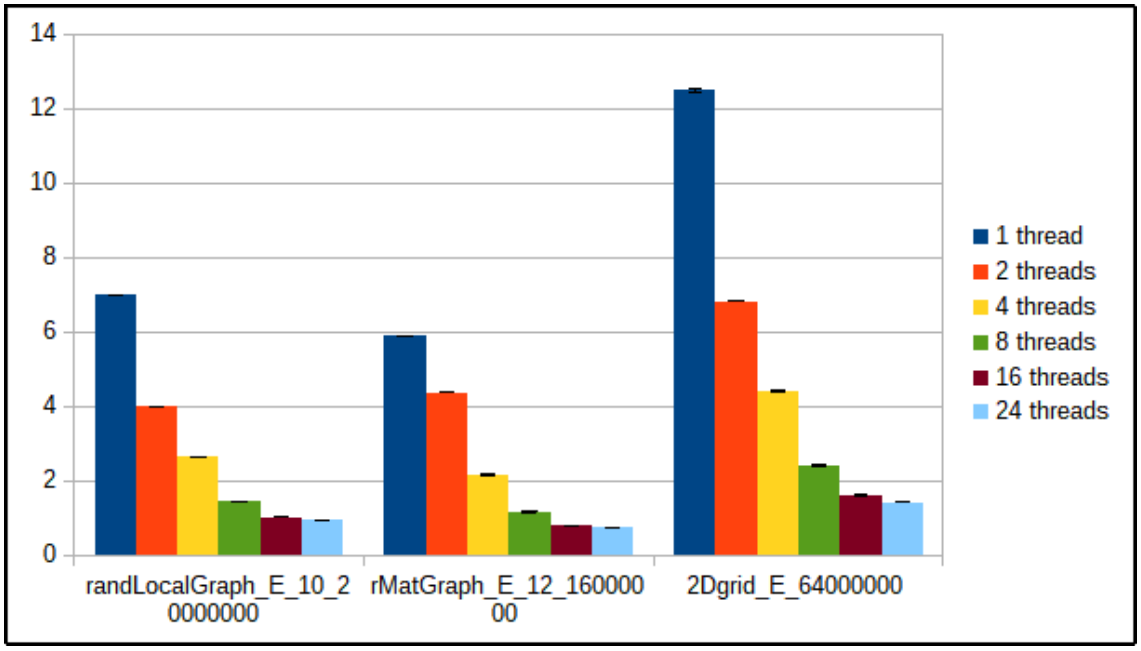


Figure A.187: Execution times (in minutes) of Spanning Forest ran on Intel 16-16 using LCWS with WShare (**continuous tit-for-tat**).

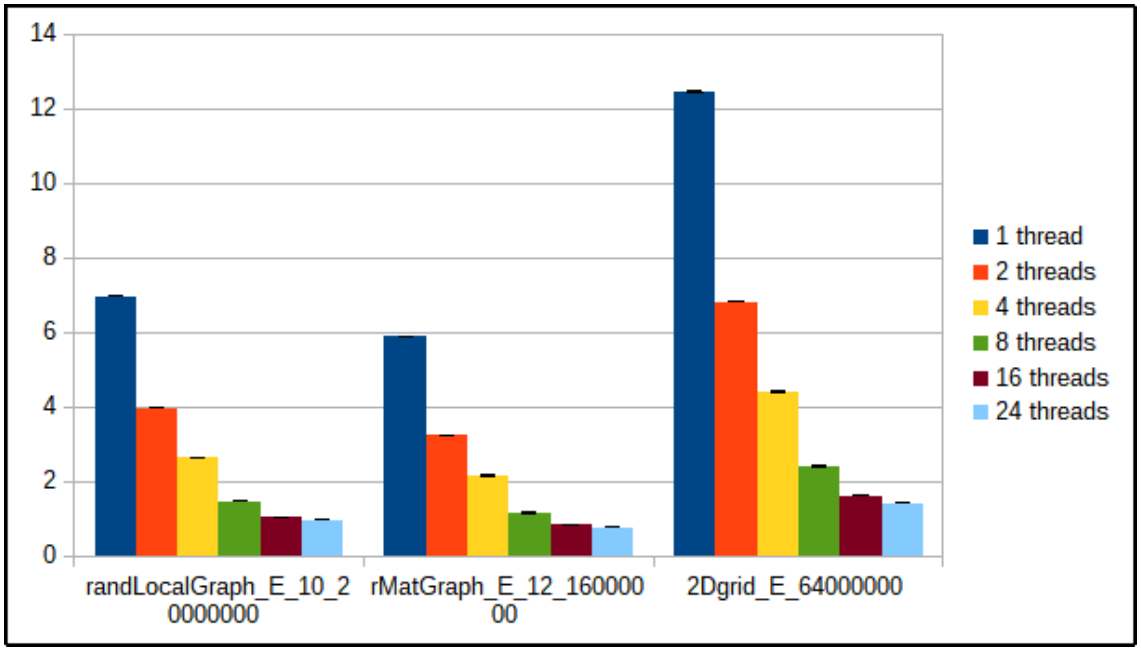


Figure A.188: Execution times (in minutes) of Spanning Forest ran on Intel 16-16 using SBLCWS with WShare (**first version**).

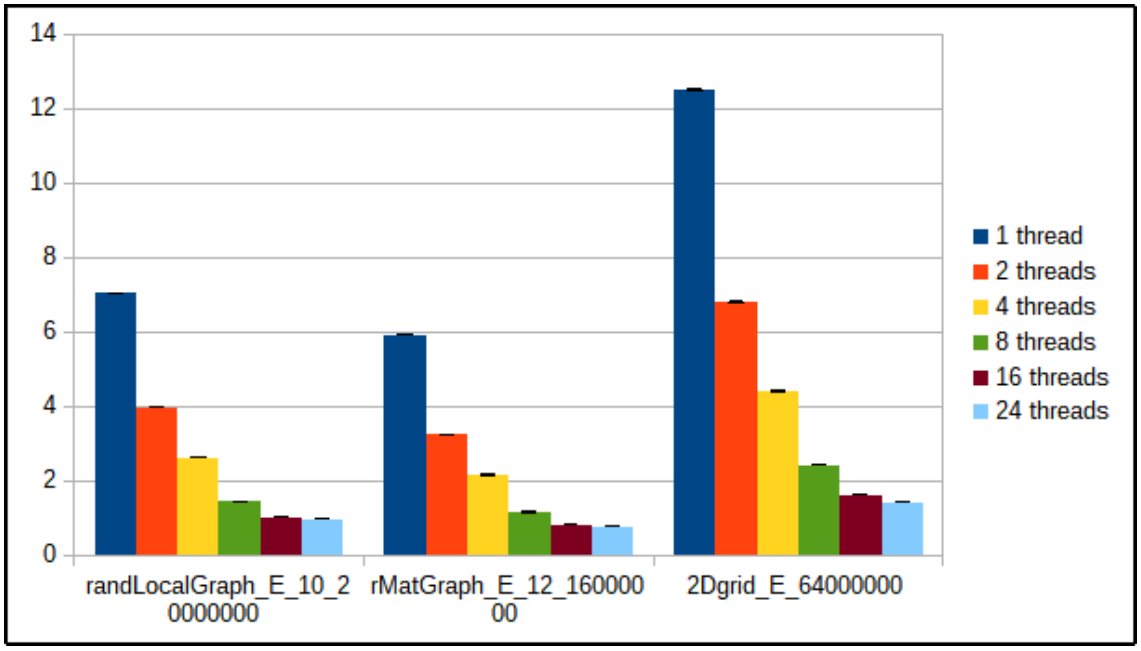


Figure A.189: Execution times (in minutes) of Spanning Forest ran on Intel 16-16 using SBLCWS with WShare (**second version**).

A.9 Breadth First Search

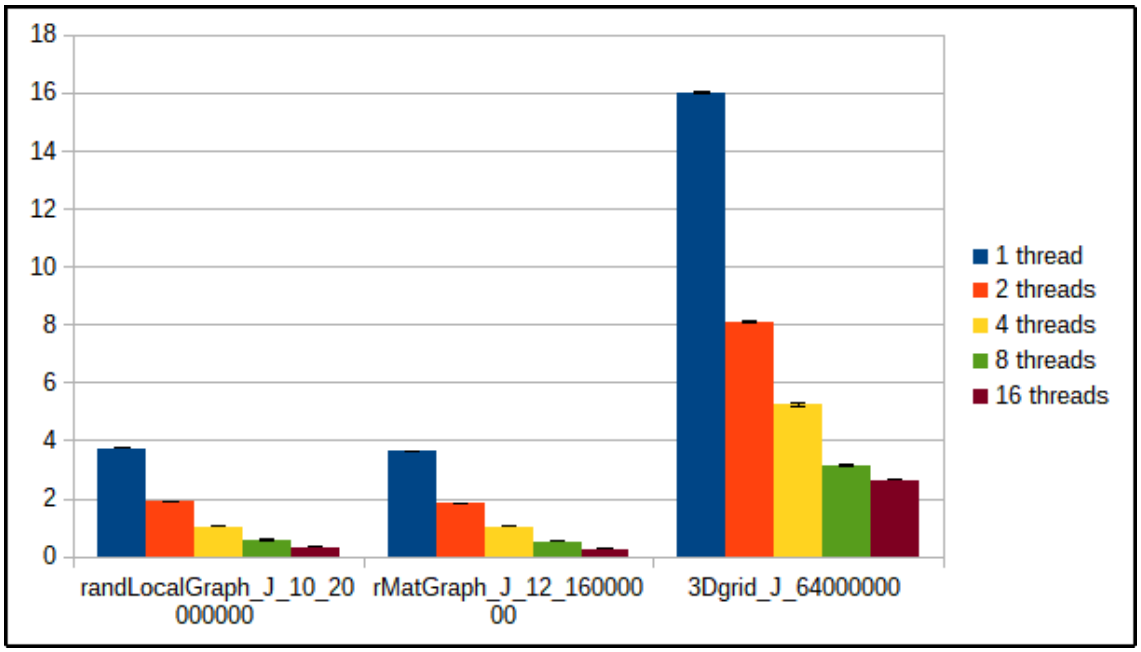


Figure A.190: Execution times (in minutes) of Breadth First Search ran on Intel 12-24 using WSteal

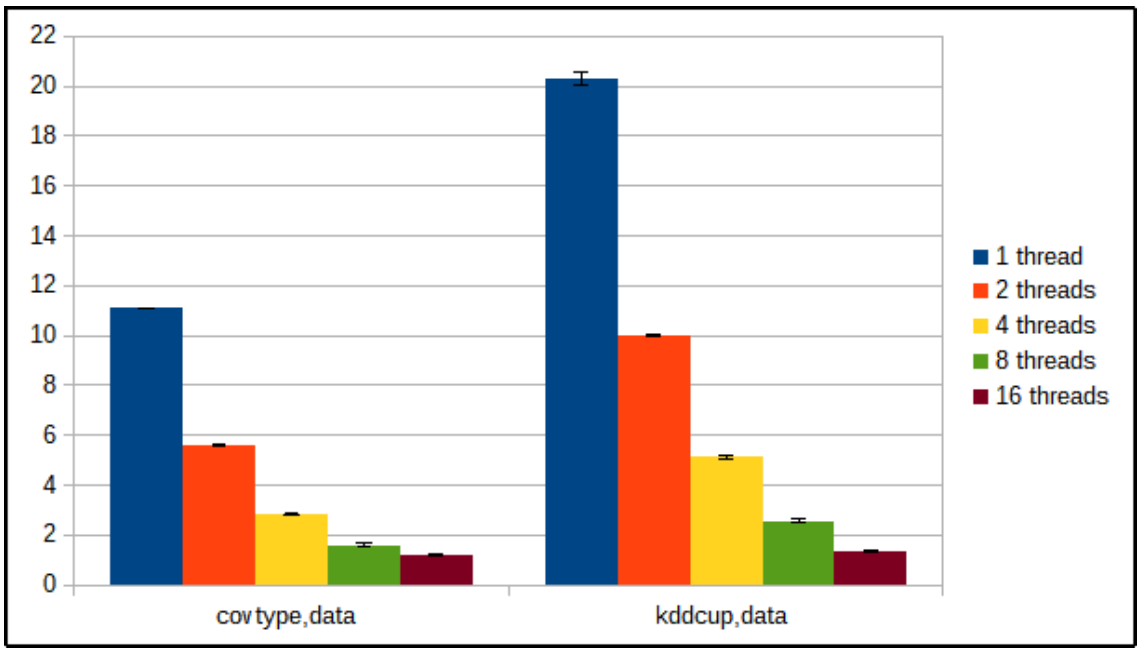


Figure A.191: Execution times (in minutes) of Breadth First Search ran on Intel 12-24 using LCWS

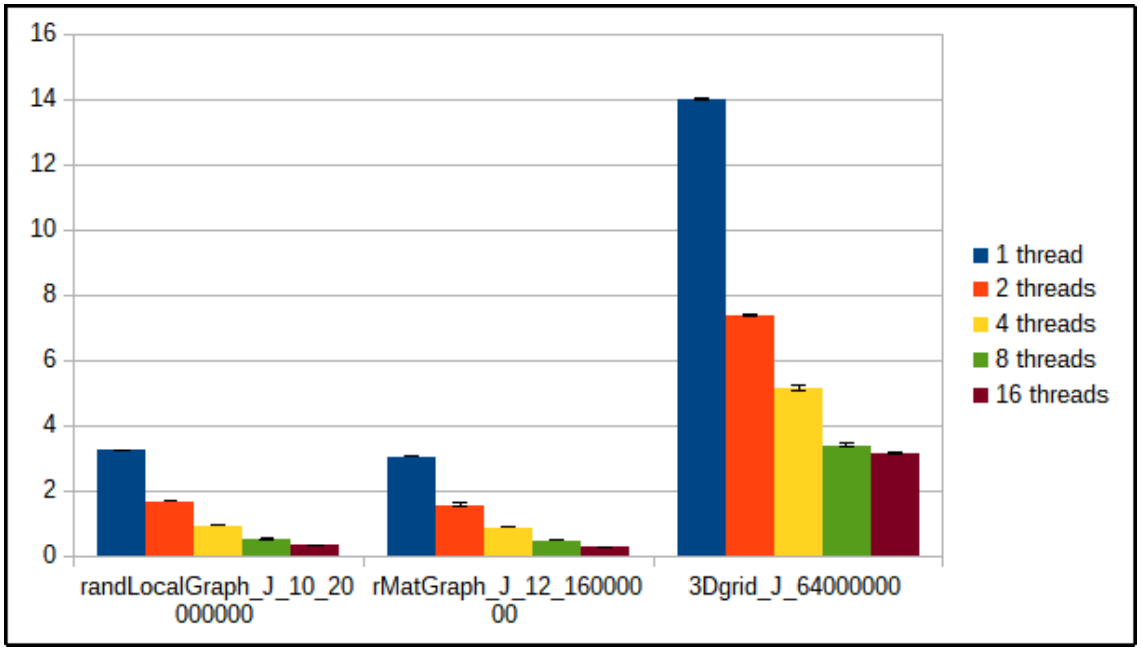


Figure A.192: Execution times (in minutes) of Breadth First Search ran on Intel 12-24 using SBLCWS (**first version**)

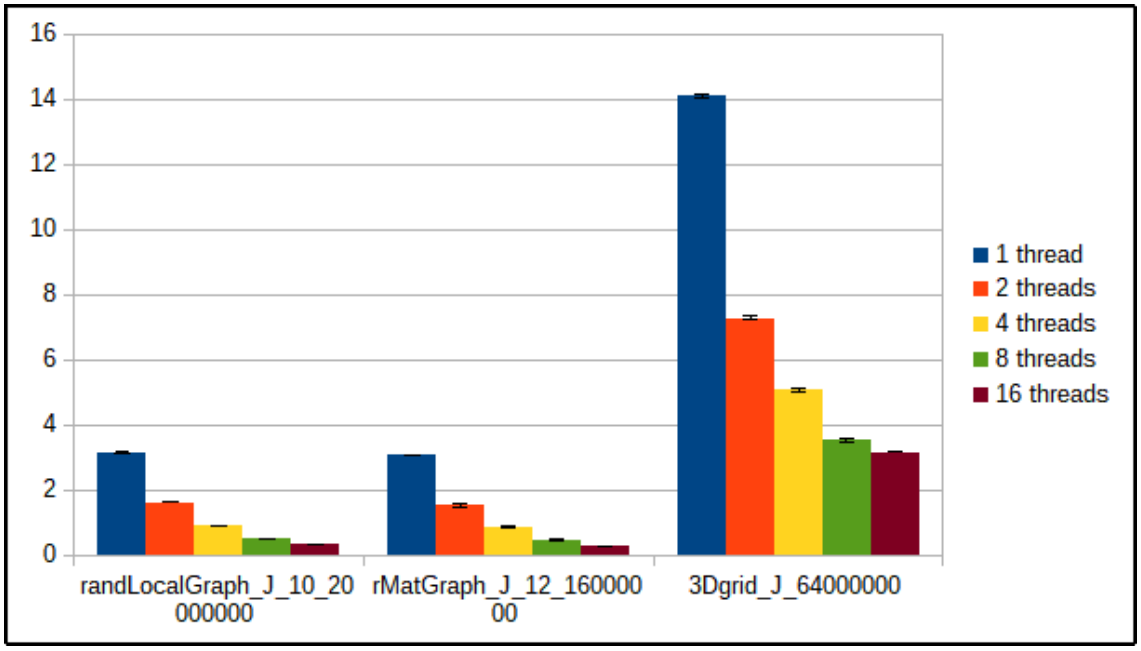


Figure A.193: Execution times (in minutes) of Breadth First Search ran on Intel 12-24 using SBLCWS (**second version**)

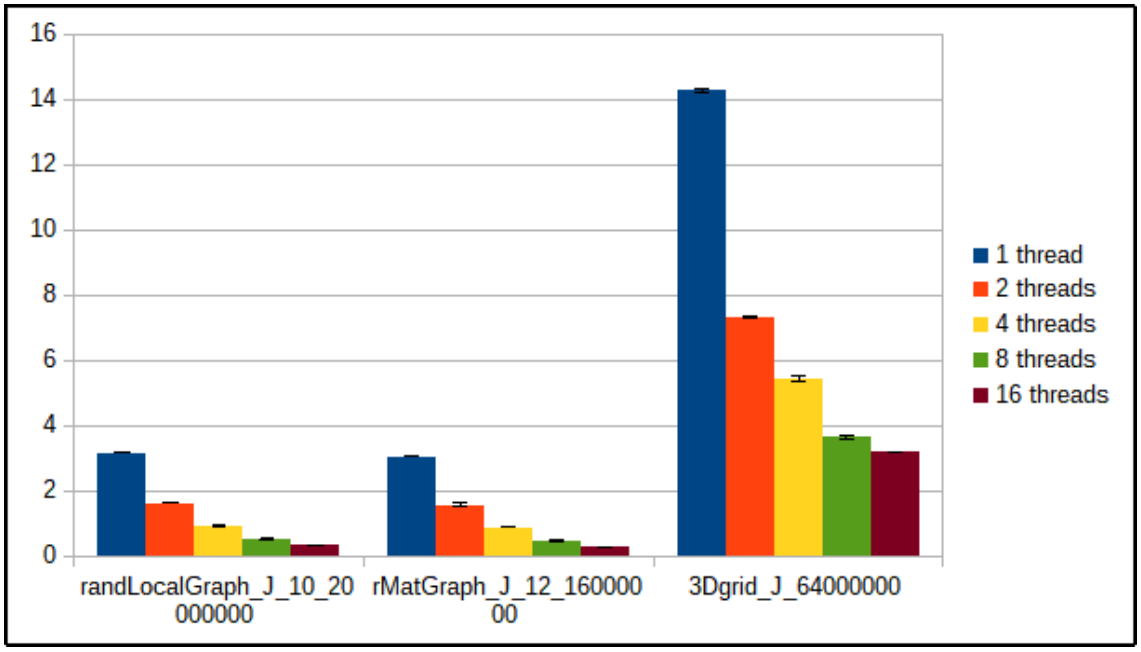


Figure A.194: Execution times (in minutes) of Breadth First Search ran on Intel 12-24 using LCWS with WShare (**tit-for-tat**).

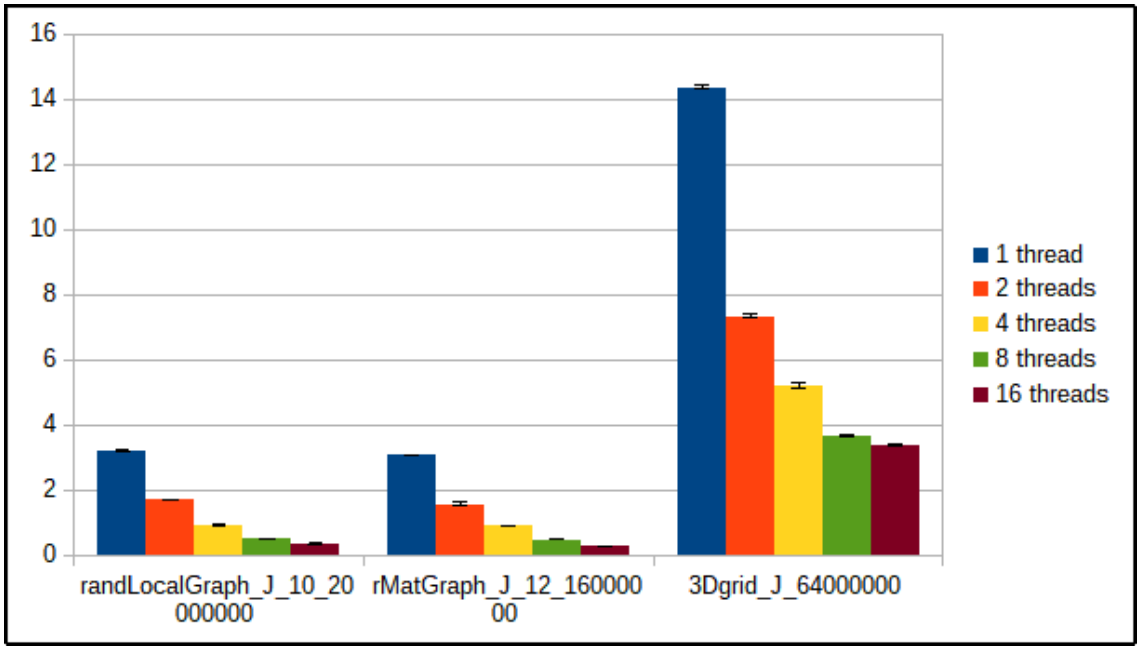


Figure A.195: Execution times (in minutes) of Breadth First Search ran on Intel 12-24 using LCWS with WShare (**continuous tit-for-tat**).

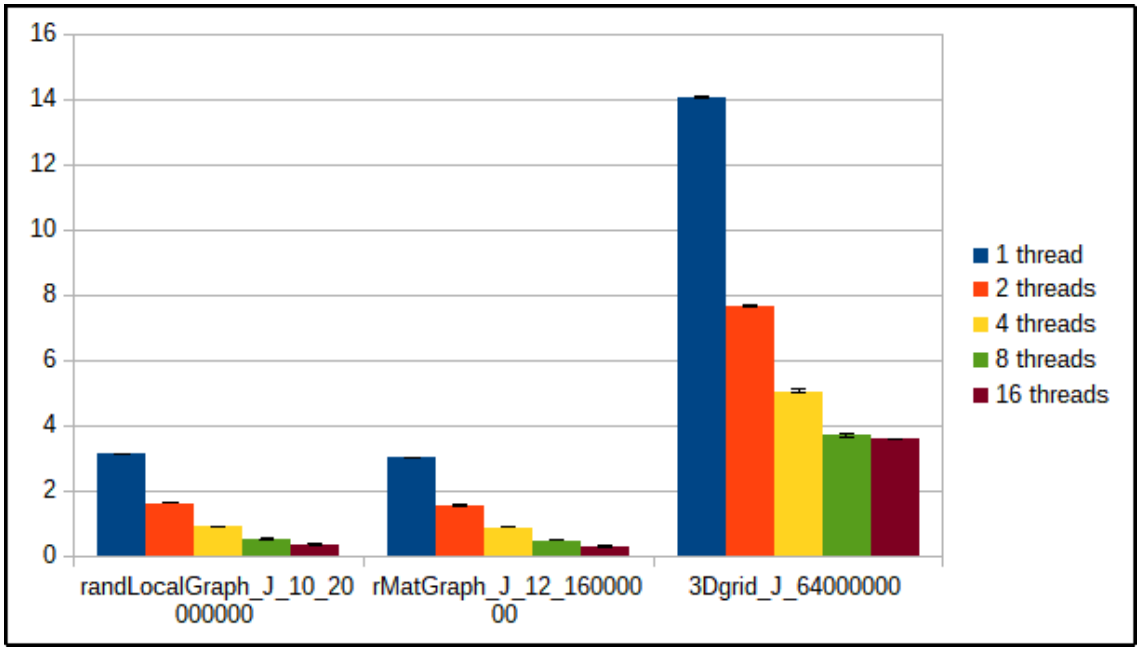


Figure A.196: Execution times (in minutes) of Breadth First Search ran on Intel 12-24 using SBLCWS with WShare (**first version**).

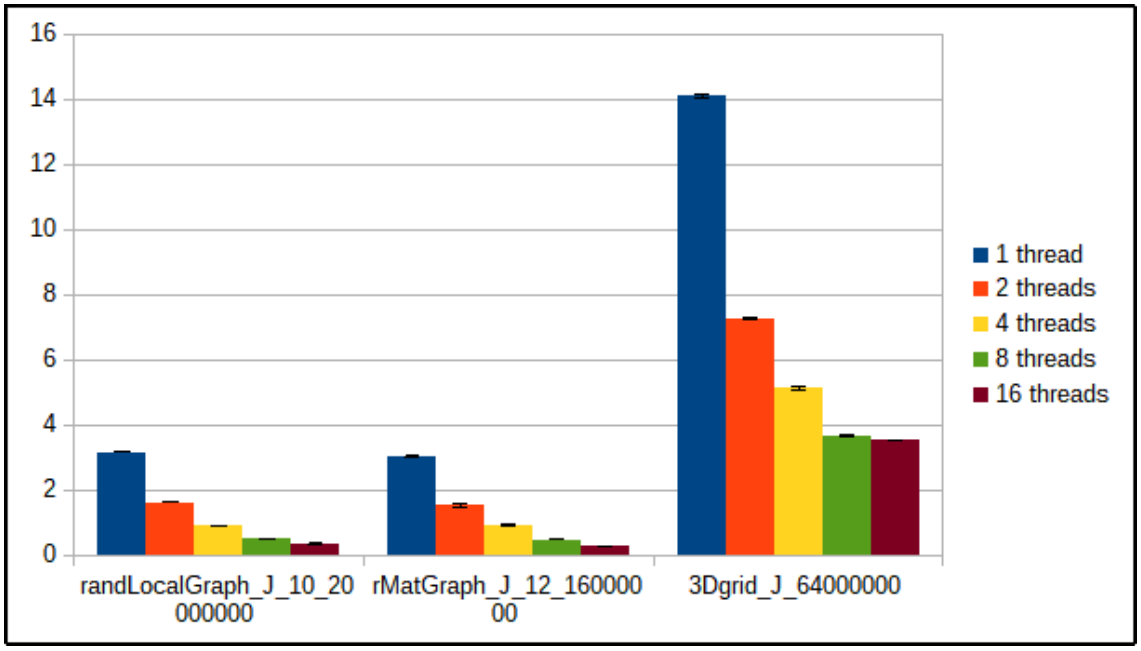


Figure A.197: Execution times (in minutes) of Breadth First Search ran on Intel 12-24 using SBLCWS with WShare (**second version**).

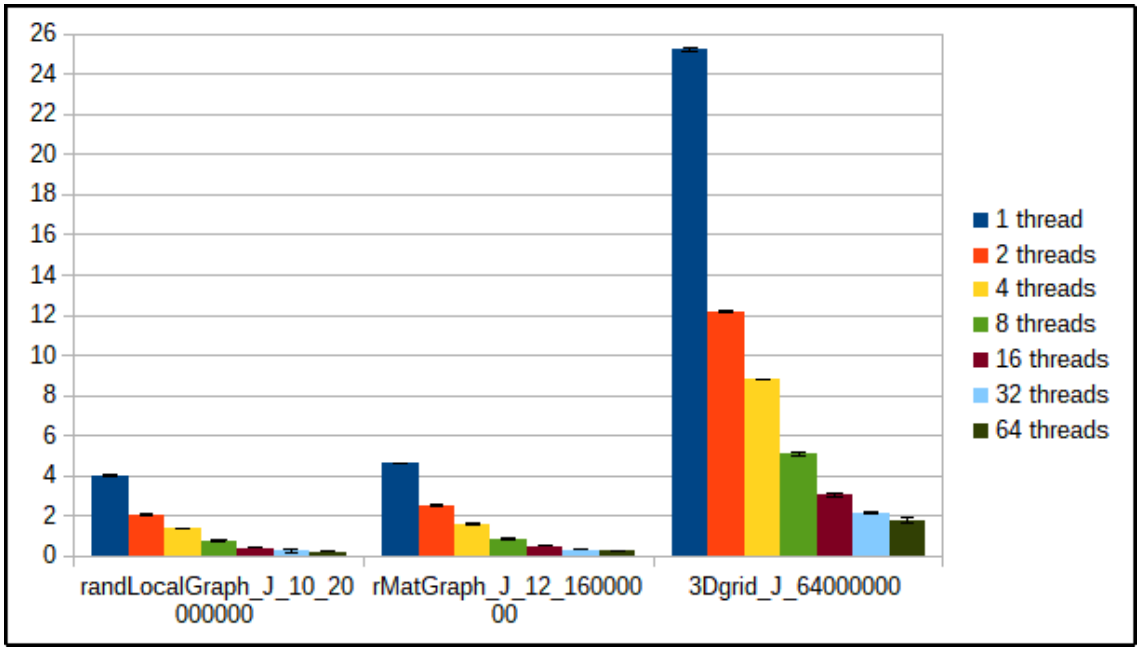


Figure A.198: Execution times (in minutes) of Breadth First Search ran on AMD 32-64 using WSteal

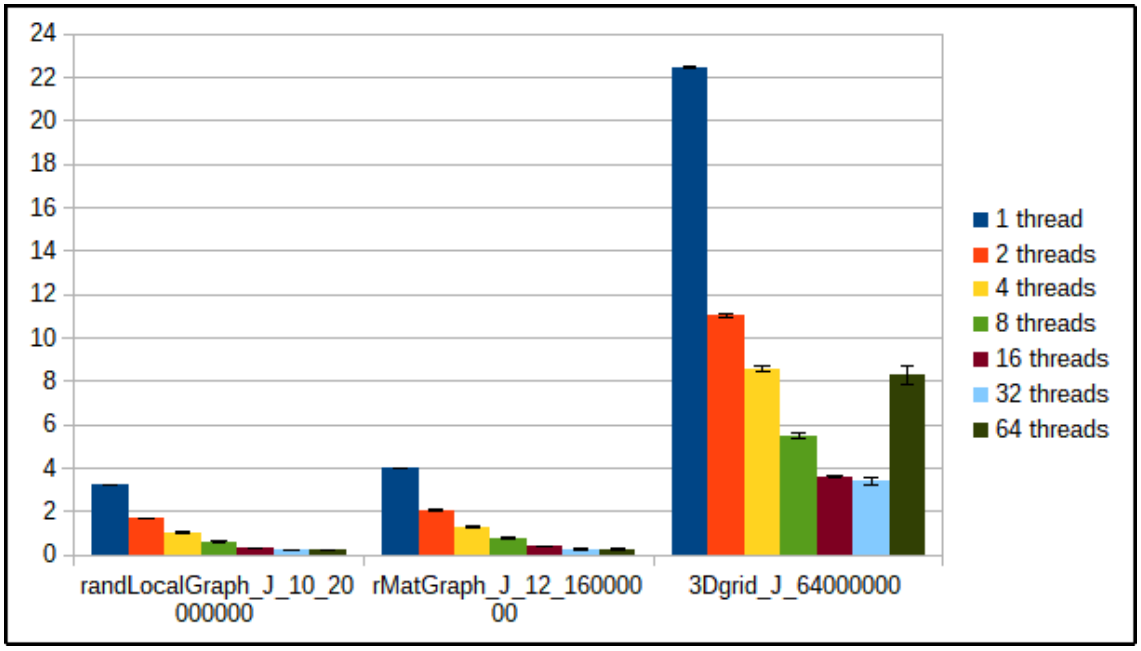


Figure A.199: Execution times (in minutes) of Breadth First Search ran on AMD 32-64 using LCWS

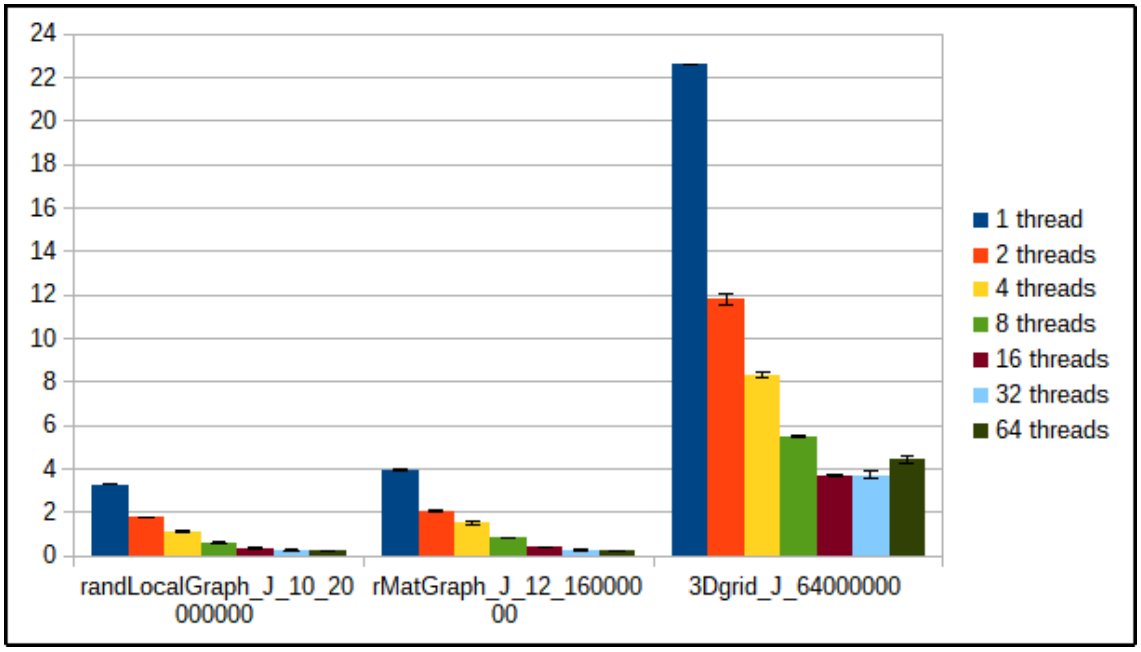


Figure A.200: Execution times (in minutes) of Breadth First Search ran on AMD 32-64 using SBLCWS (**first version**)

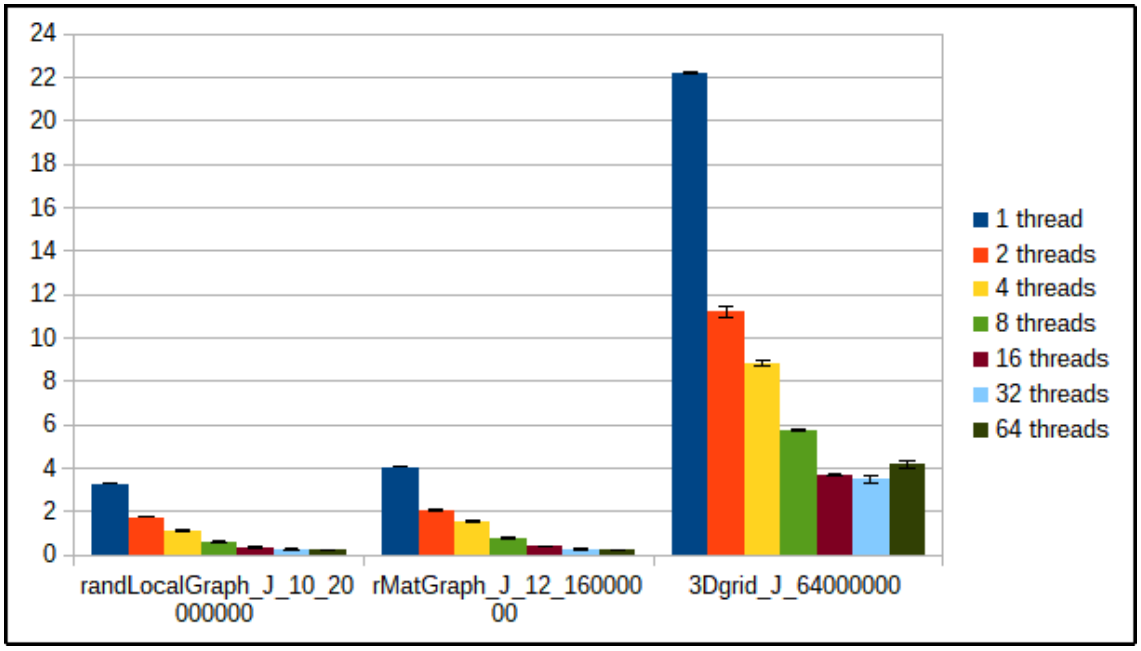


Figure A.201: Execution times (in minutes) of Breadth First Search ran on AMD 32-64 using SBLCWS (**second version**)

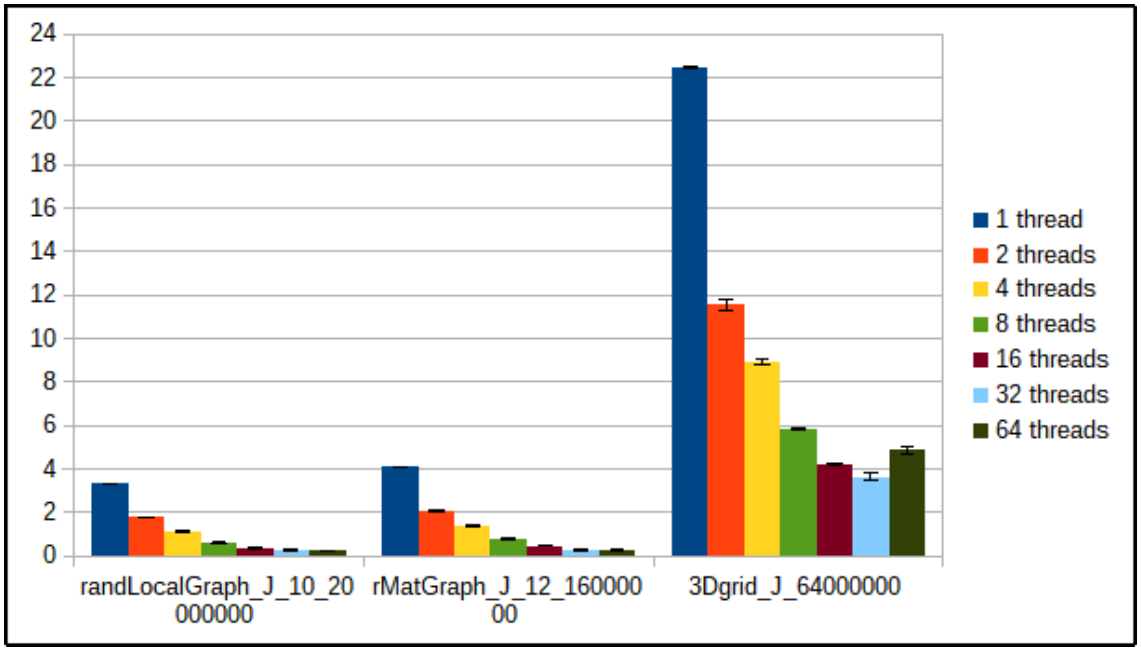


Figure A.202: Execution times (in minutes) of Breadth First Search ran on AMD 32-64 using LCWS with WShare (**tit-for-tat**).

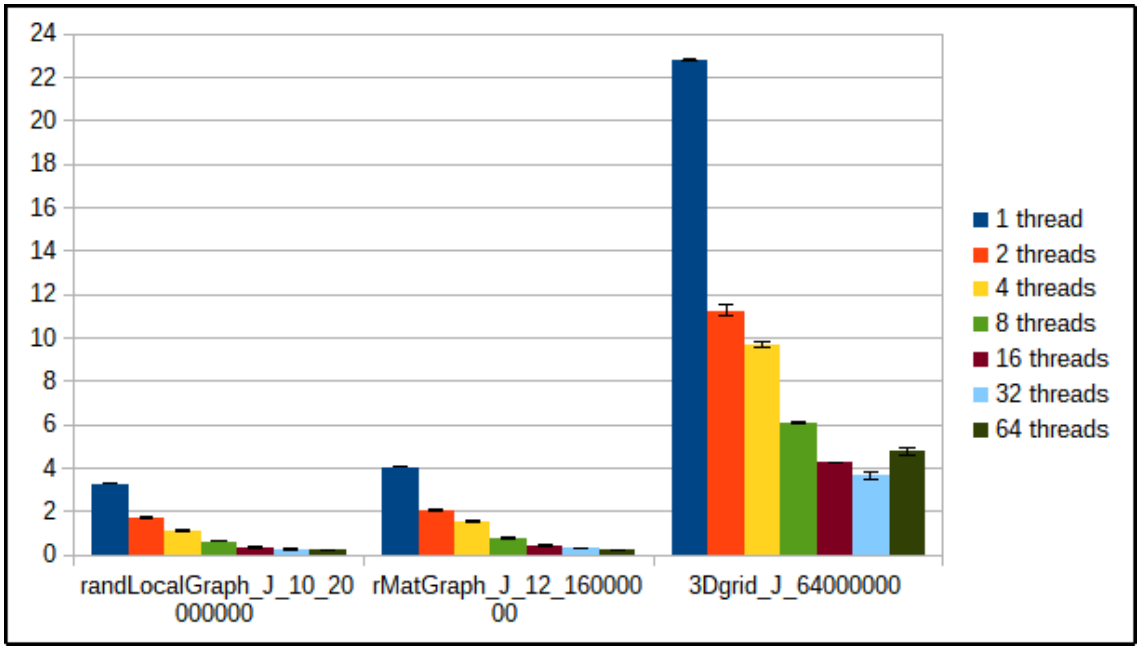


Figure A.203: Execution times (in minutes) of Breadth First Search ran on AMD 32-64 using LCWS with WShare (**continuous tit-for-tat**).

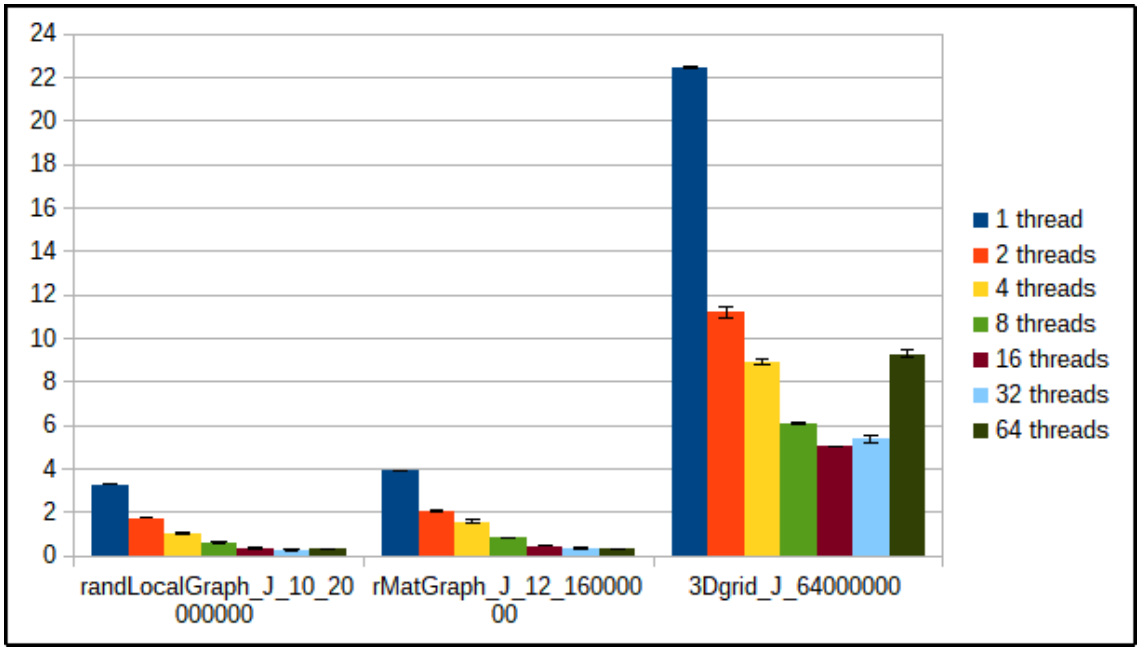


Figure A.204: Execution times (in minutes) of Breadth First Search ran on AMD 32-64 using SBLCWS with WShare (**first version**).

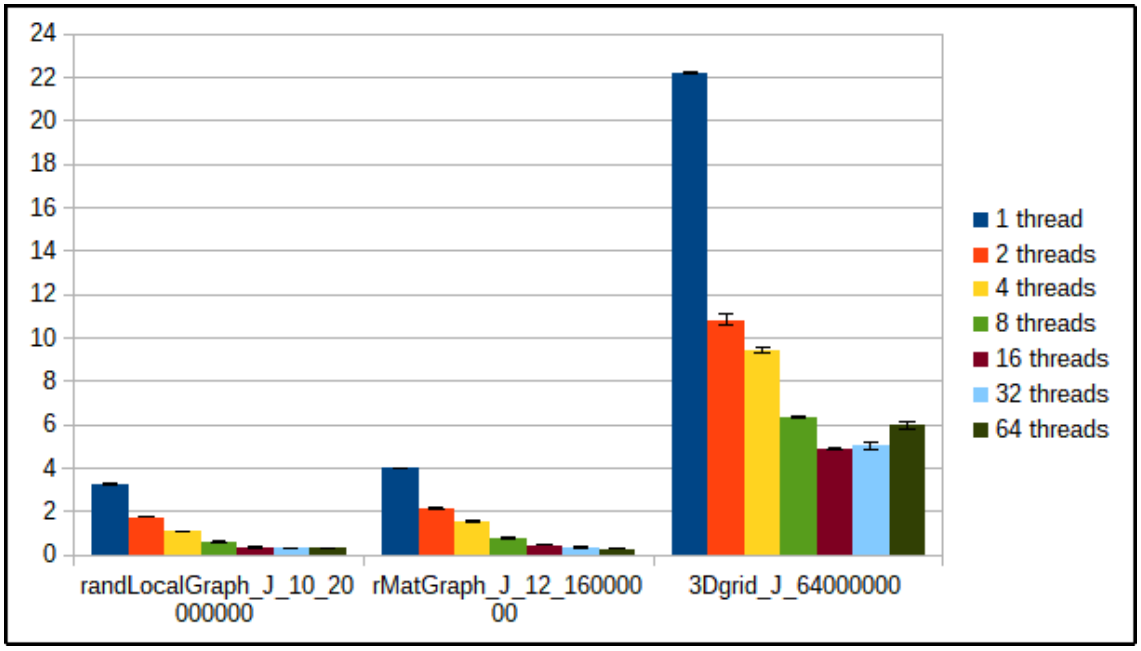


Figure A.205: Execution times (in minutes) of Breadth First Search ran on AMD 32-64 using SBLCWS with WShare (**second version**).

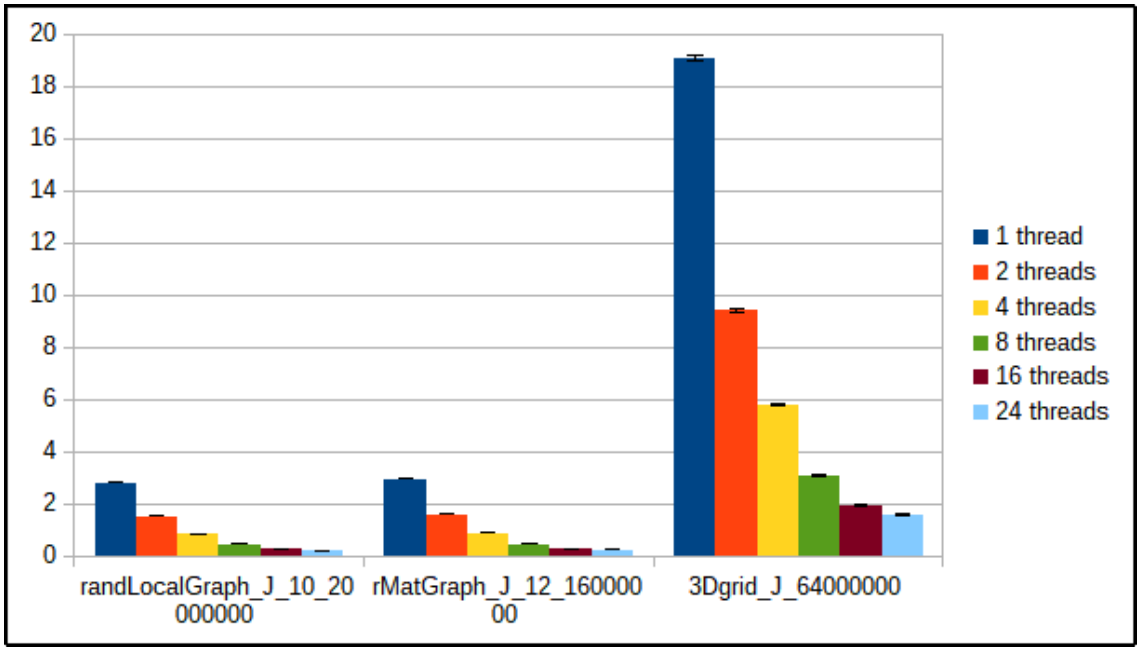


Figure A.206: Execution times (in minutes) of Breadth First Search ran on Intel 16-16 using WSteal

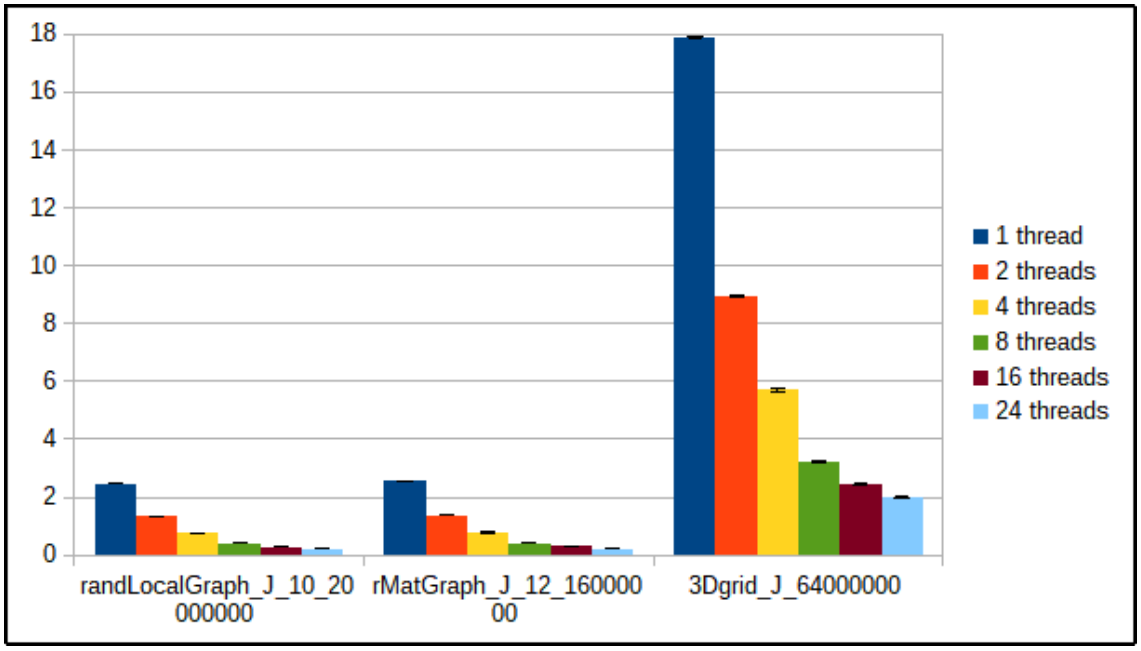


Figure A.207: Execution times (in minutes) of Breadth First Search ran on Intel 16-16 using LCWS

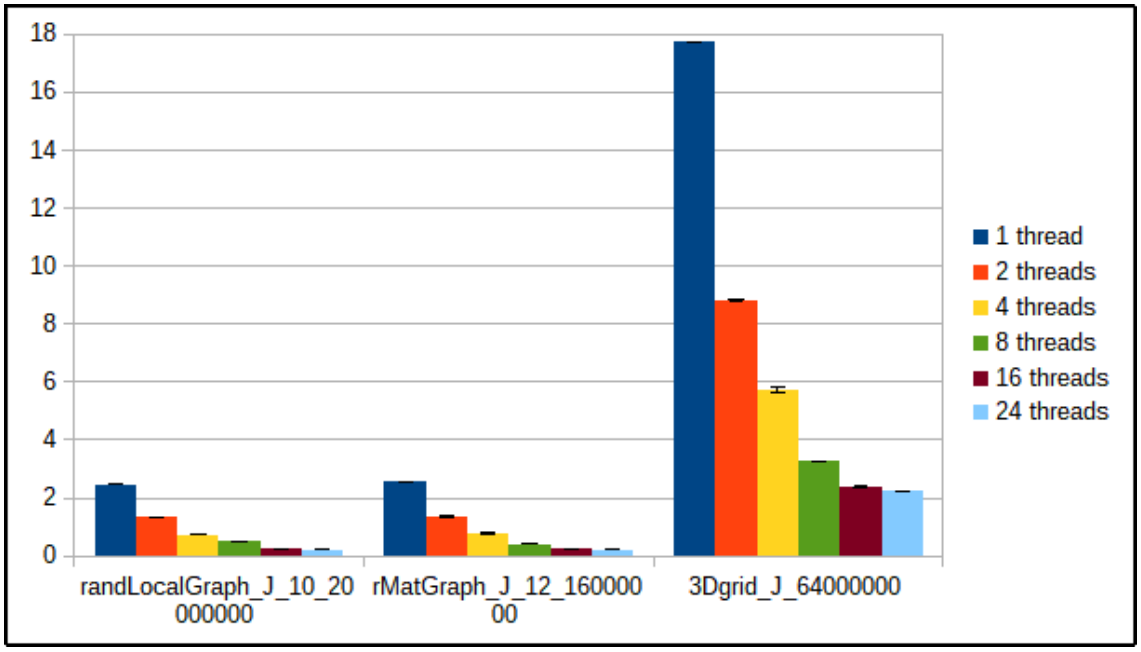


Figure A.208: Execution times (in minutes) of Breadth First Search ran on Intel 16-16 using SBLCWS (**first version**)

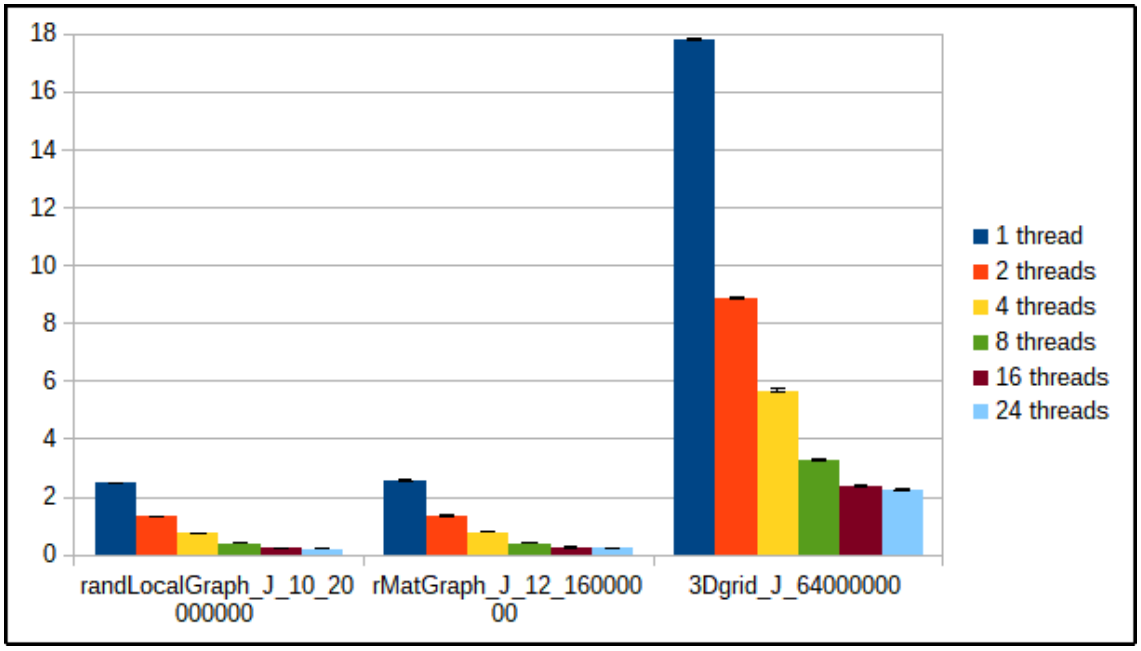


Figure A.209: Execution times (in minutes) of Breadth First Search ran on Intel 16-16 using SBLCWS (**second version**)

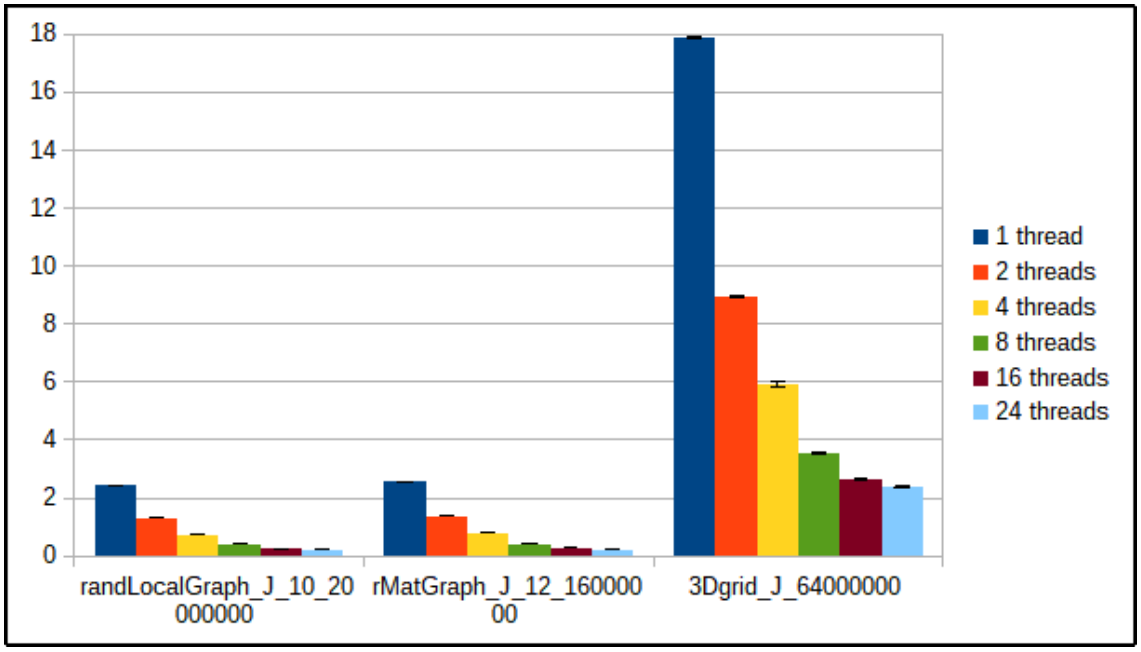


Figure A.210: Execution times (in minutes) of Breadth First Search ran on Intel 16-16 using LCWS with WShare (**tit-for-tat**).

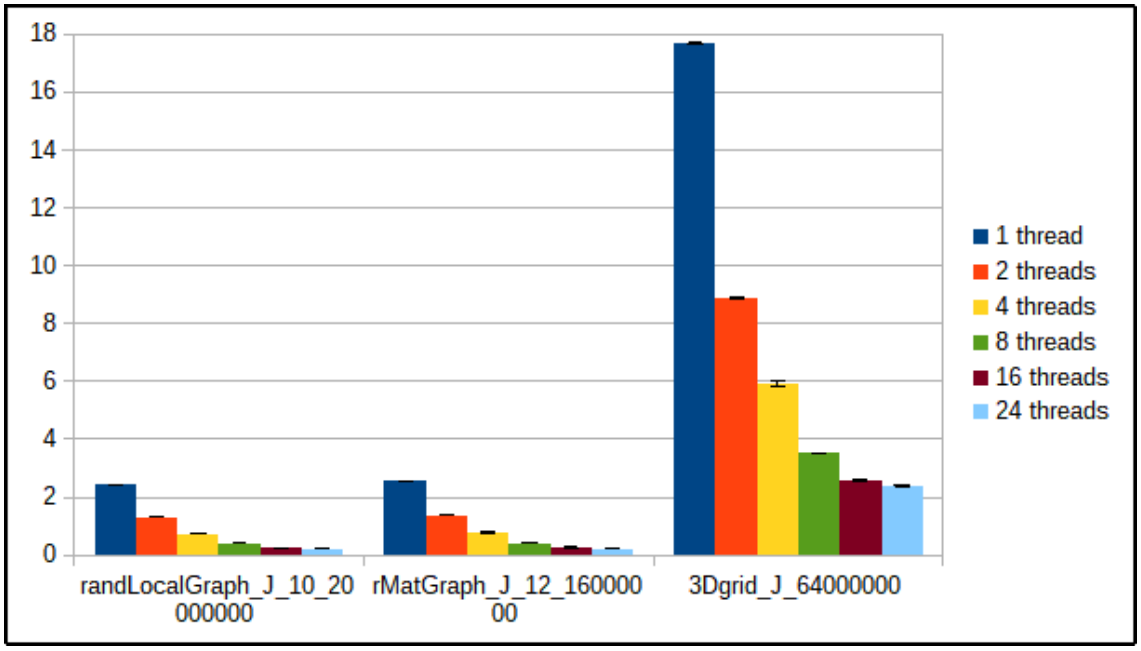


Figure A.211: Execution times (in minutes) of Breadth First Search ran on Intel 16-16 using LCWS with WShare (**continuous tit-for-tat**).

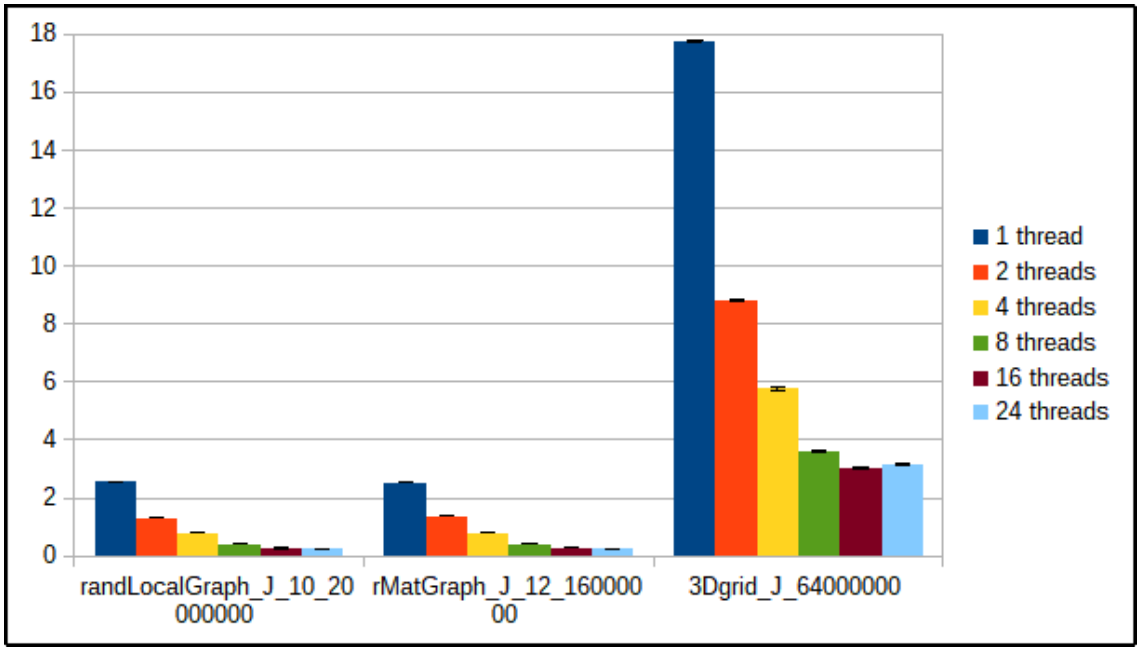


Figure A.212: Execution times (in minutes) of Breadth First Search ran on Intel 16-16 using SBLCWS with WShare (**first version**).

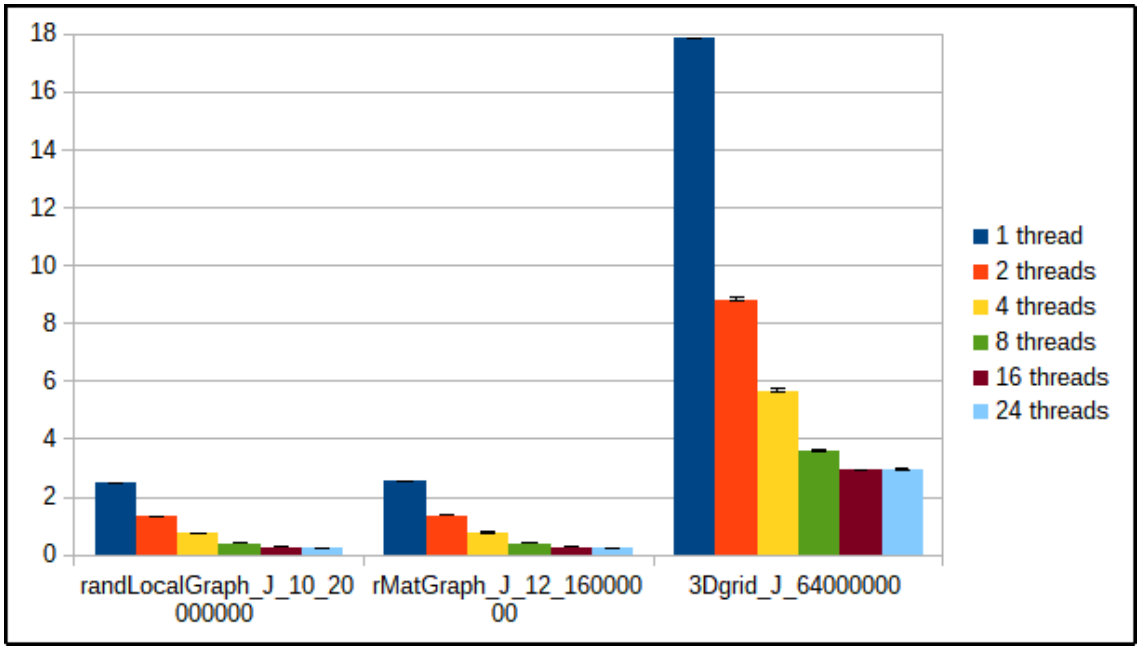


Figure A.213: Execution times (in minutes) of Breadth First Search ran on Intel 16-16 using SBLCWS with WShare (**second version**).

A.10 Maximal Matching

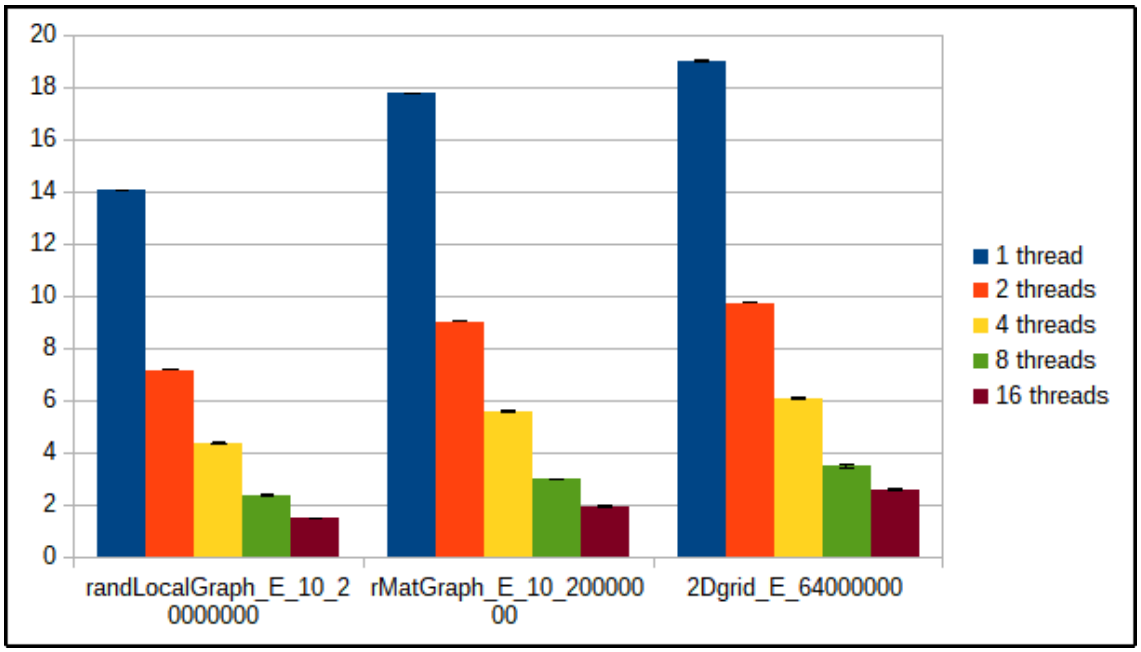


Figure A.214: Execution times (in minutes) of Maximal Matching ran on Intel 12-24 using WSteal

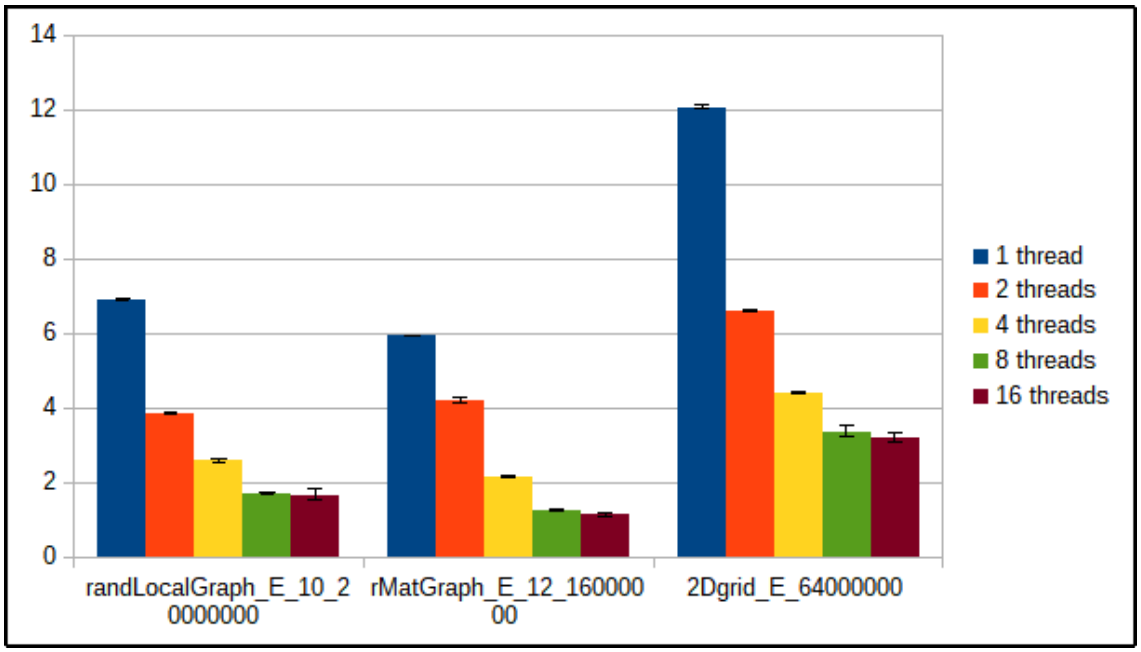


Figure A.215: Execution times (in minutes) of Maximal Matching ran on Intel 12-24 using LCWS

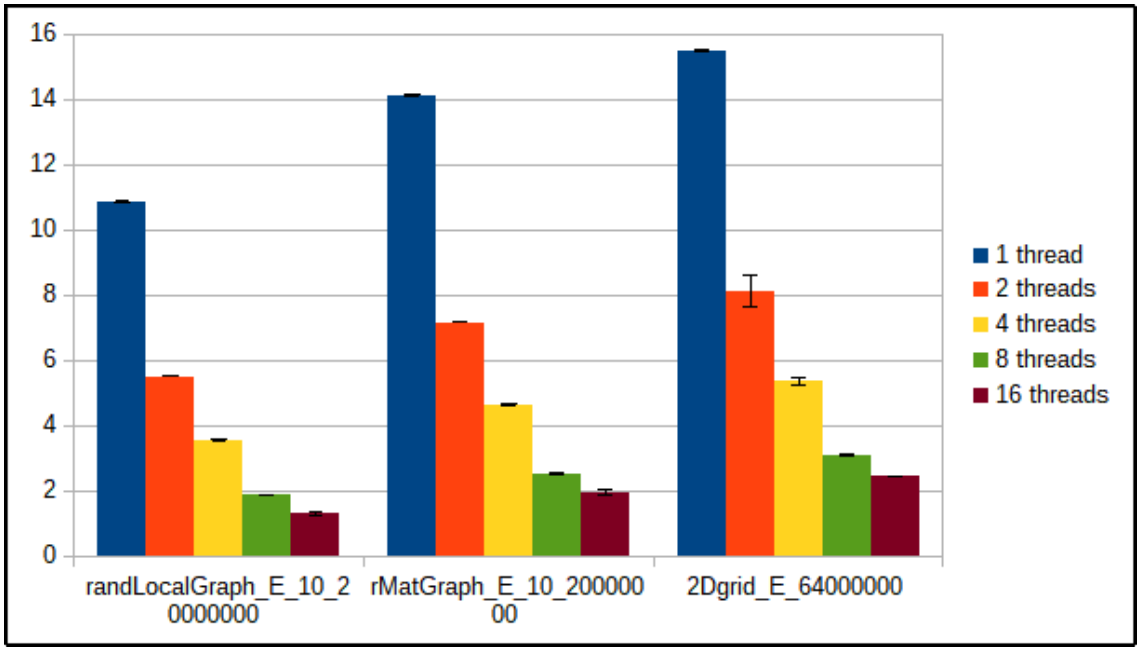


Figure A.216: Execution times (in minutes) of Maximal Matching ran on Intel 12-24 using SBLCWS (**first version**)

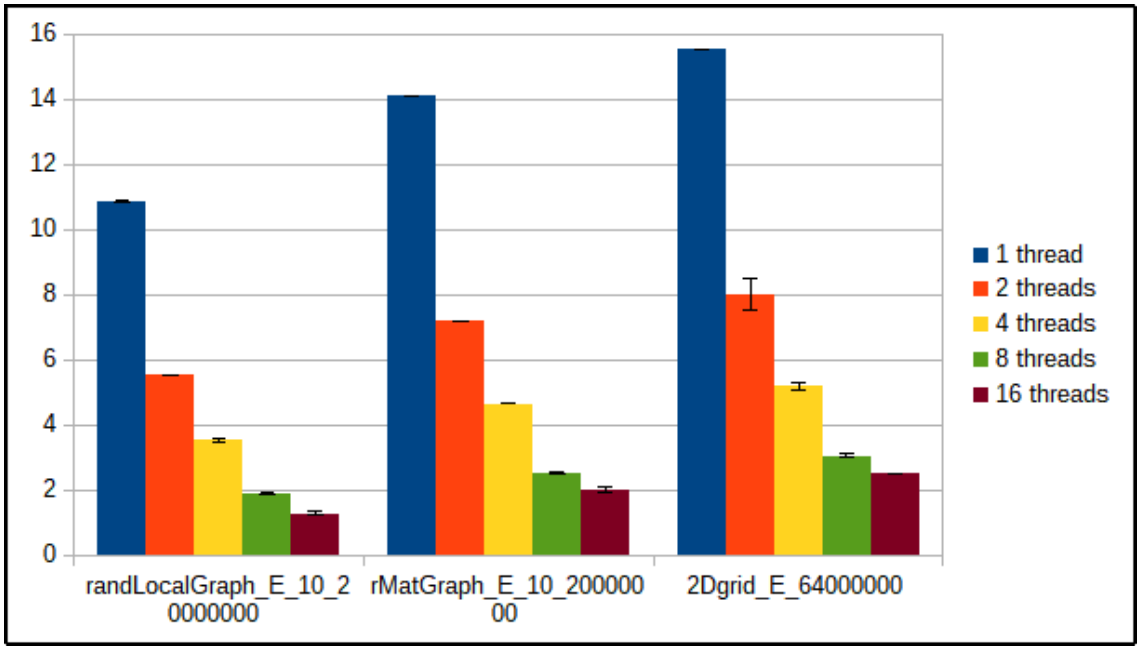


Figure A.217: Execution times (in minutes) of Maximal Matching ran on Intel 12-24 using SBLCWS (**second version**)

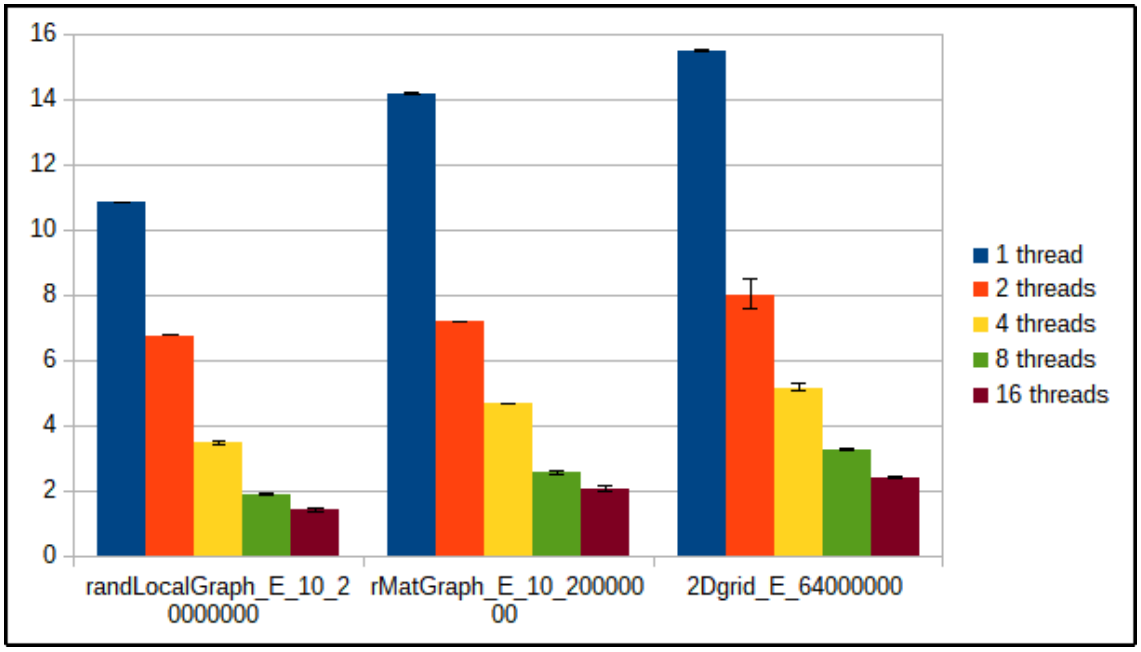


Figure A.218: Execution times (in minutes) of Maximal Matching ran on Intel 12-24 using LCWS with WShare (**tit-for-tat**).

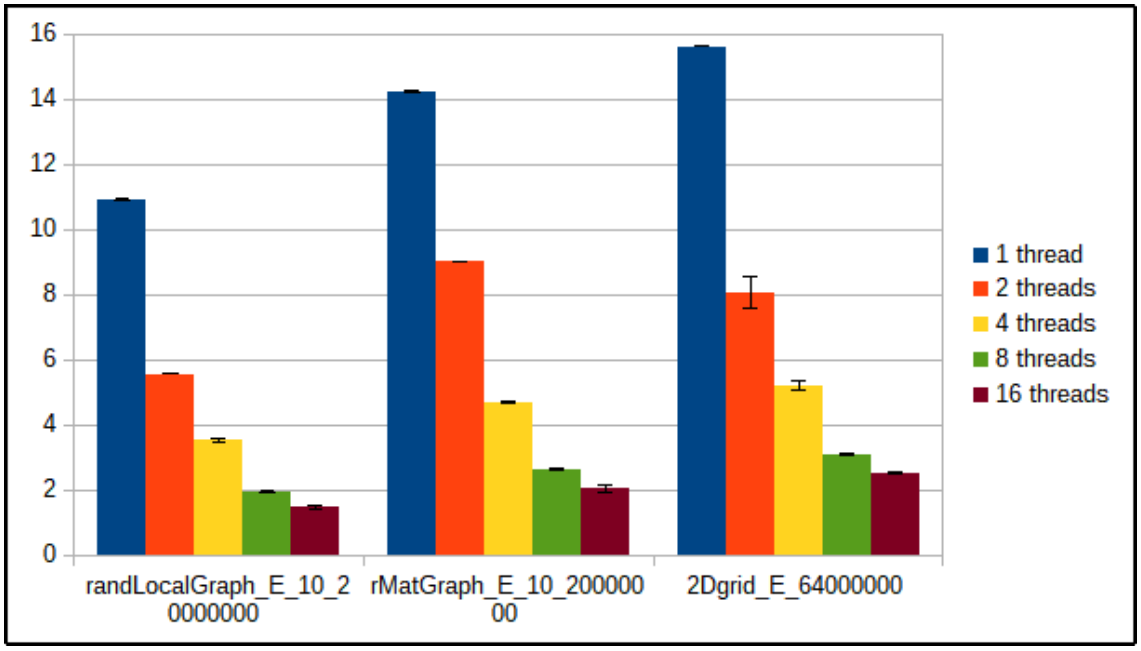


Figure A.219: Execution times (in minutes) of Maximal Matching ran on Intel 12-24 using LCWS with WShare (**continuous tit-for-tat**).

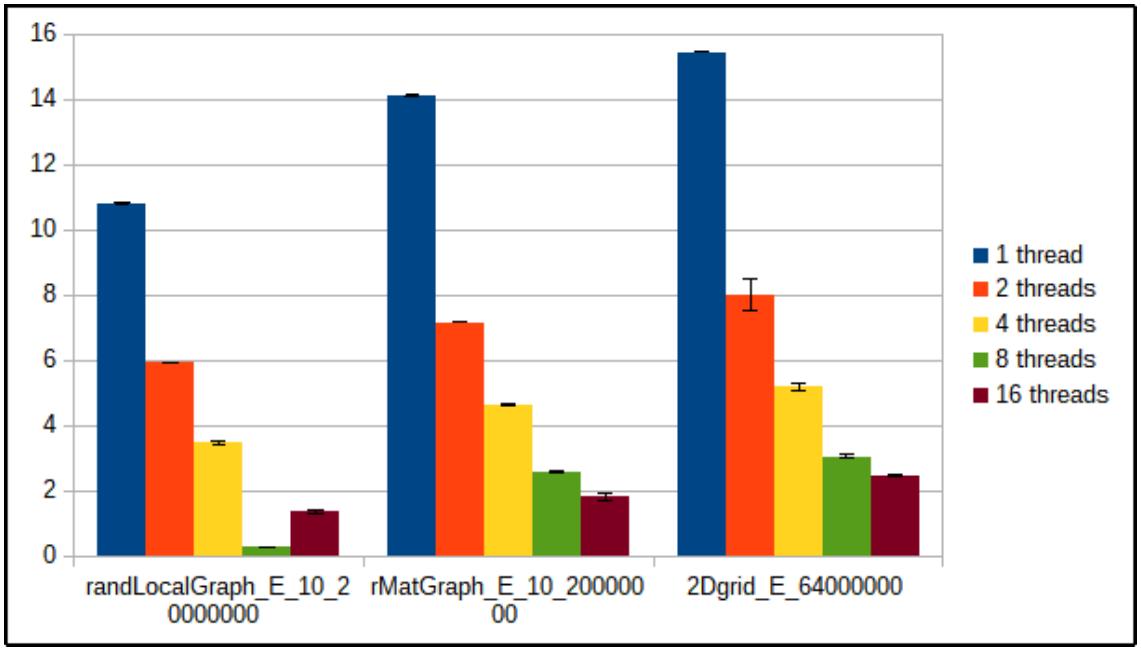


Figure A.220: Execution times (in minutes) of Maximal Matching ran on Intel 12-24 using SBLCWS with WShare (**first version**).

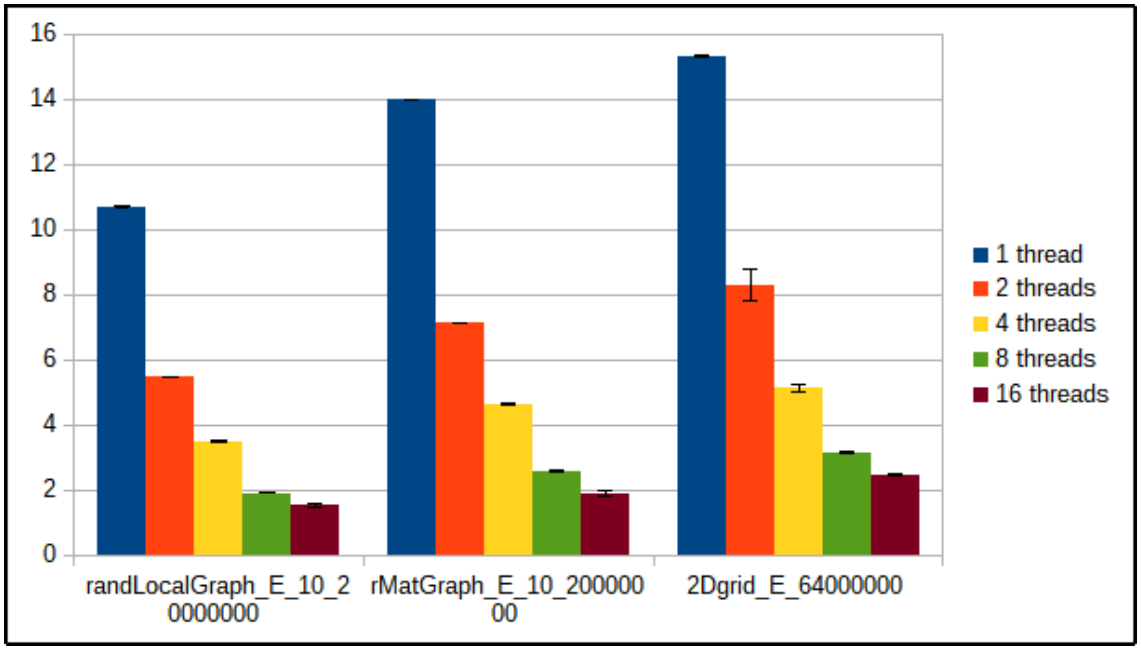


Figure A.221: Execution times (in minutes) of Maximal Matching ran on Intel 12-24 using SBLCWS with WShare (**second version**).

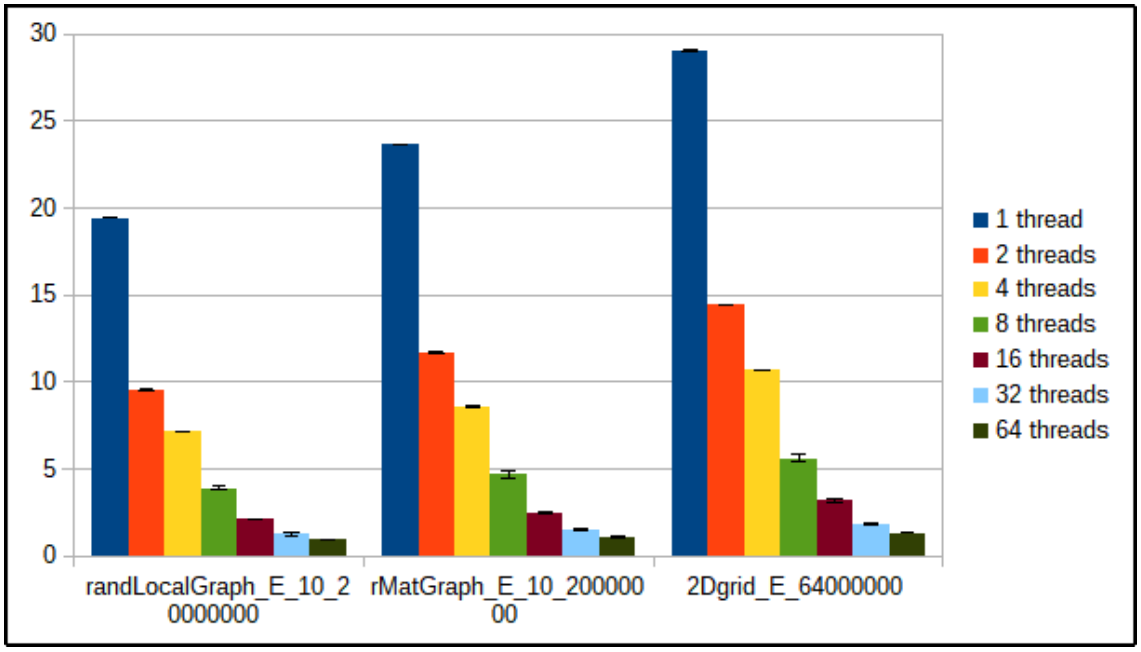


Figure A.222: Execution times (in minutes) of Maximal Matching ran on AMD 32-64 using WSteal

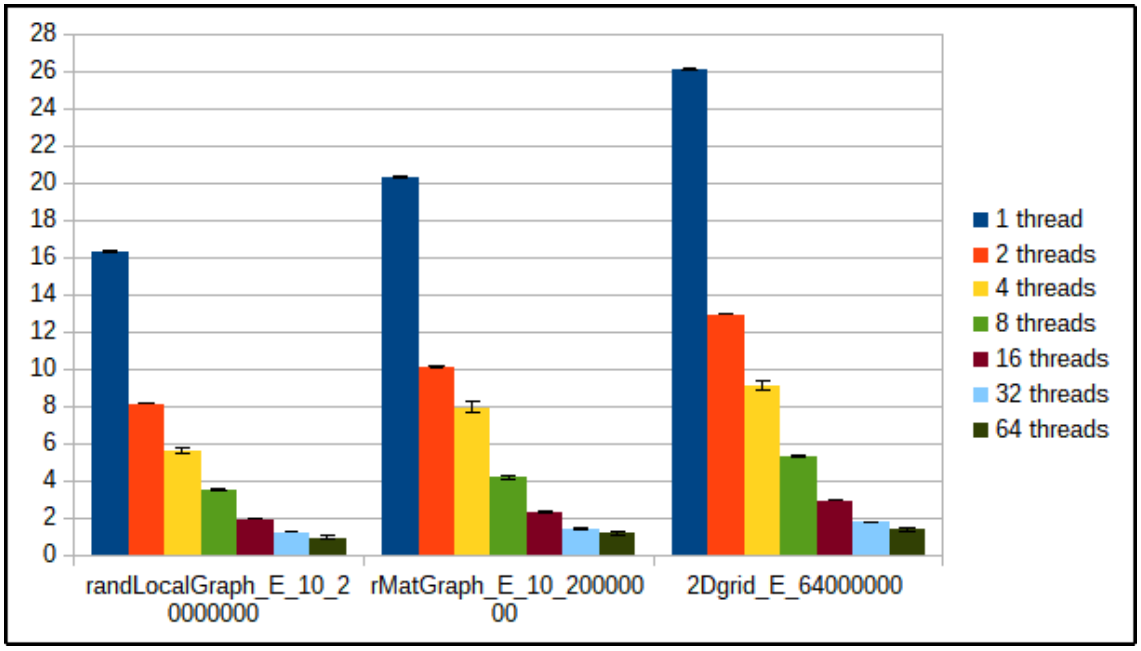


Figure A.223: Execution times (in minutes) of Maximal Matching ran on AMD 32-64 using LCWS

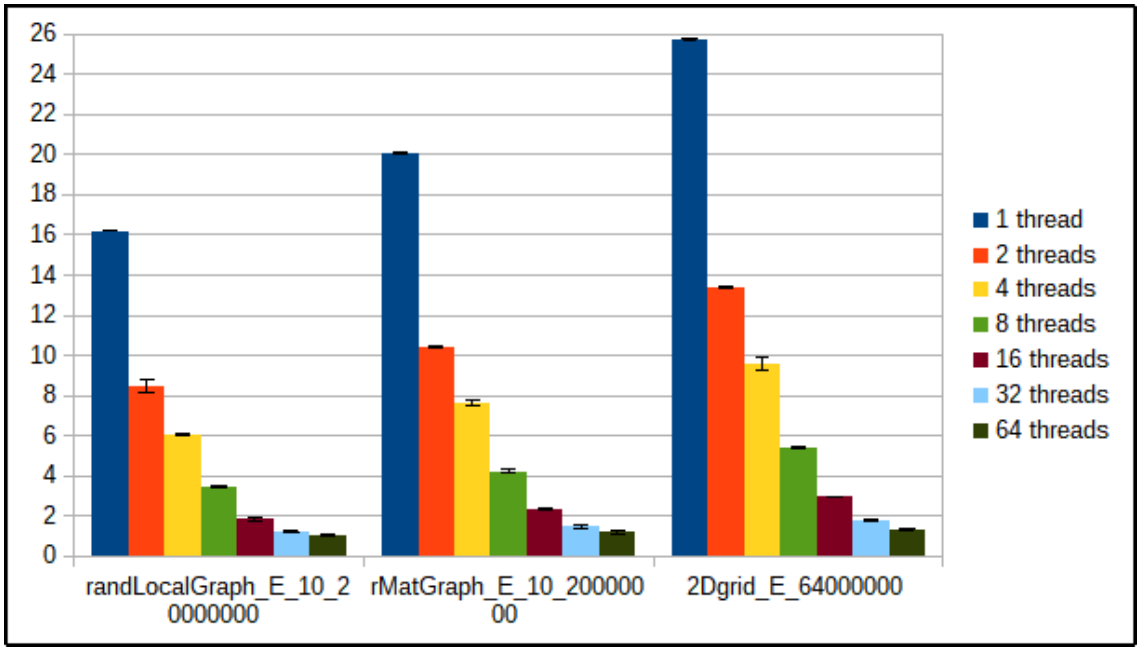


Figure A.224: Execution times (in minutes) of Maximal Matching ran on AMD 32-64 using SBLCWS (**first version**)

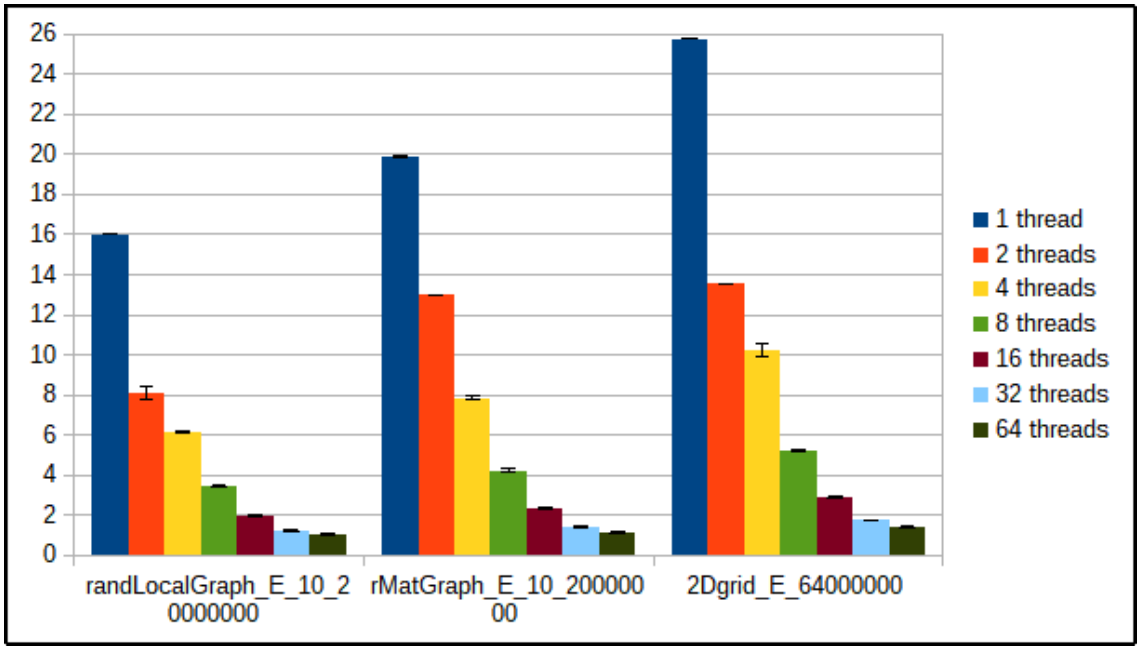


Figure A.225: Execution times (in minutes) of Maximal Matching ran on AMD 32-64 using SBLCWS (**second version**)

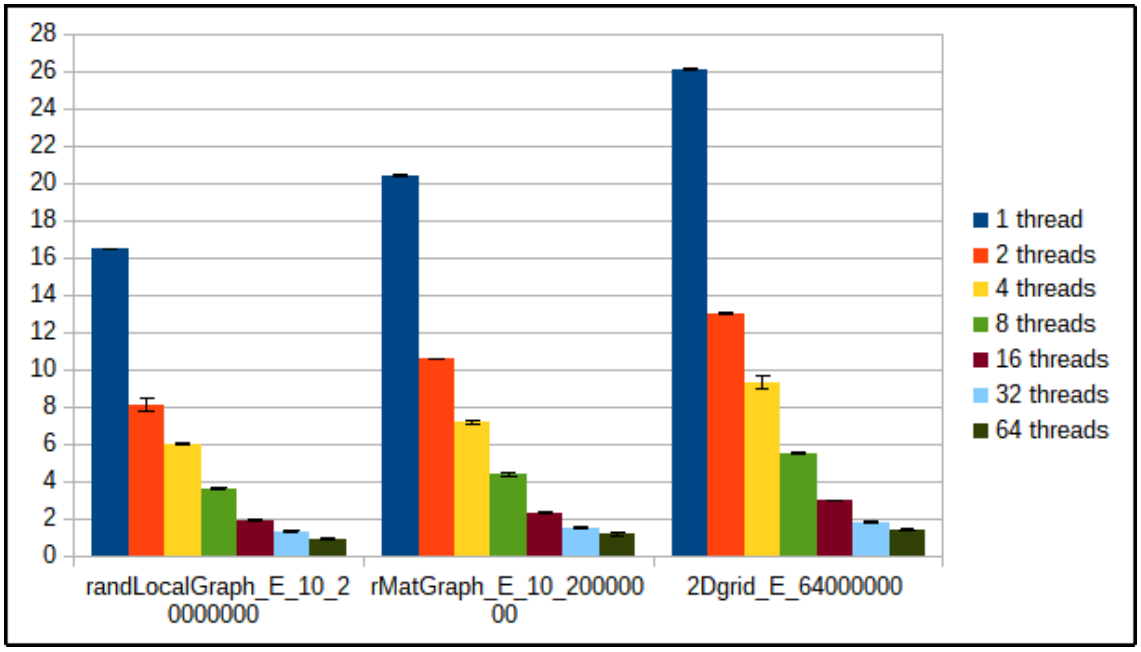


Figure A.226: Execution times (in minutes) of Maximal Matching ran on AMD 32-64 using LCWS with WShare (**tit-for-tat**).

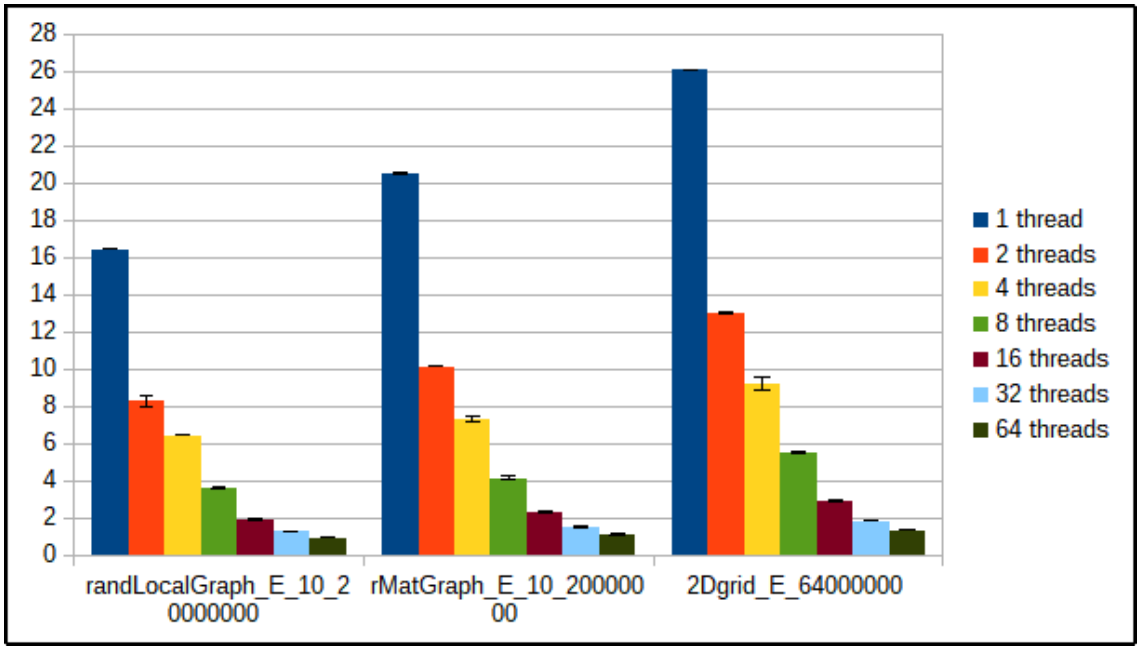


Figure A.227: Execution times (in minutes) of Maximal Matching ran on AMD 32-64 using LCWS with WShare (**continuous tit-for-tat**).

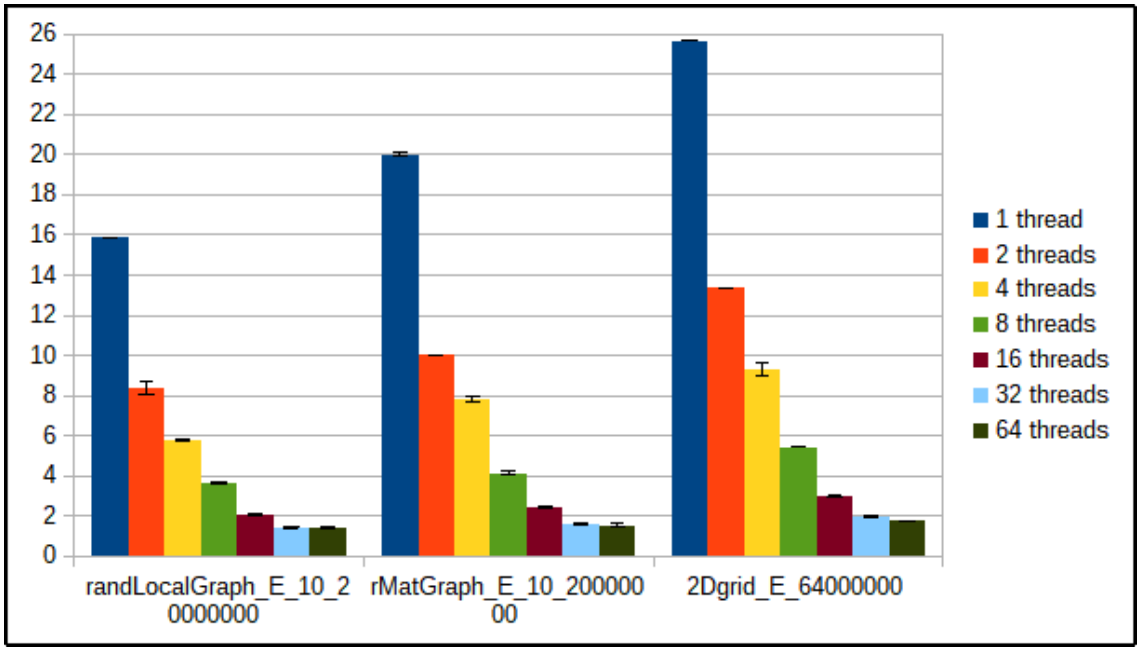


Figure A.228: Execution times (in minutes) of Maximal Matching ran on AMD 32-64 using SBLCWS with WShare (**first version**).

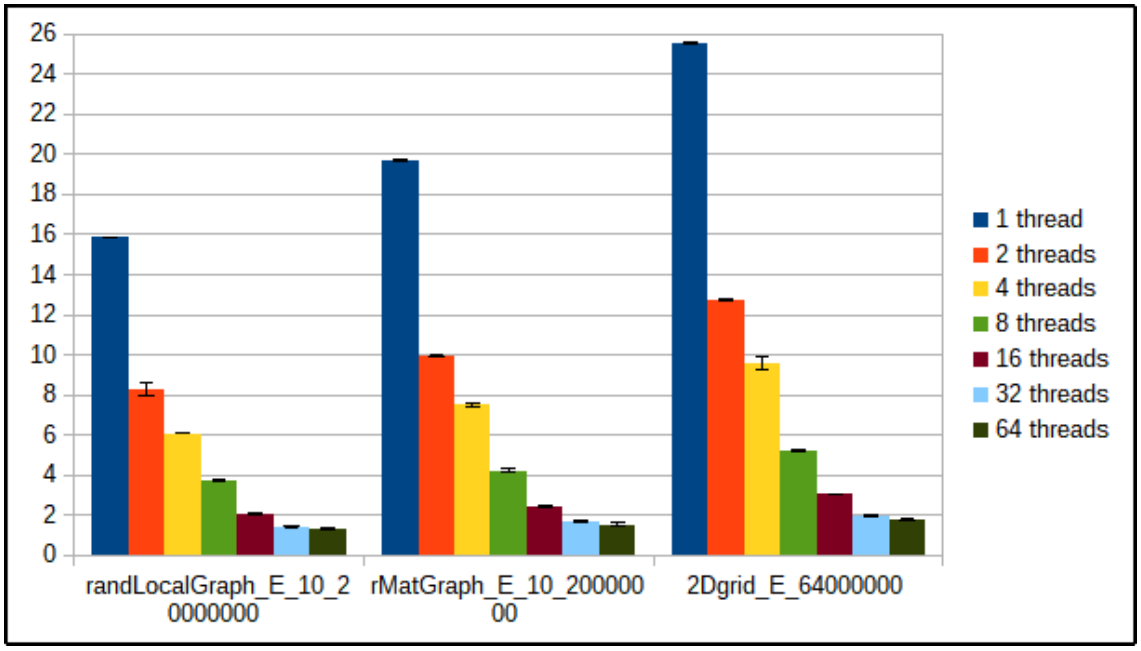


Figure A.229: Execution times (in minutes) of Maximal Matching ran on AMD 32-64 using SBLCWS with WShare (**second version**).

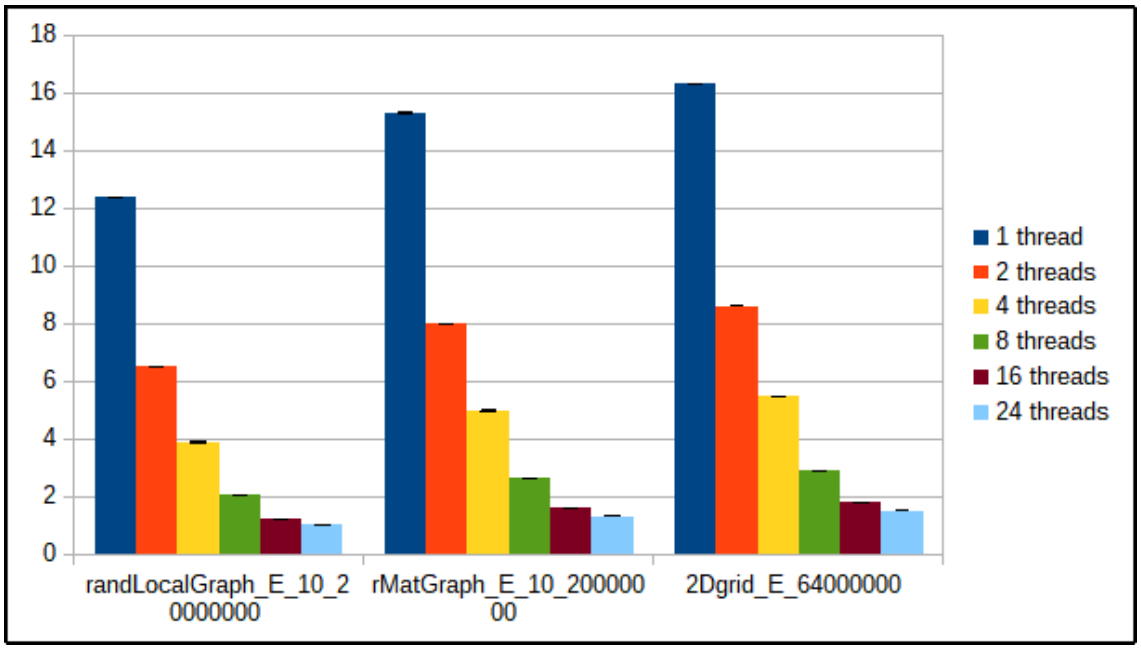


Figure A.230: Execution times (in minutes) of Maximal Matching ran on Intel 16-16 using WSteal

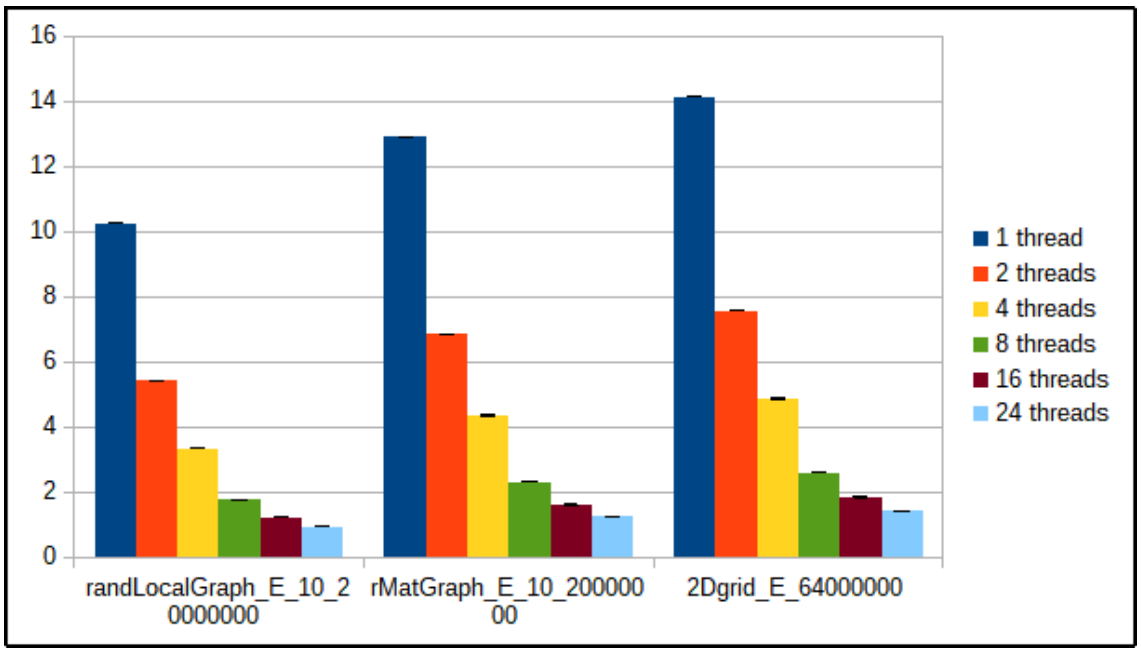


Figure A.231: Execution times (in minutes) of Maximal Matching ran on Intel 16-16 using LCWS

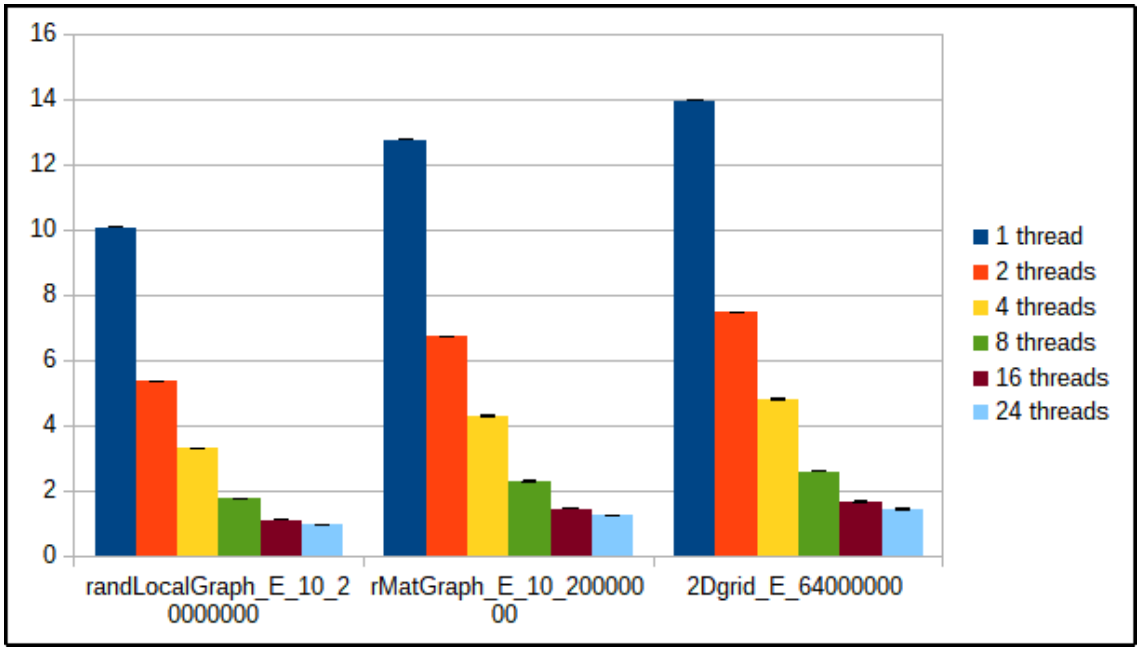


Figure A.232: Execution times (in minutes) of Maximal Matching ran on Intel 16-16 using SBLCWS (**first version**)

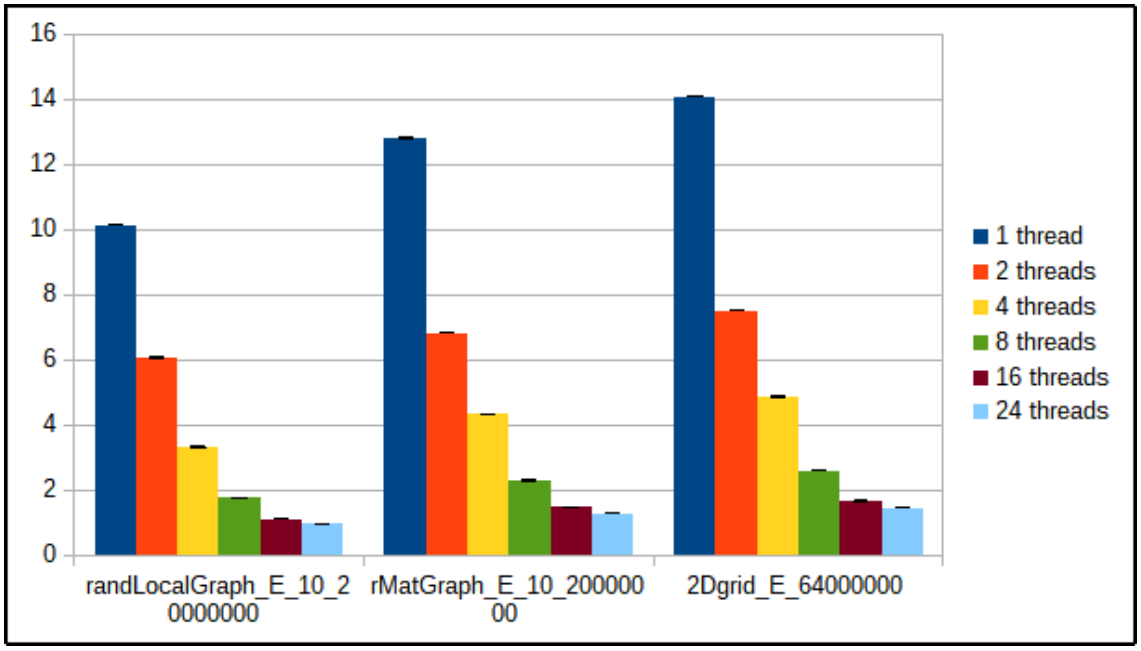


Figure A.233: Execution times (in minutes) of Maximal Matching ran on Intel 16-16 using SBLCWS (**second version**)

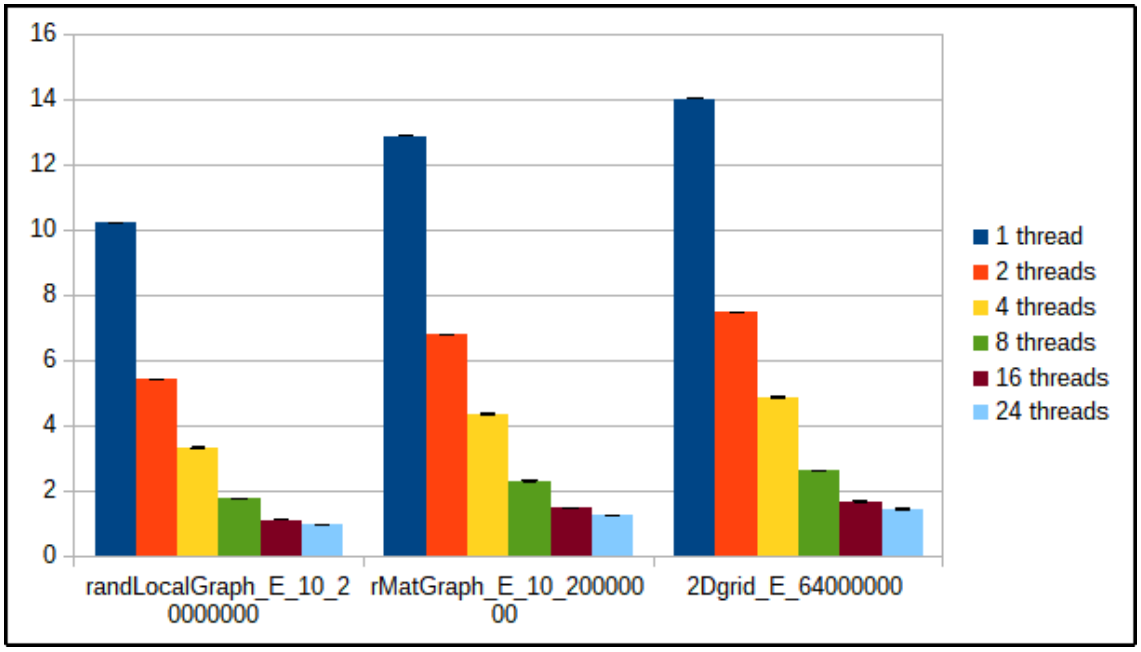


Figure A.234: Execution times (in minutes) of Maximal Matching ran on Intel 16-16 using LCWS with WShare (**tit-for-tat**).

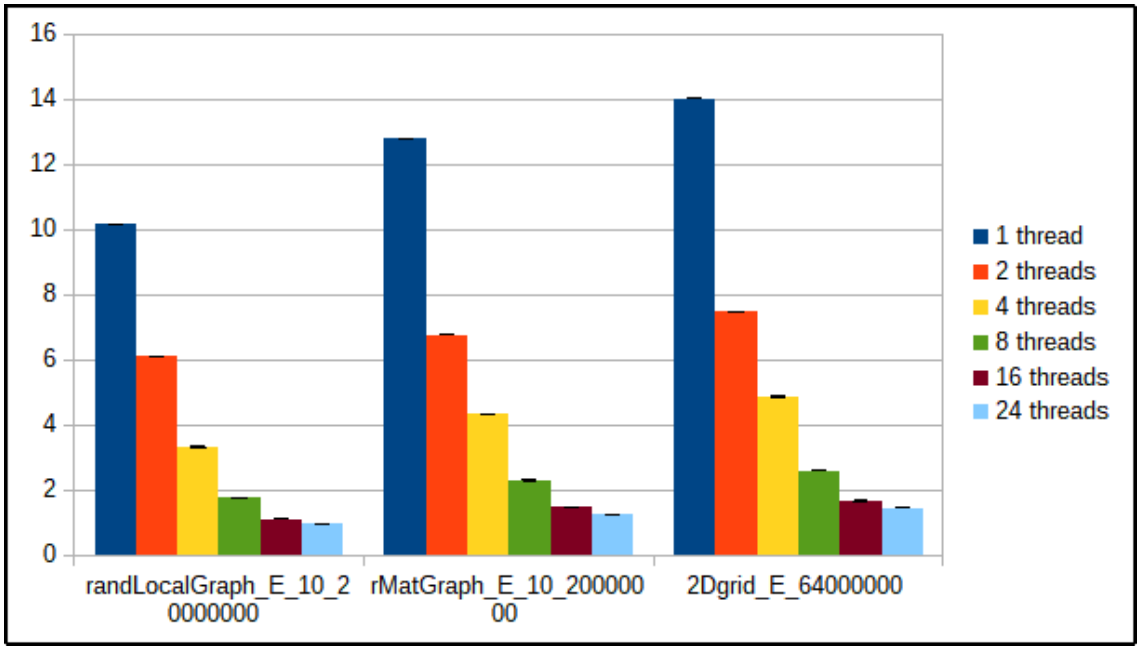


Figure A.235: Execution times (in minutes) of Maximal Matching ran on Intel 16-16 using LCWS with WShare (**continuous tit-for-tat**).

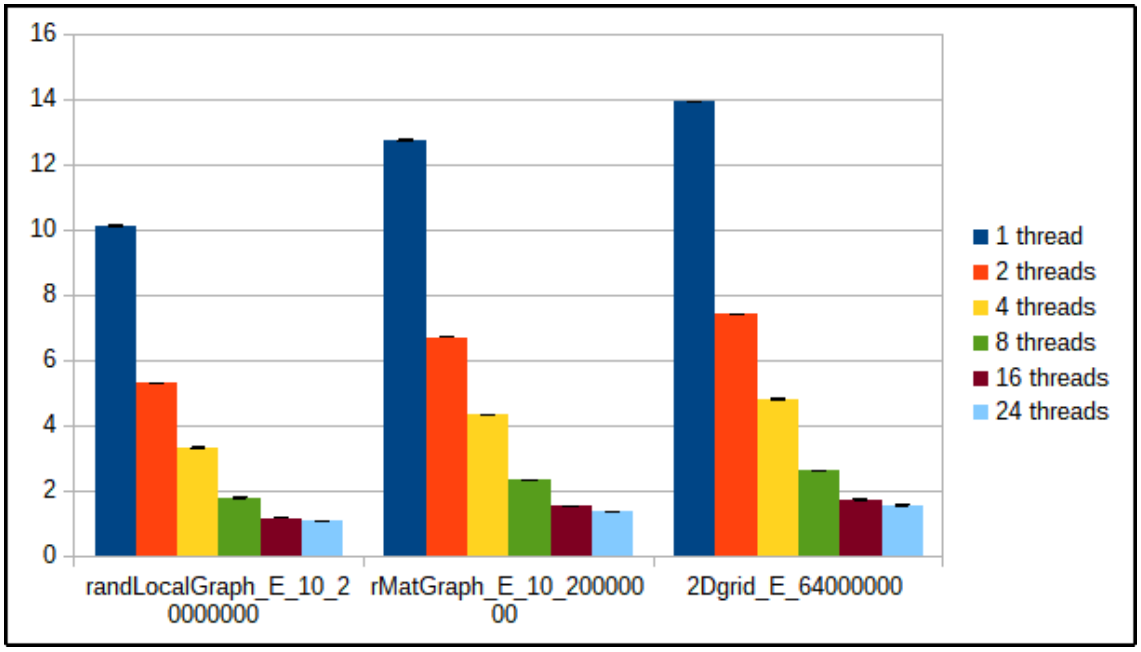


Figure A.236: Execution times (in minutes) of Maximal Matching ran on Intel 16-16 using SBLCWS with WShare (**first version**).

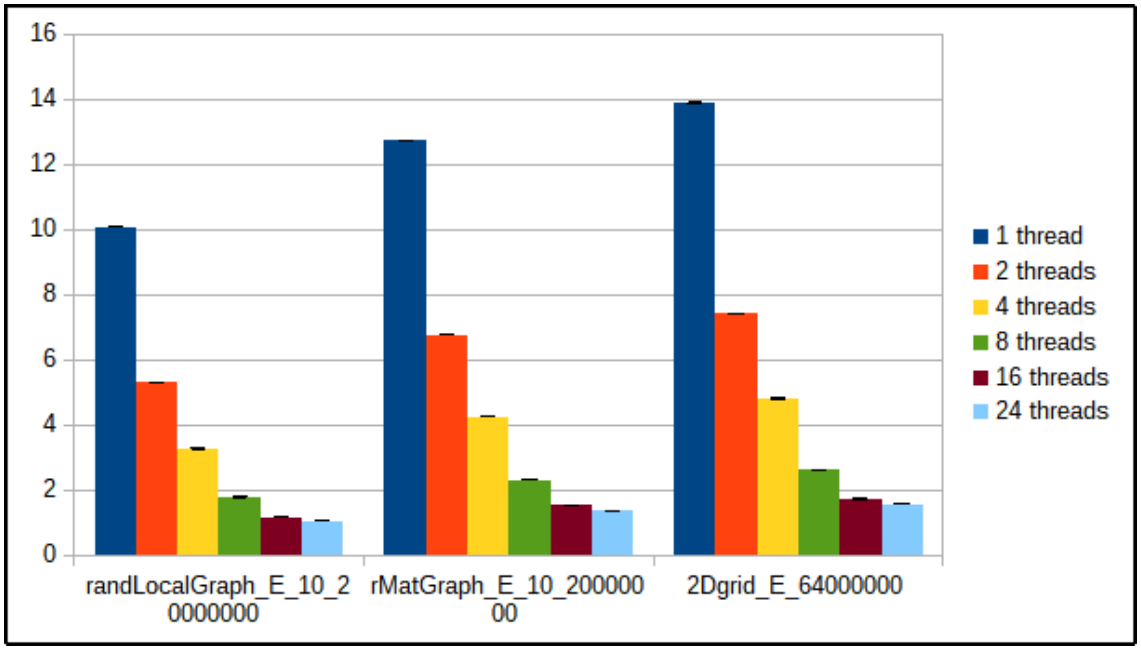


Figure A.237: Execution times (in minutes) of Maximal Matching ran on Intel 16-16 using SBLCWS with WShare (**second version**).

A.11 Maximal Independent Set

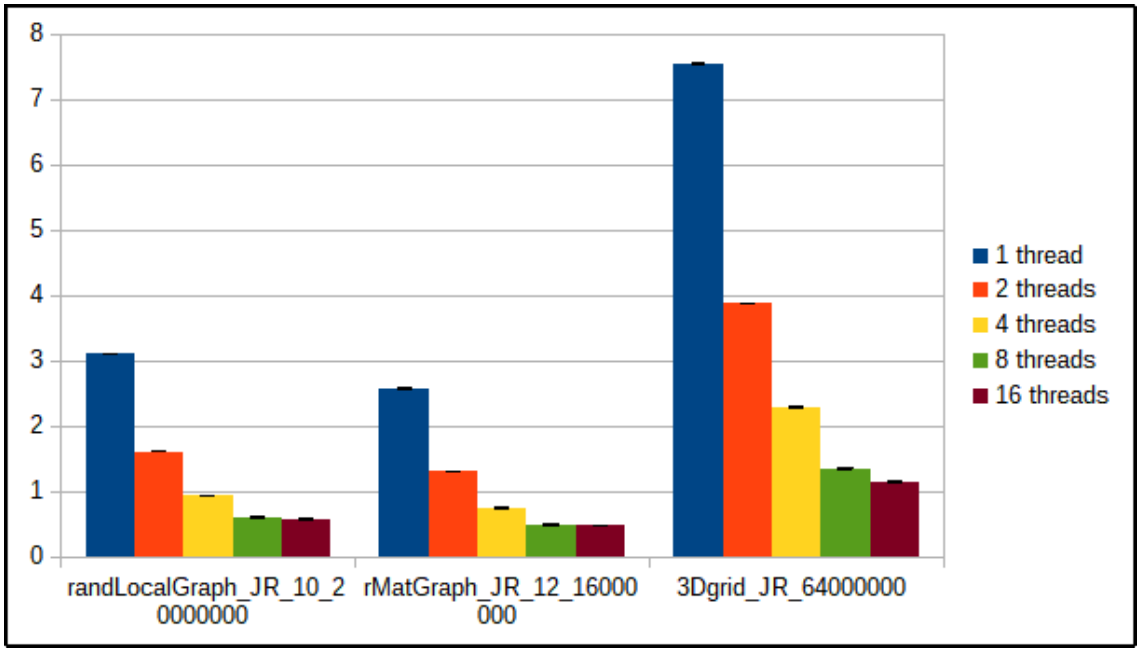


Figure A.238: Execution times (in minutes) of Maximal Independent Set ran on Intel 12-24 using WSteal

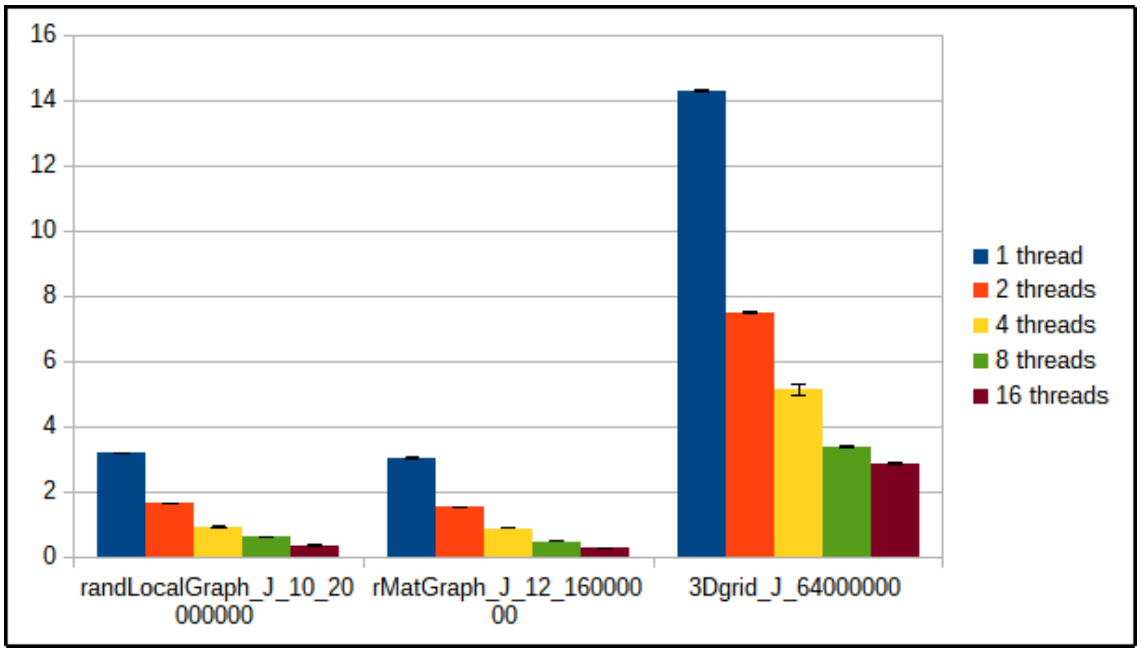


Figure A.239: Execution times (in minutes) of Maximal Independent Set ran on Intel 12-24 using LCWS

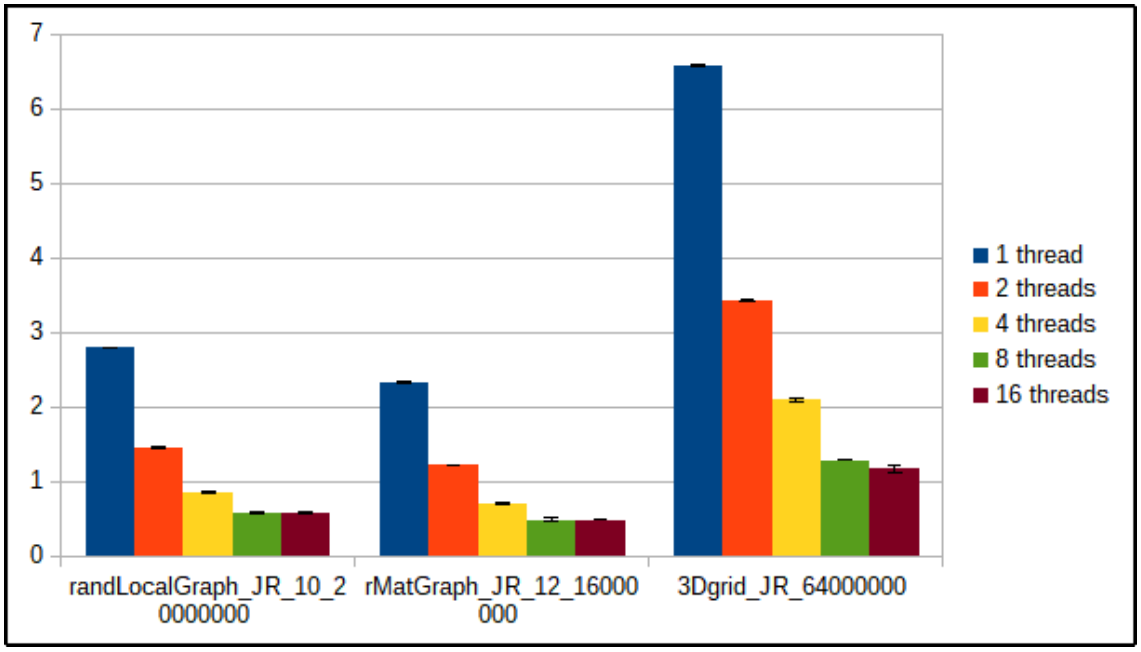


Figure A.240: Execution times (in minutes) of Maximal Independent Set ran on Intel 12-24 using SBLCWS (**first version**)

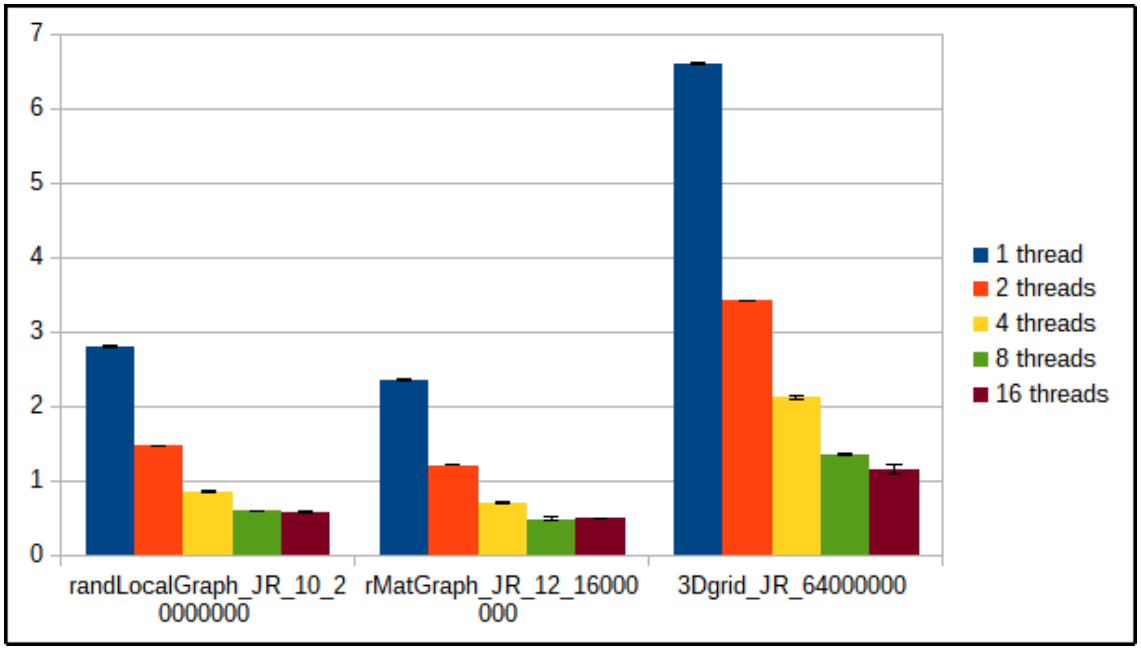


Figure A.241: Execution times (in minutes) of Maximal Independent Set ran on Intel 12-24 using SBLCWS (**second version**)

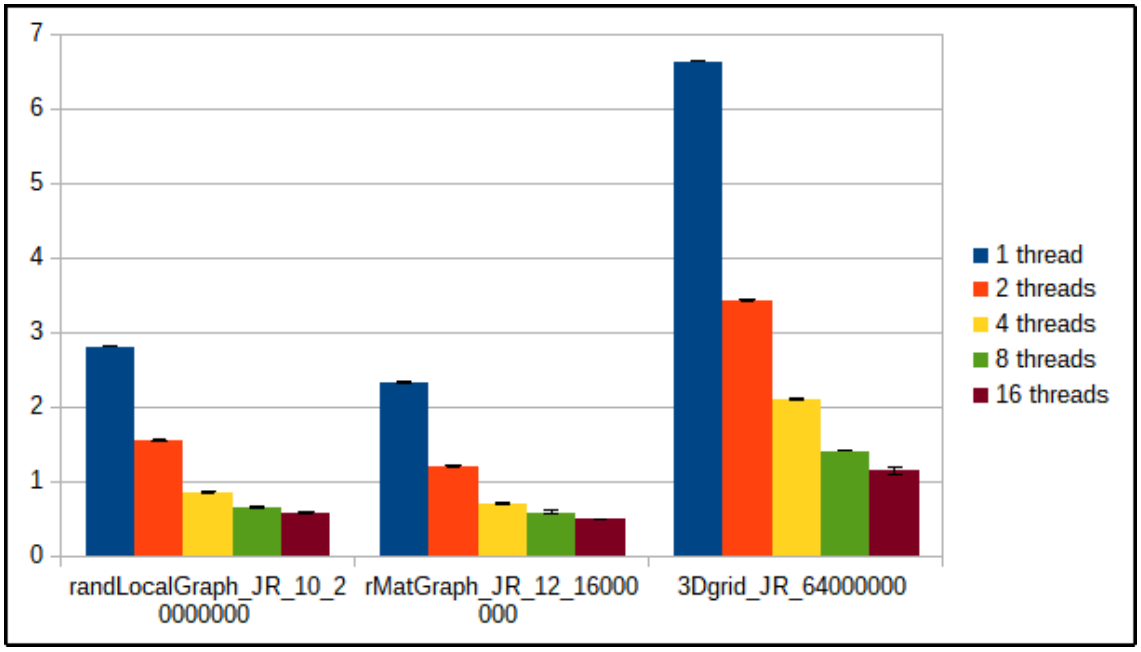


Figure A.242: Execution times (in minutes) of Maximal Independent Set ran on Intel 12-24 using LCWS with WShare (**tit-for-tat**).

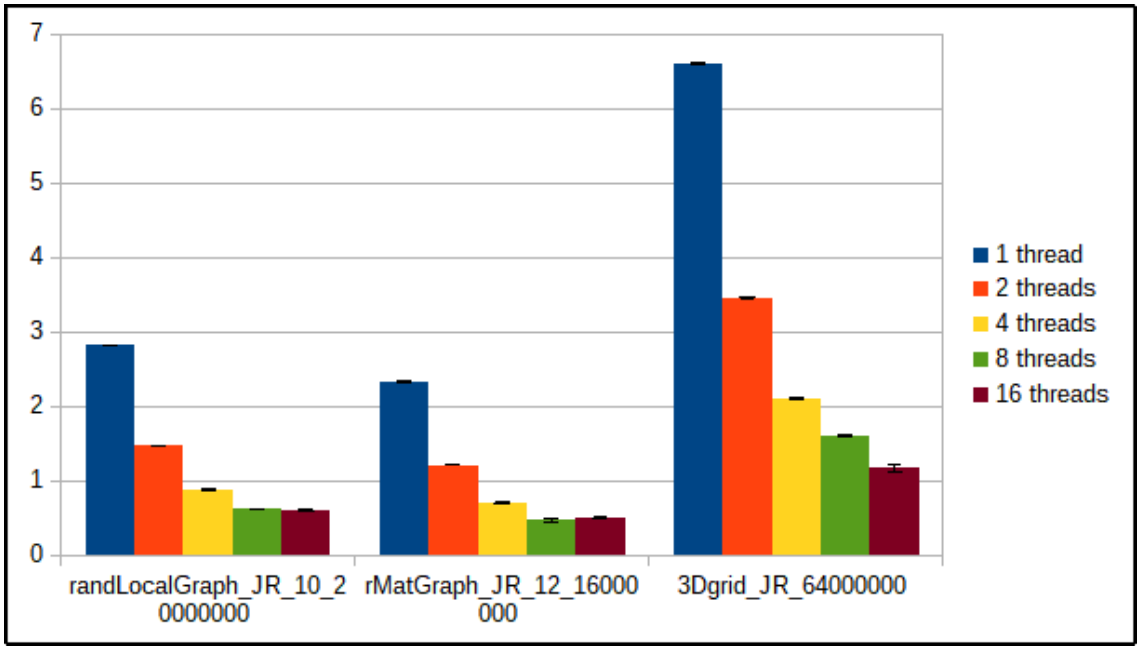


Figure A.243: Execution times (in minutes) of Maximal Independent Set ran on Intel 12-24 using LCWS with WShare (**continuous tit-for-tat**).

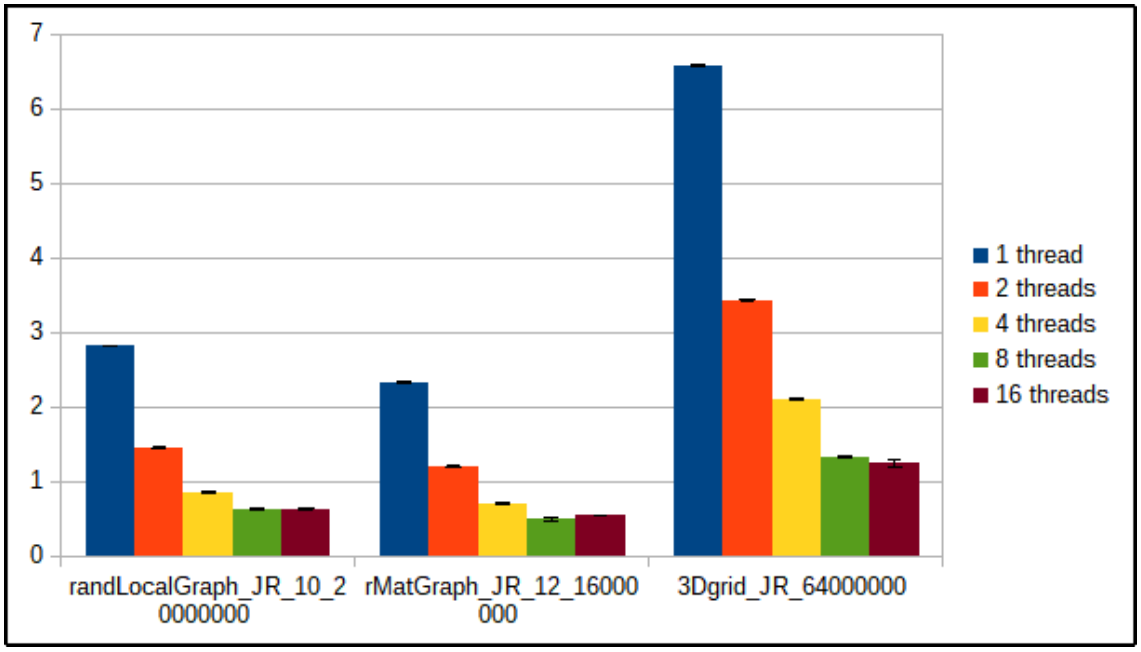


Figure A.244: Execution times (in minutes) of Maximal Independent Set ran on Intel 12-24 using SBLCWS with WShare (**first version**).

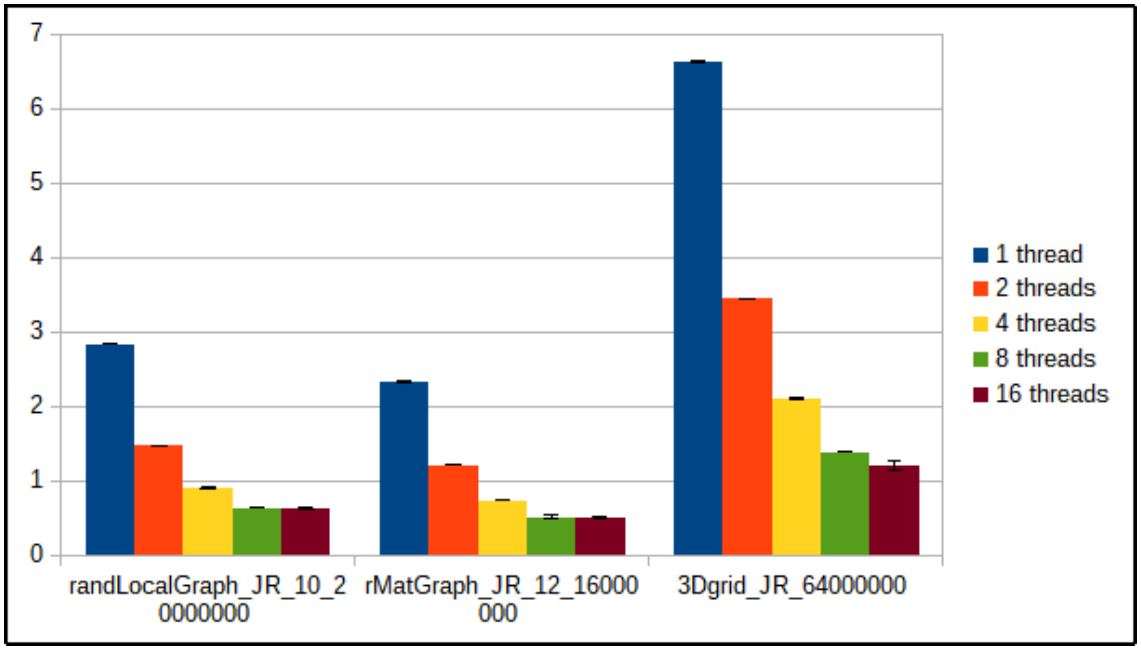


Figure A.245: Execution times (in minutes) of Maximal Independent Set ran on Intel 12-24 using SBLCWS with WShare (**second version**).

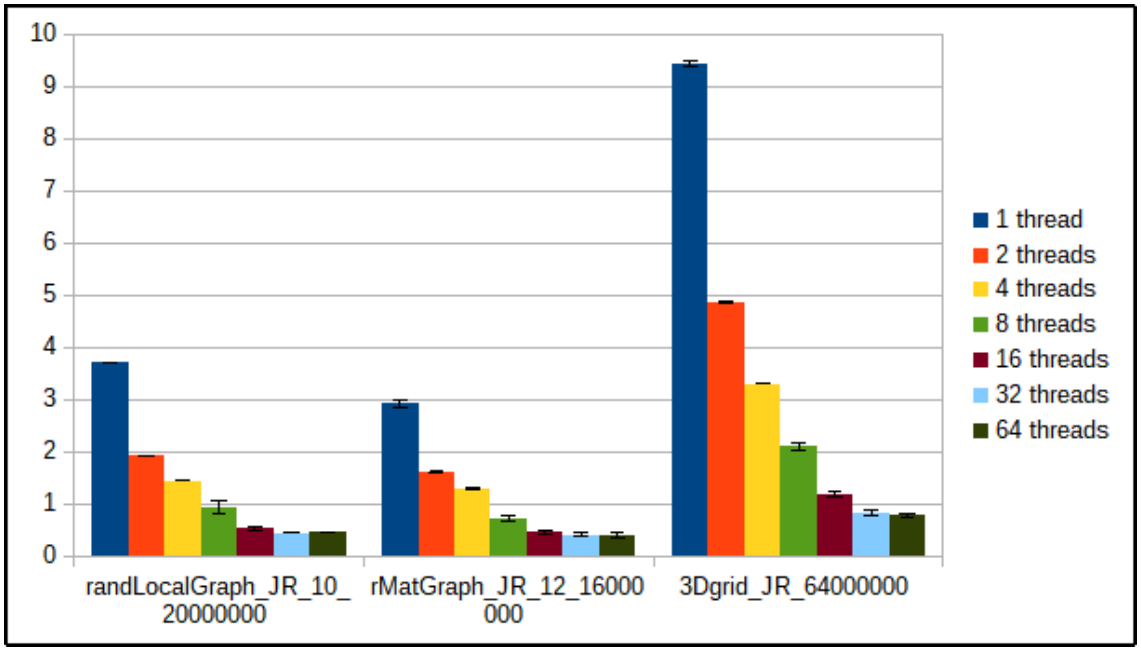


Figure A.246: Execution times (in minutes) of Maximal Independent Set ran on AMD 32-64 using WSteal

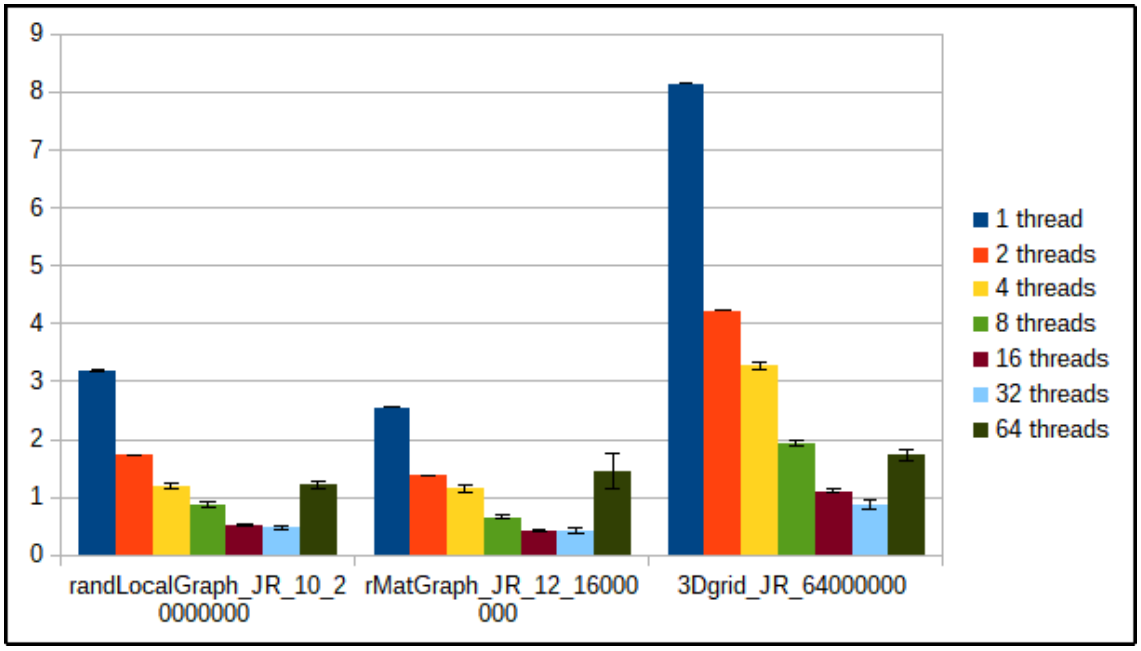


Figure A.247: Execution times (in minutes) of Maximal Independent Set ran on AMD 32-64 using LCWS

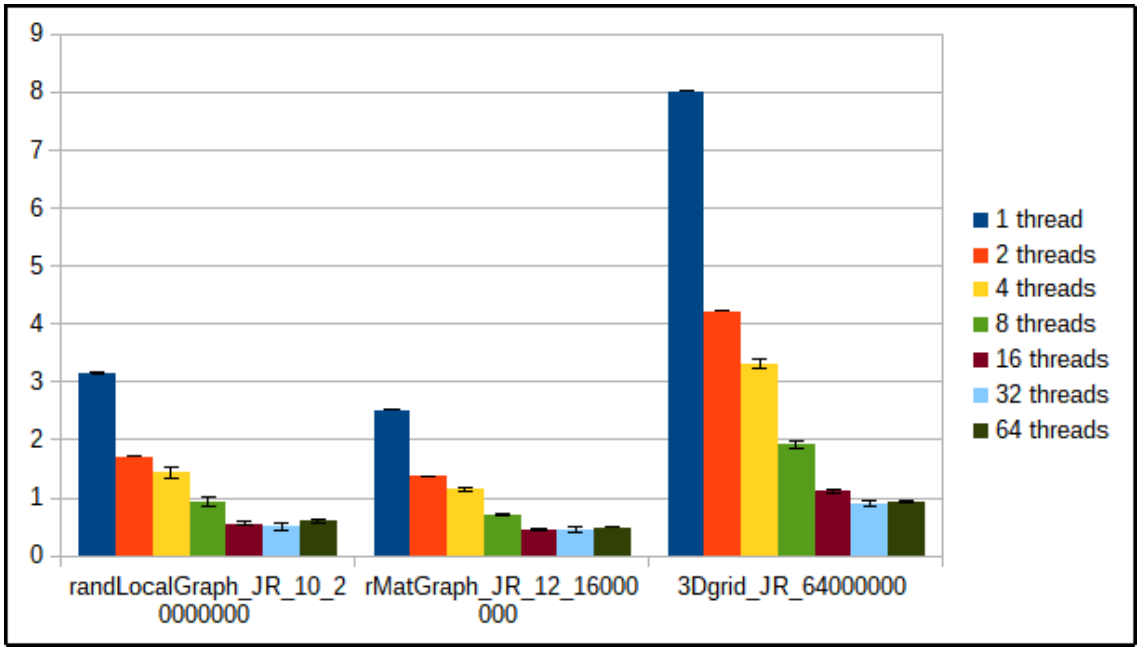


Figure A.248: Execution times (in minutes) of Maximal Independent Set ran on AMD 32-64 using SBLCWS (**first version**)

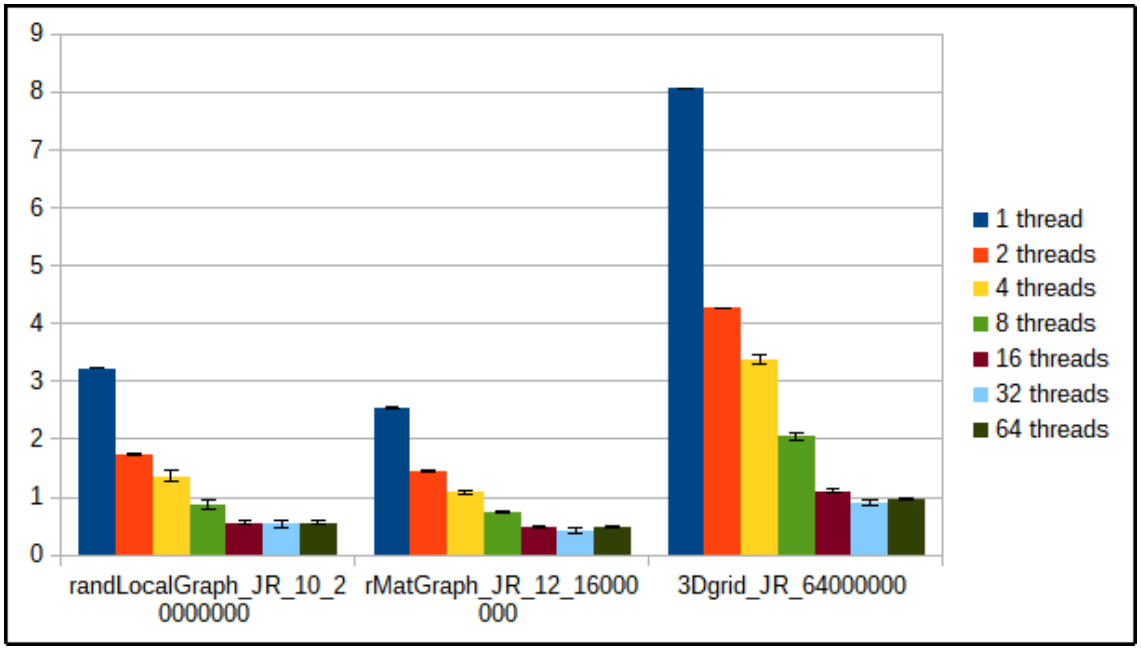


Figure A.249: Execution times (in minutes) of Maximal Independent Set ran on AMD 32-64 using SBLCWS (**second version**)

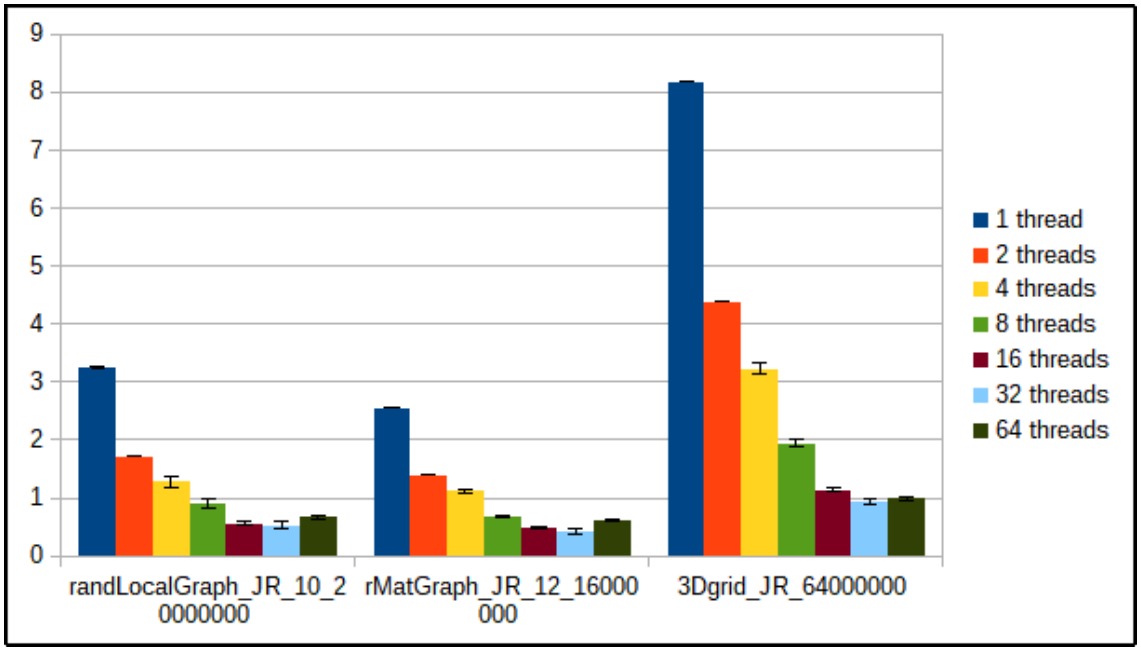


Figure A.250: Execution times (in minutes) of Maximal Independent Set ran on AMD 32-64 using LCWS with WShare (**tit-for-tat**).

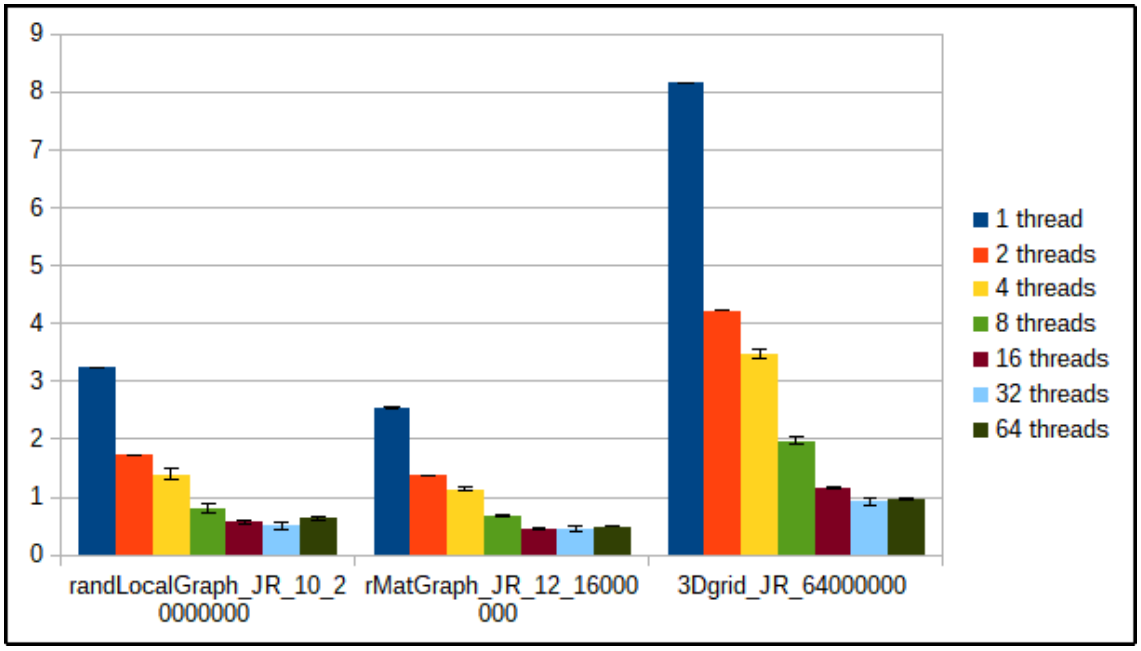


Figure A.251: Execution times (in minutes) of Maximal Independent Set ran on AMD 32-64 using LCWS with WShare (**continuous tit-for-tat**).

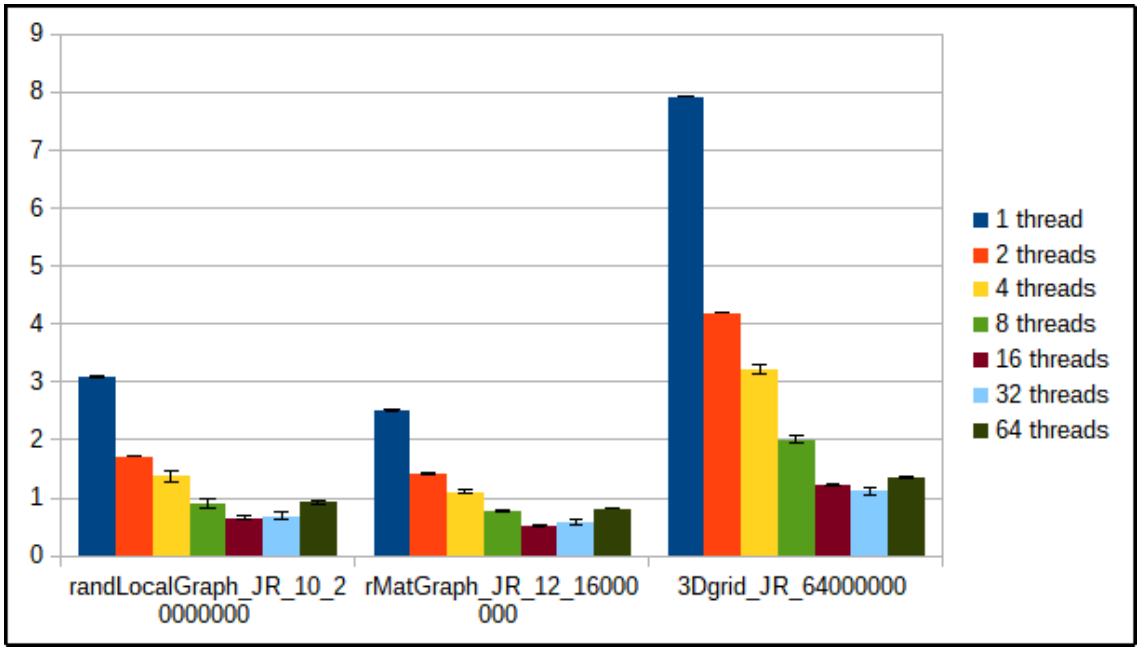


Figure A.252: Execution times (in minutes) of Maximal Independent Set ran on AMD 32-64 using SBLCWS with WShare (**first version**).

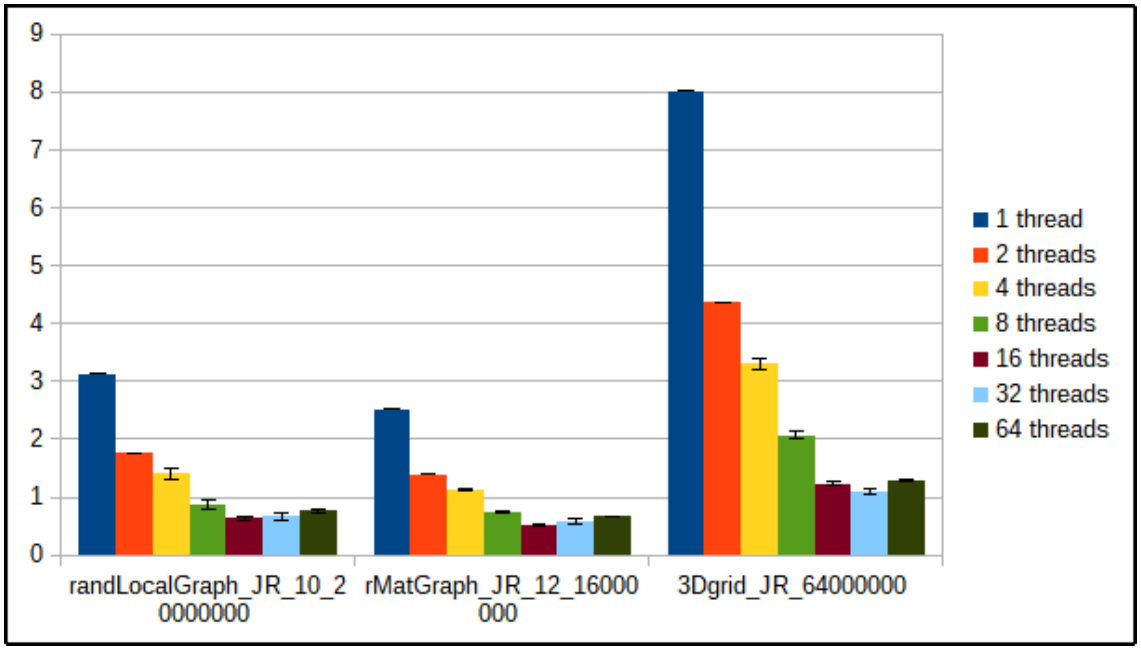


Figure A.253: Execution times (in minutes) of Maximal Independent Set ran on AMD 32-64 using SBLCWS with WShare (**second version**).

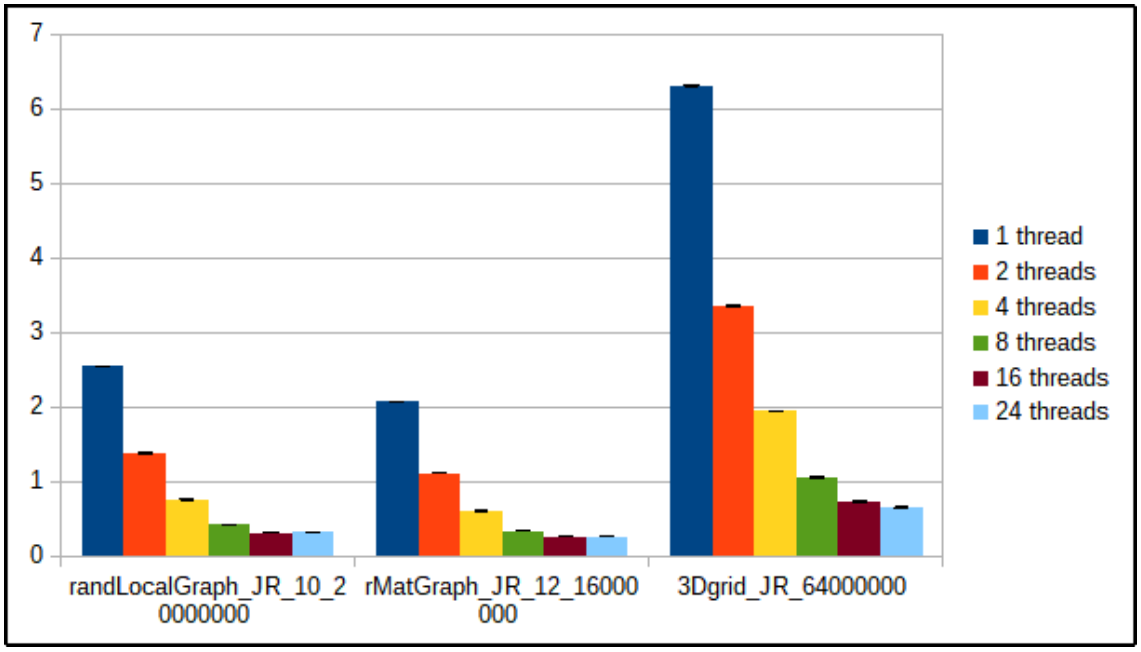


Figure A.254: Execution times (in minutes) of Maximal Independent Set ran on Intel 16-16 using WSteal

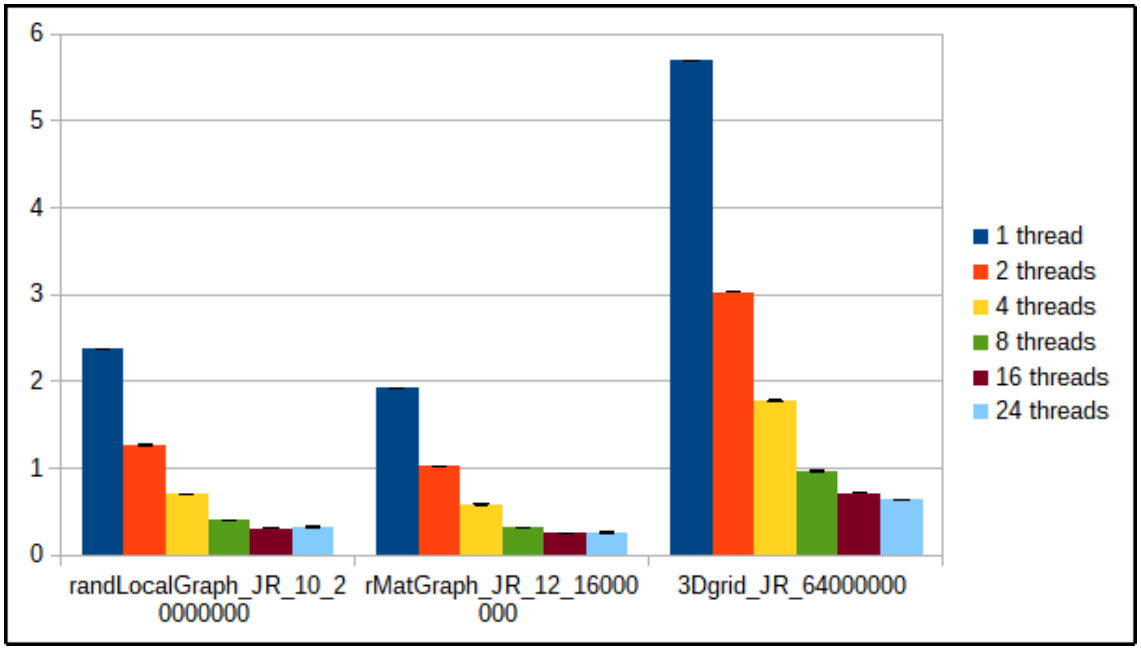


Figure A.255: Execution times (in minutes) of Maximal Independent Set ran on Intel 16-16 using LCWS

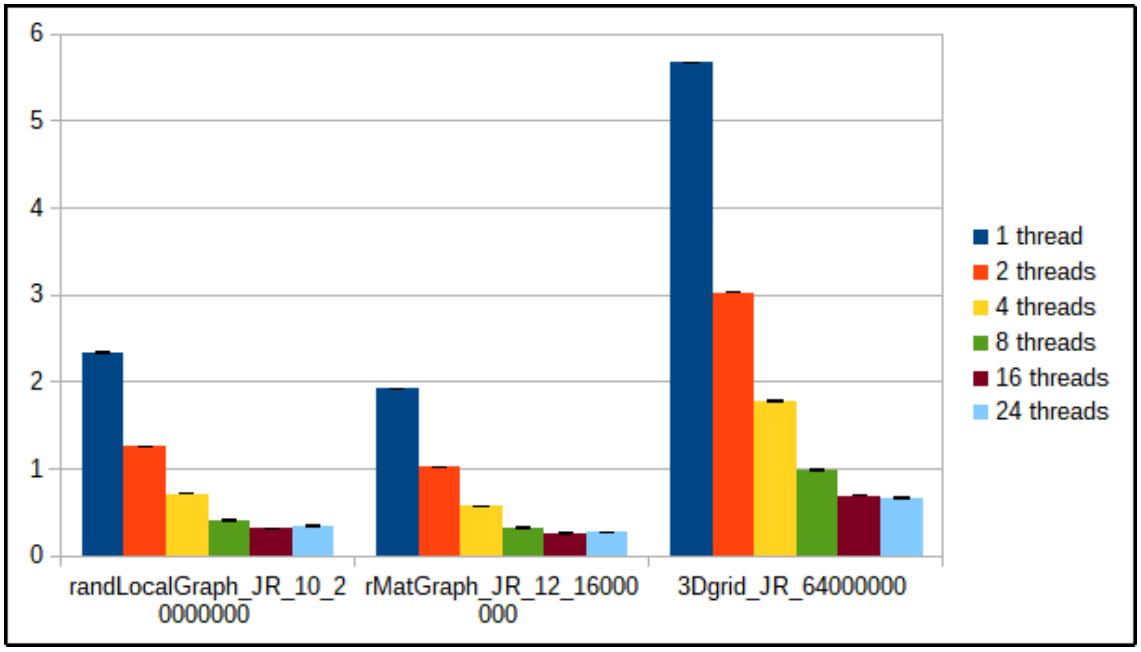


Figure A.256: Execution times (in minutes) of Maximal Independent Set ran on Intel 16-16 using SBLCWS (**first version**)

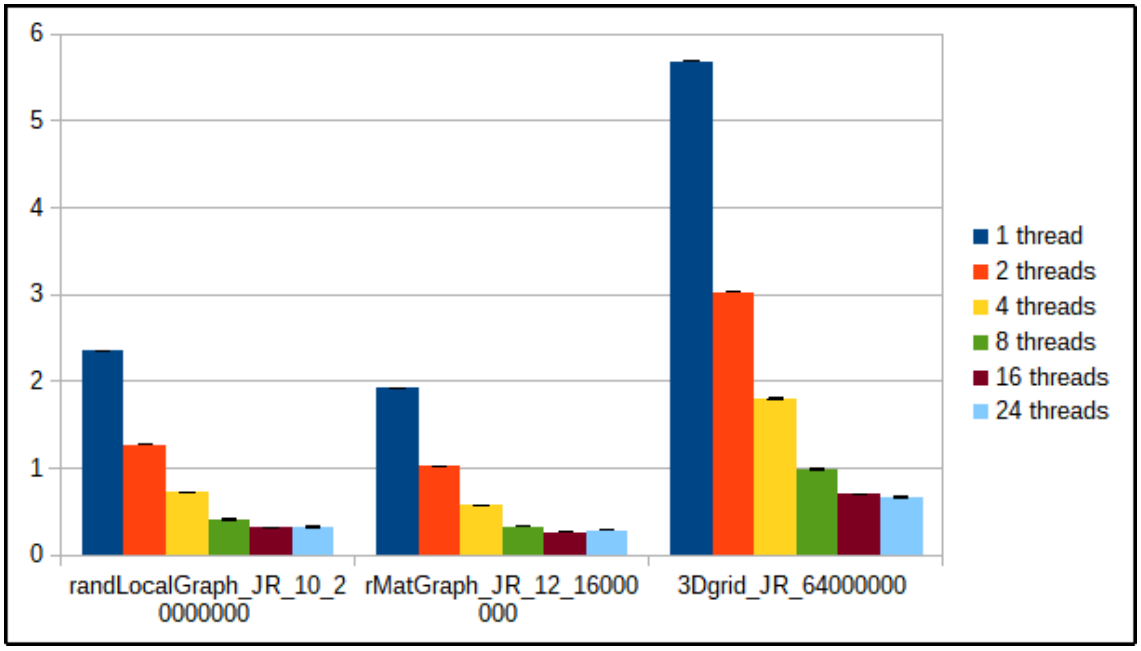


Figure A.257: Execution times (in minutes) of Maximal Independent Set ran on Intel 16-16 using SBLCWS (**second version**)

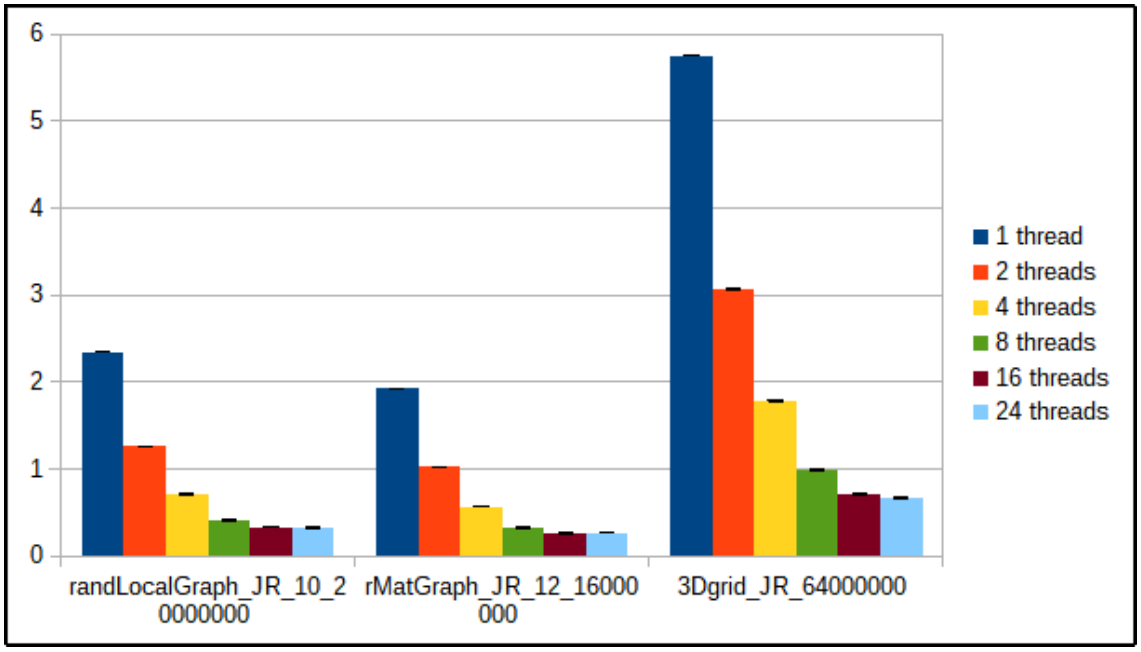


Figure A.258: Execution times (in minutes) of Maximal Independent Set ran on Intel 16-16 using LCWS with WShare (**tit-for-tat**).

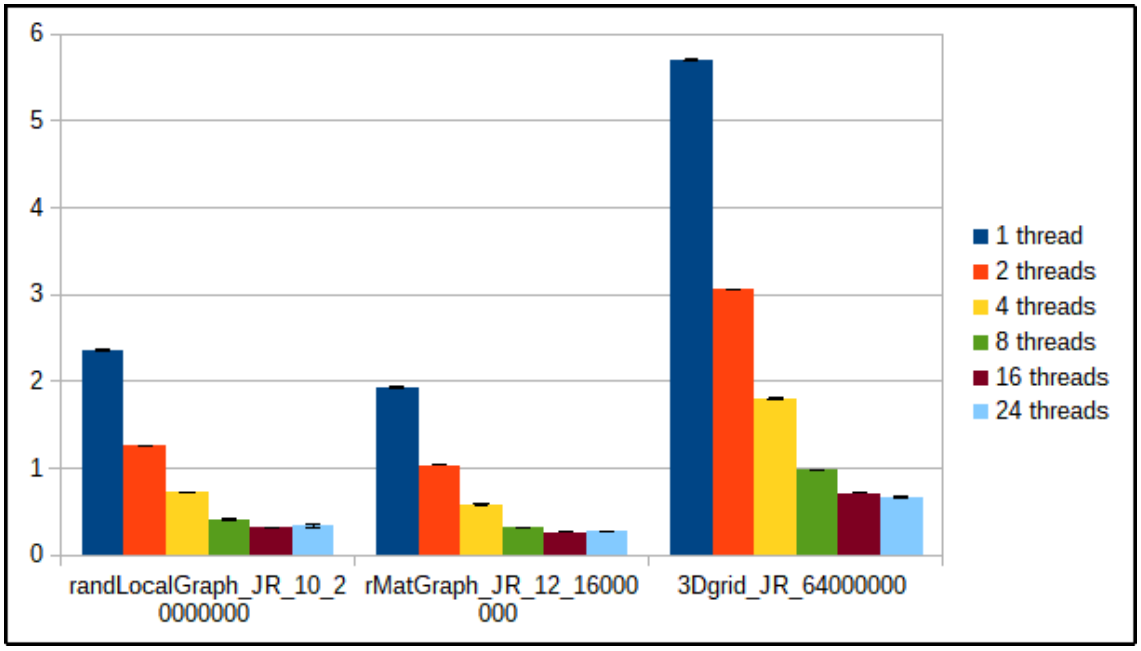


Figure A.259: Execution times (in minutes) of Maximal Independent Set ran on Intel 16-16 using LCWS with WShare (**continuous tit-for-tat**).

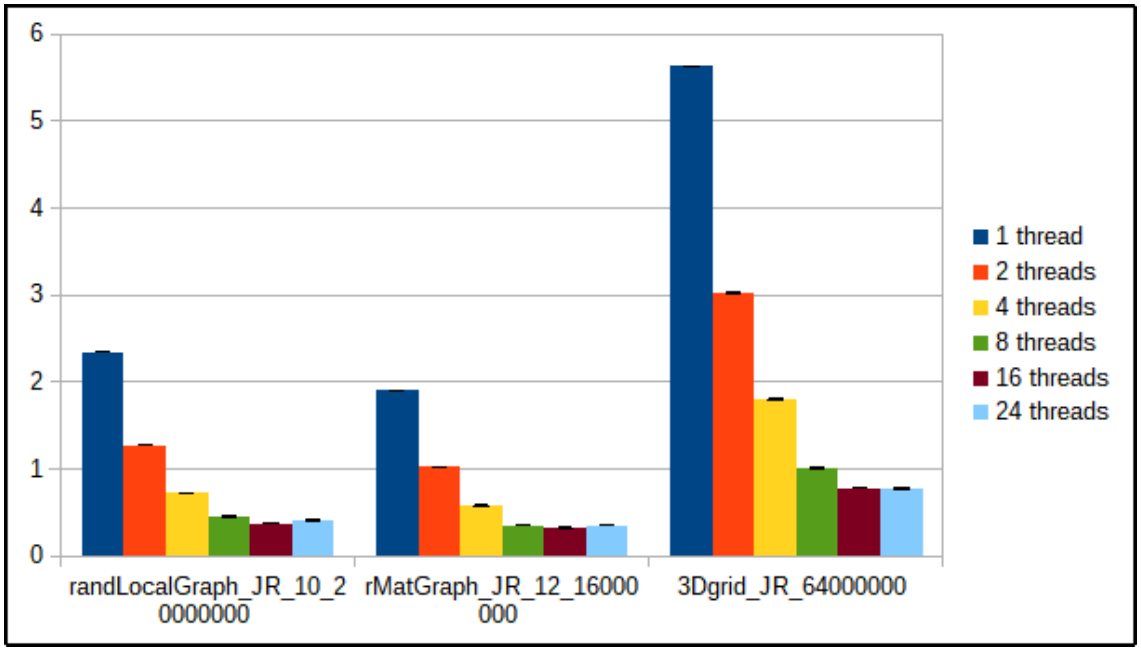


Figure A.260: Execution times (in minutes) of Maximal Independent Set ran on Intel 16-16 using SBLCWS with WShare (**first version**).

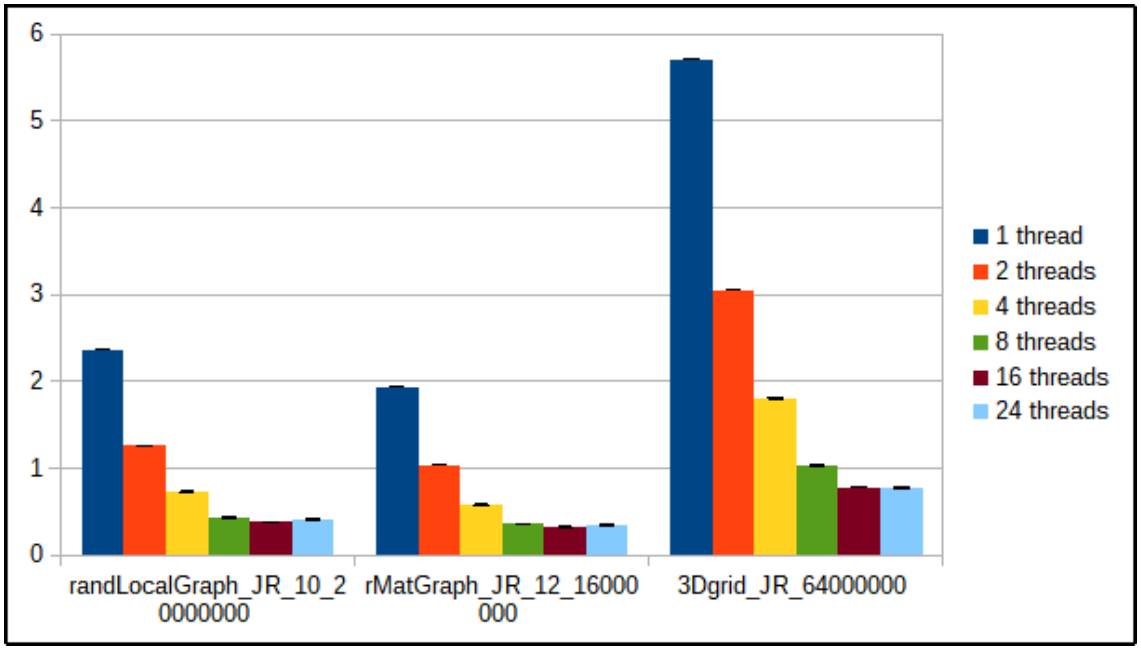


Figure A.261: Execution times (in minutes) of Maximal Independent Set ran on Intel 16-16 using SBLCWS with WShare (**second version**).

A.12 Nearest Neighbors

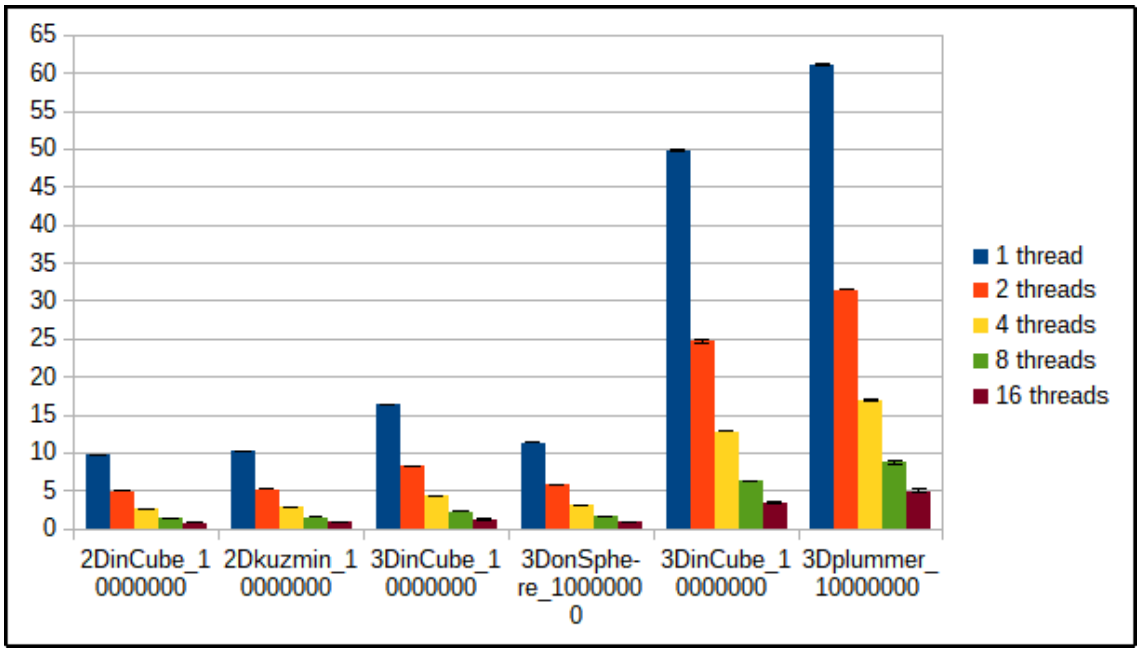


Figure A.262: Execution times (in minutes) of Nearest Neighbors ran on Intel 12-24 using WSteal

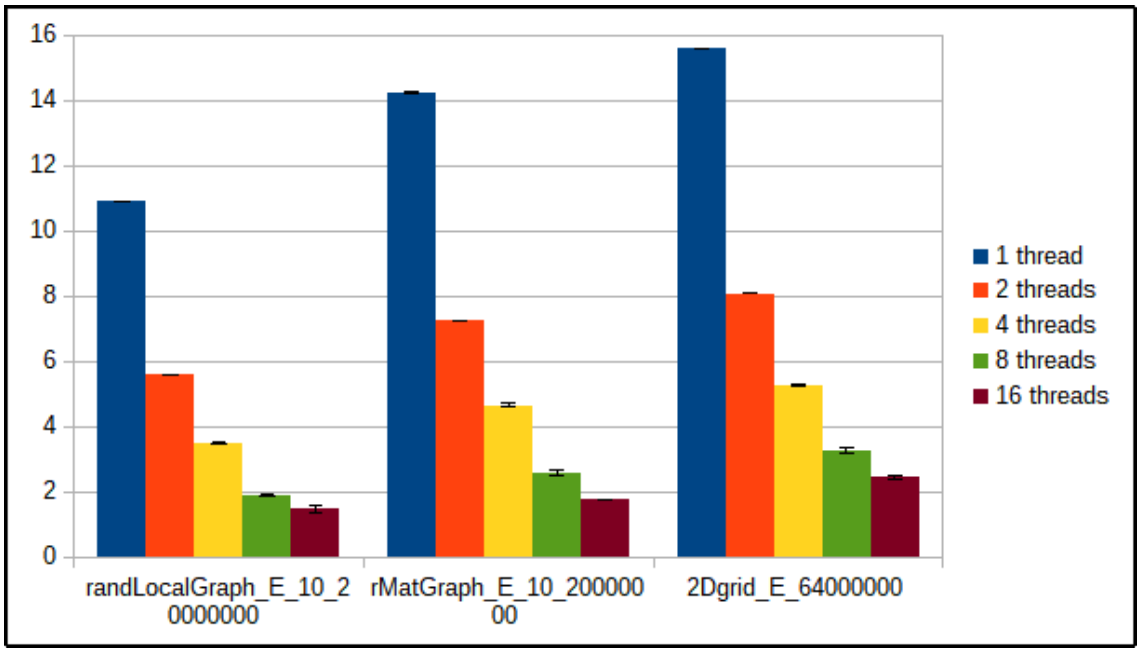


Figure A.263: Execution times (in minutes) of Nearest Neighbors ran on Intel 12-24 using LCWS

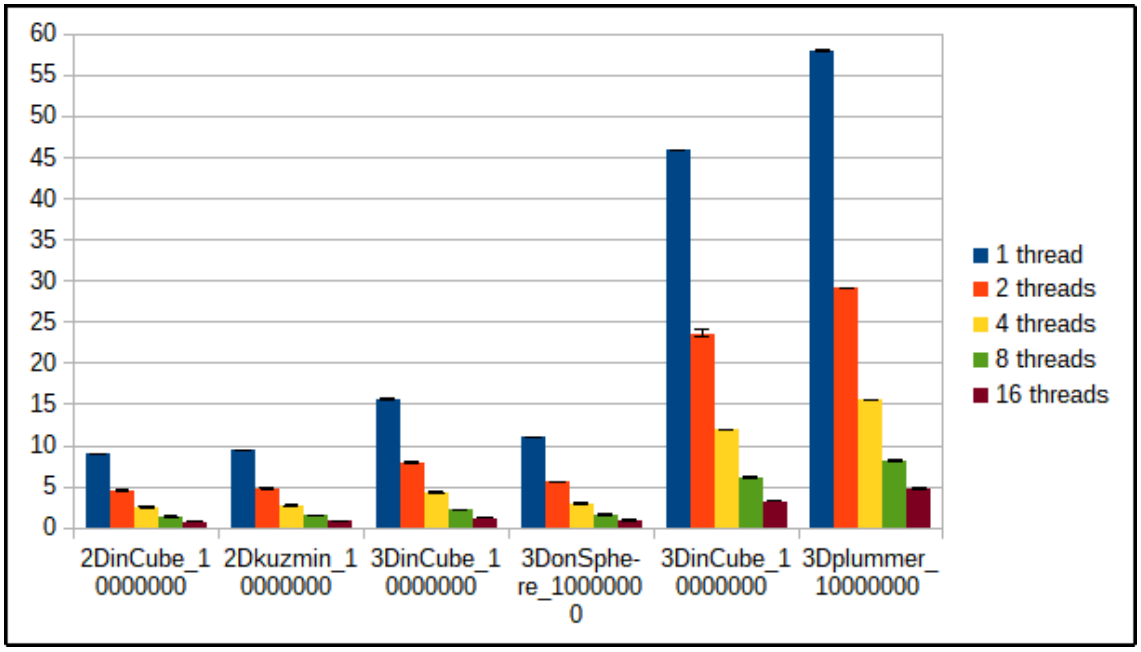


Figure A.264: Execution times (in minutes) of Nearest Neighbors ran on Intel 12-24 using SBLCWS (**first version**)

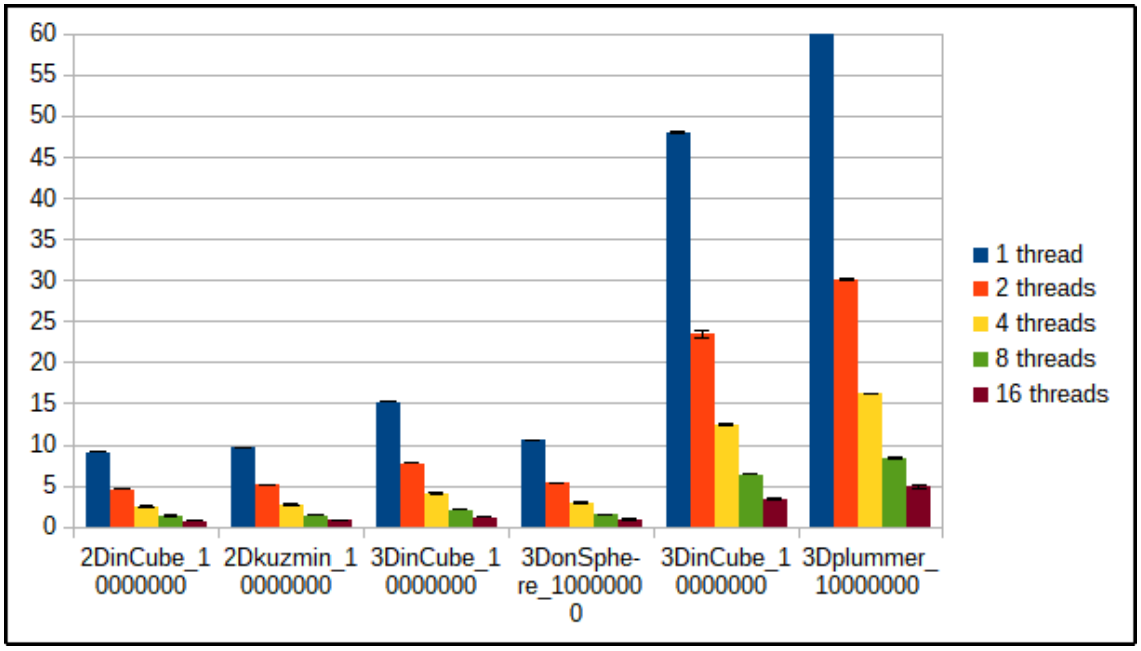


Figure A.265: Execution times (in minutes) of Nearest Neighbors ran on Intel 12-24 using SBLCWS (**second version**)

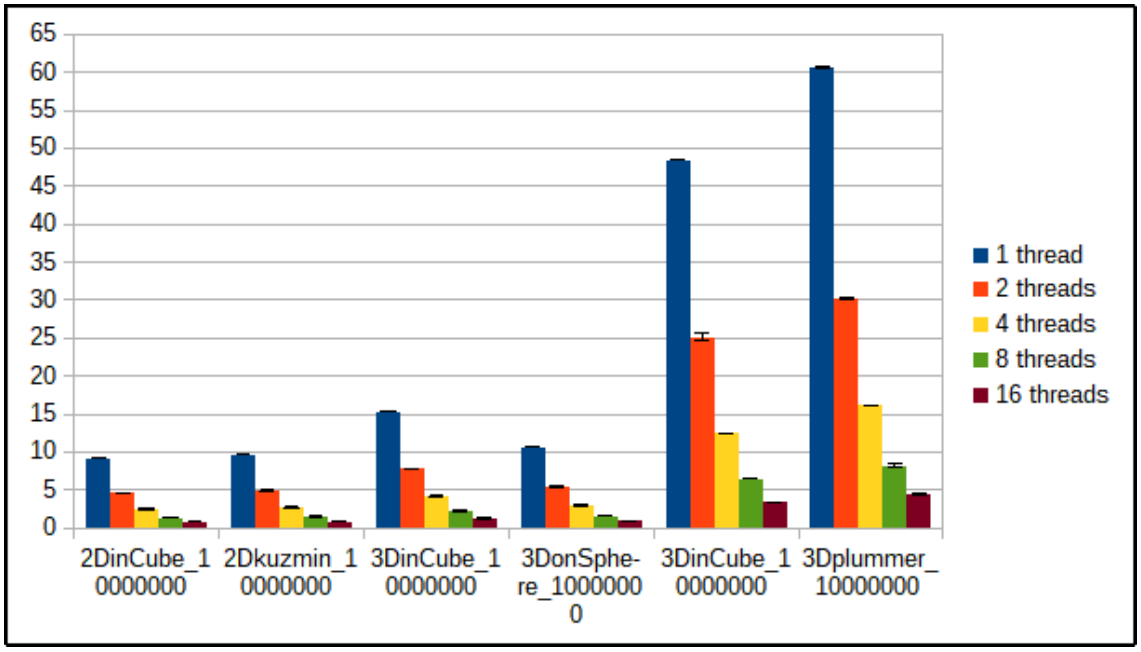


Figure A.266: Execution times (in minutes) of Nearest Neighbors ran on Intel 12-24 using LCWS with WShare (**tit-for-tat**).

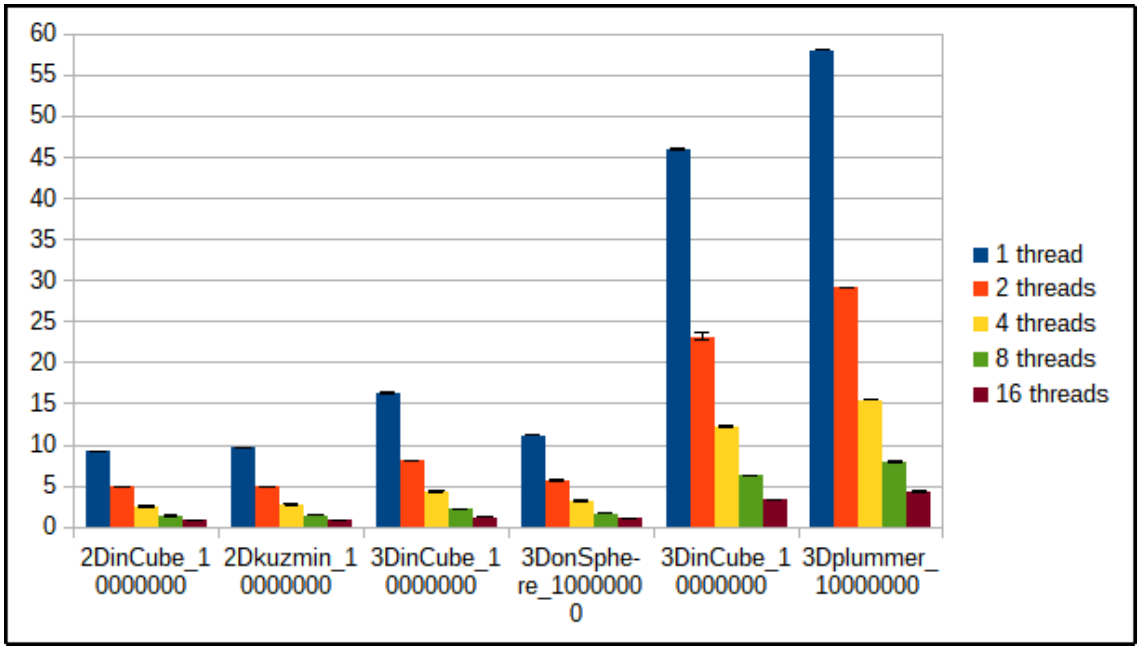


Figure A.267: Execution times (in minutes) of Nearest Neighbors ran on Intel 12-24 using LCWS with WShare (**continuous tit-for-tat**).

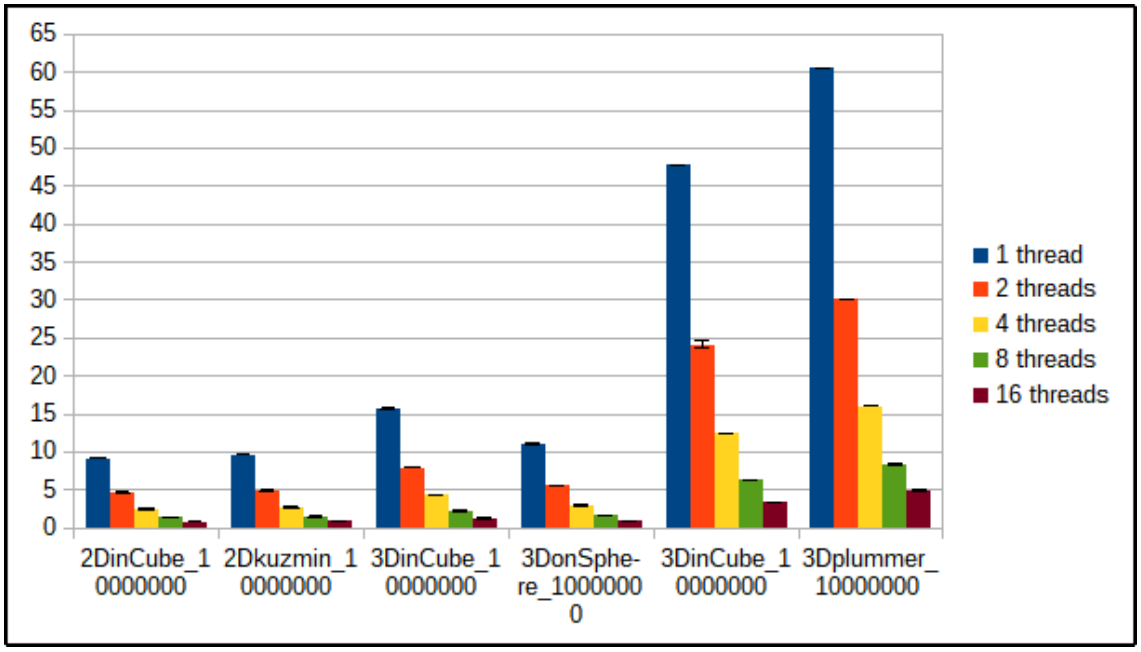


Figure A.268: Execution times (in minutes) of Nearest Neighbors ran on Intel 12-24 using SBLCWS with WShare (**first version**).

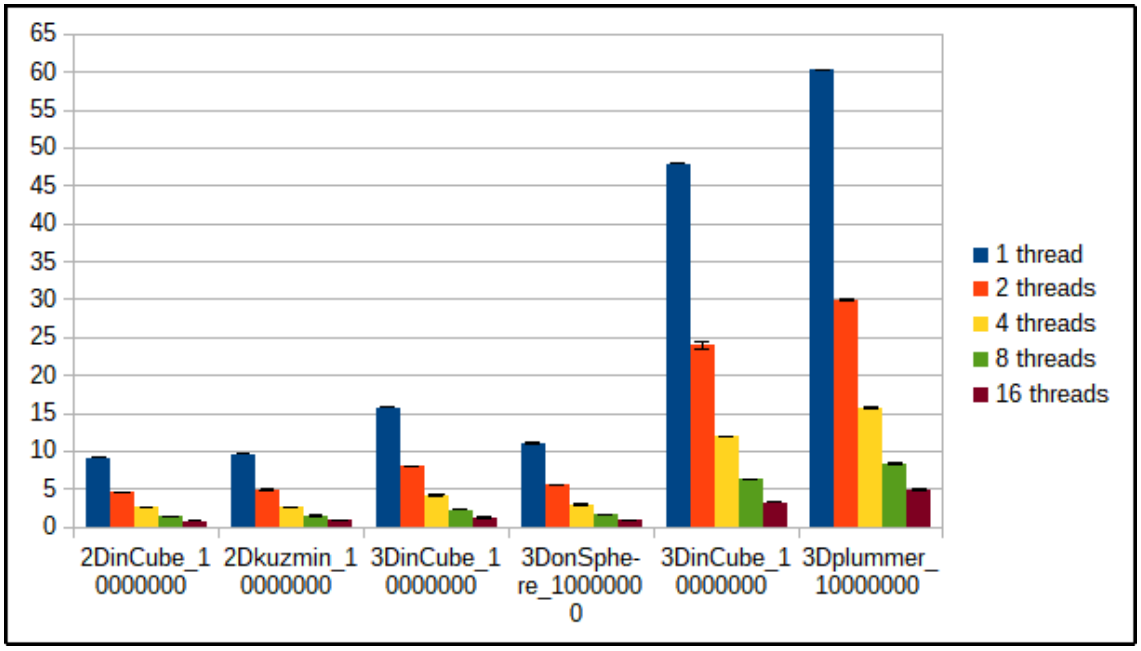


Figure A.269: Execution times (in minutes) of Nearest Neighbors ran on Intel 12-24 using SBLCWS with WShare (**second version**).

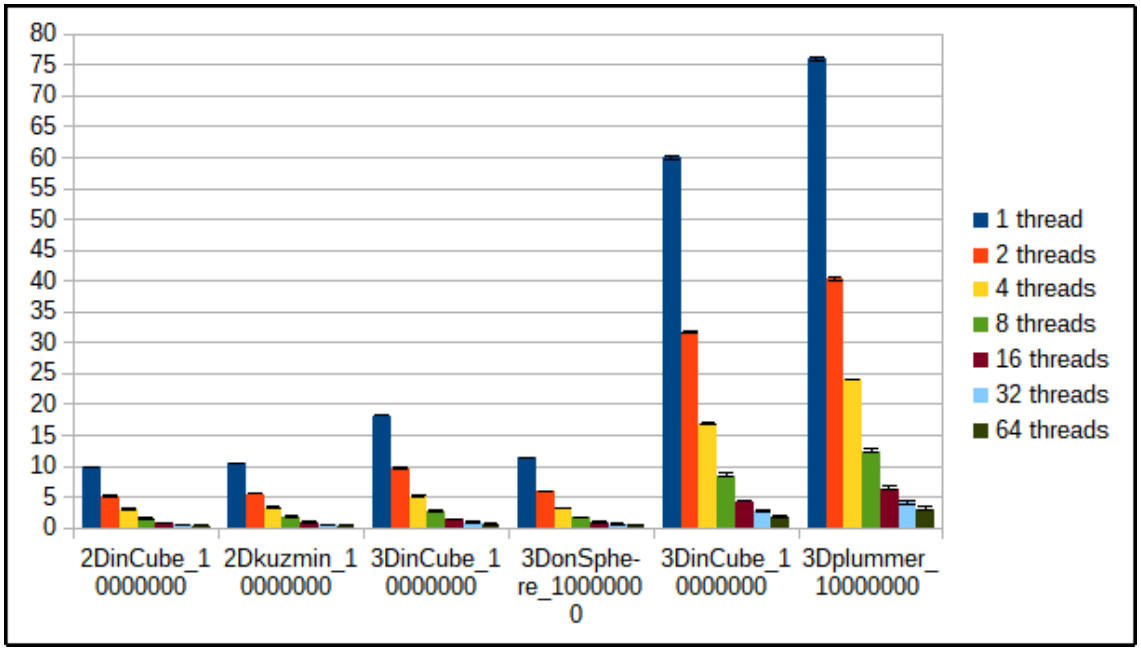


Figure A.270: Execution times (in minutes) of Nearest Neighbors ran on AMD 32-64 using WSteal

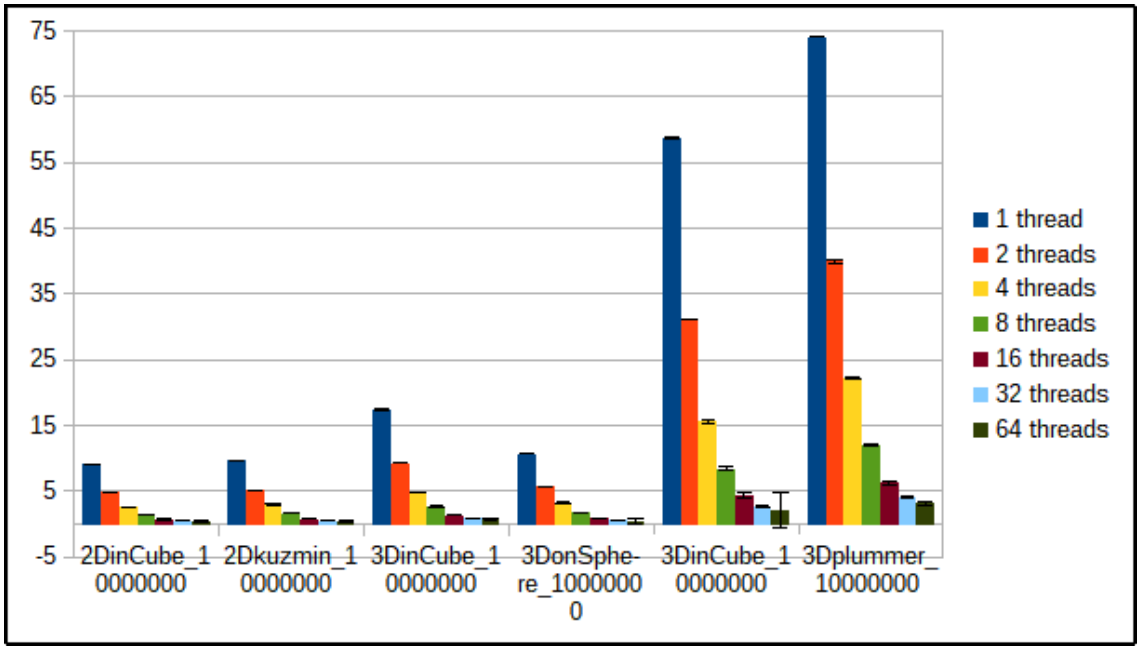


Figure A.271: Execution times (in minutes) of Nearest Neighbors ran on AMD 32-64 using LCWS

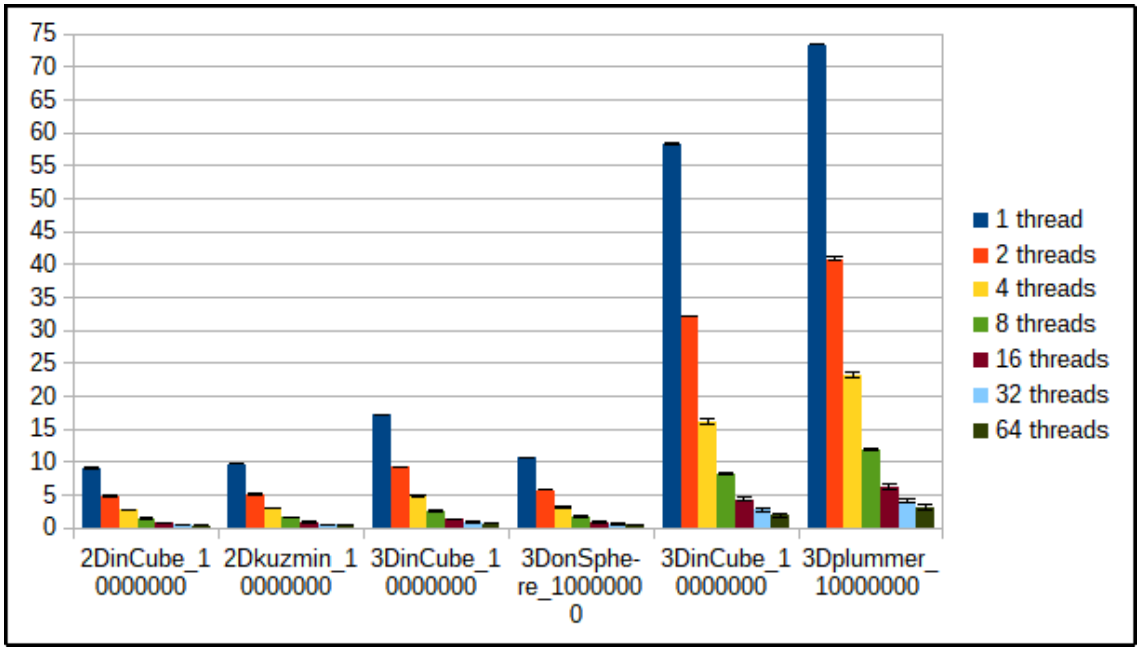


Figure A.272: Execution times (in minutes) of Nearest Neighbors ran on AMD 32-64 using SBLCWS (**first version**)

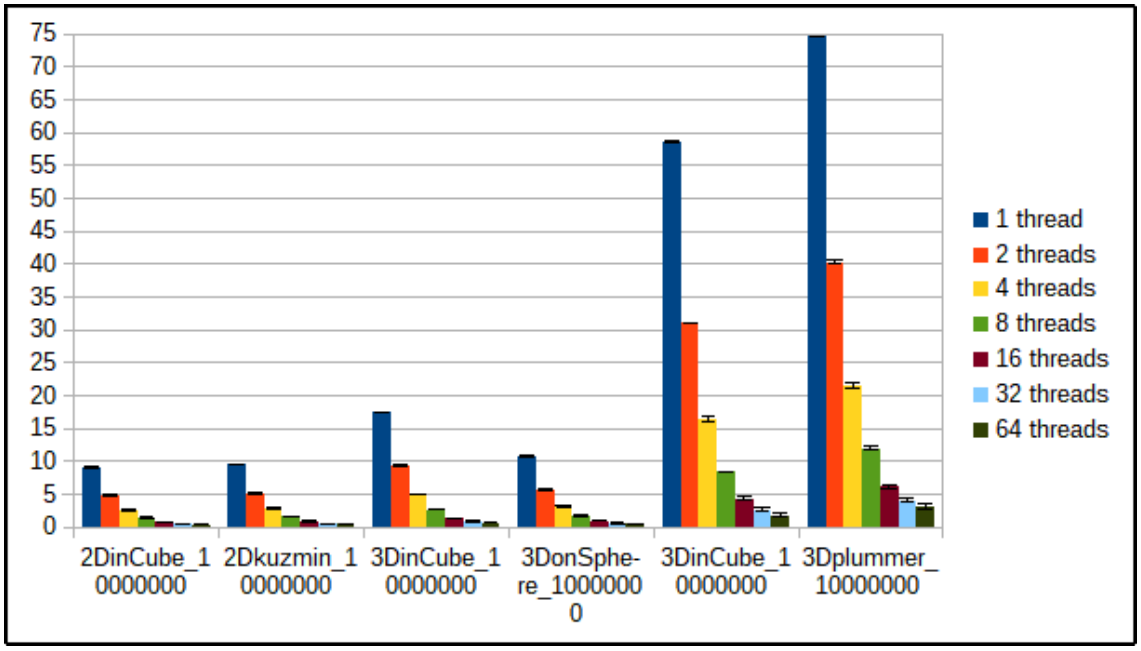


Figure A.273: Execution times (in minutes) of Nearest Neighbors ran on AMD 32-64 using SBLCWS (**second version**)

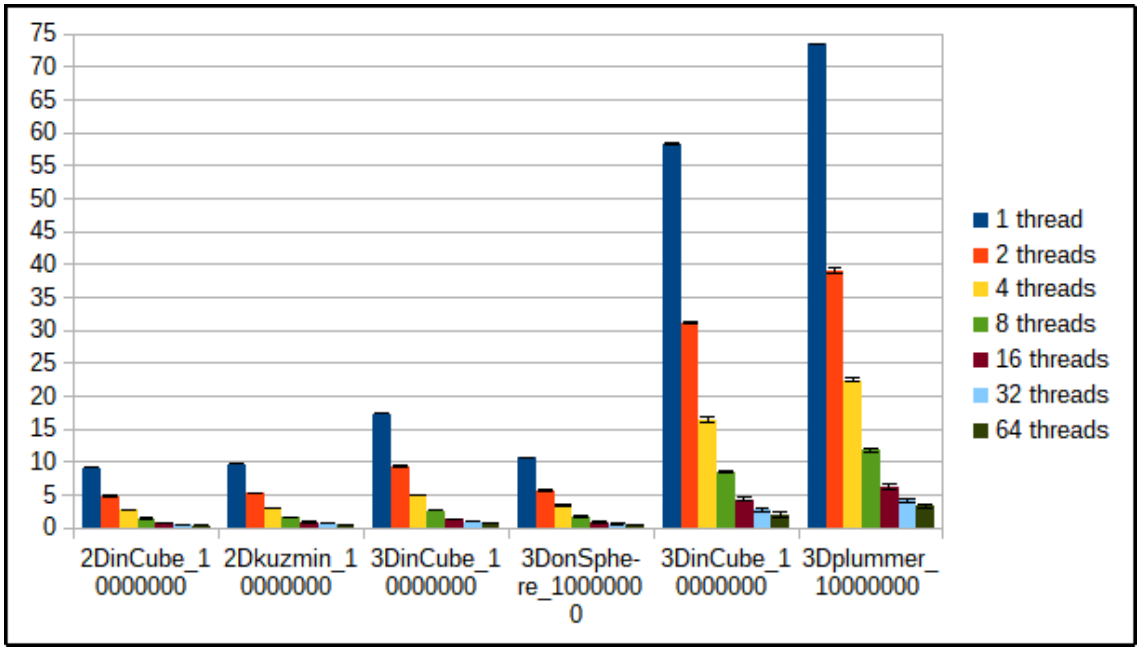


Figure A.274: Execution times (in minutes) of Nearest Neighbors ran on AMD 32-64 using LCWS with WShare (**tit-for-tat**).

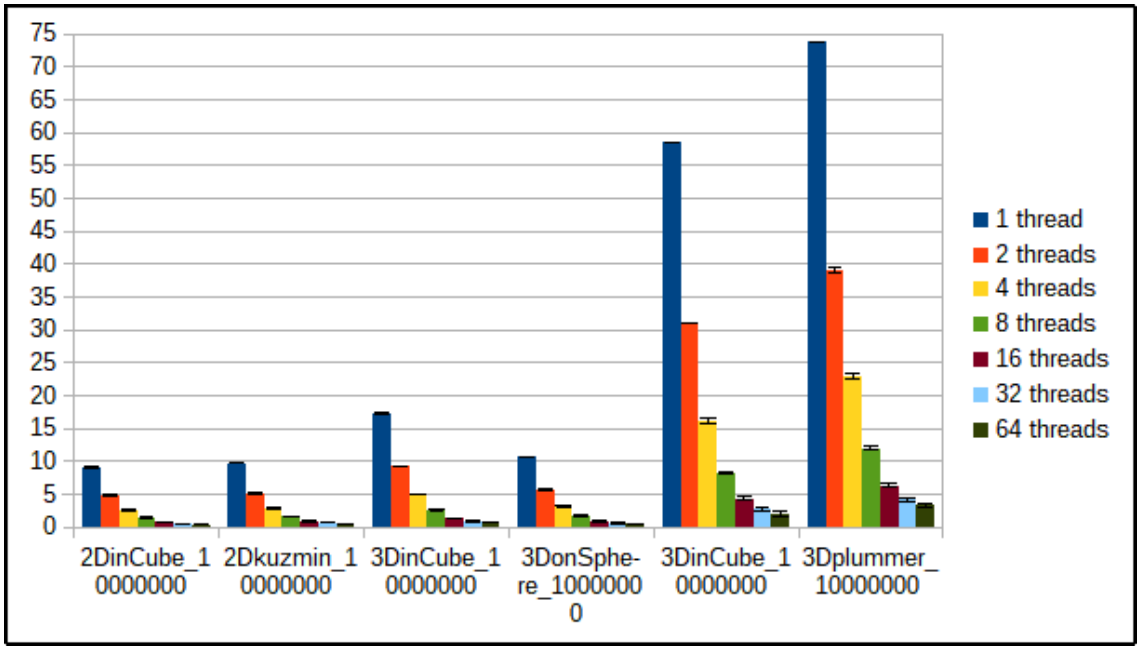


Figure A.275: Execution times (in minutes) of Nearest Neighbors ran on AMD 32-64 using LCWS with WShare (**continuous tit-for-tat**).

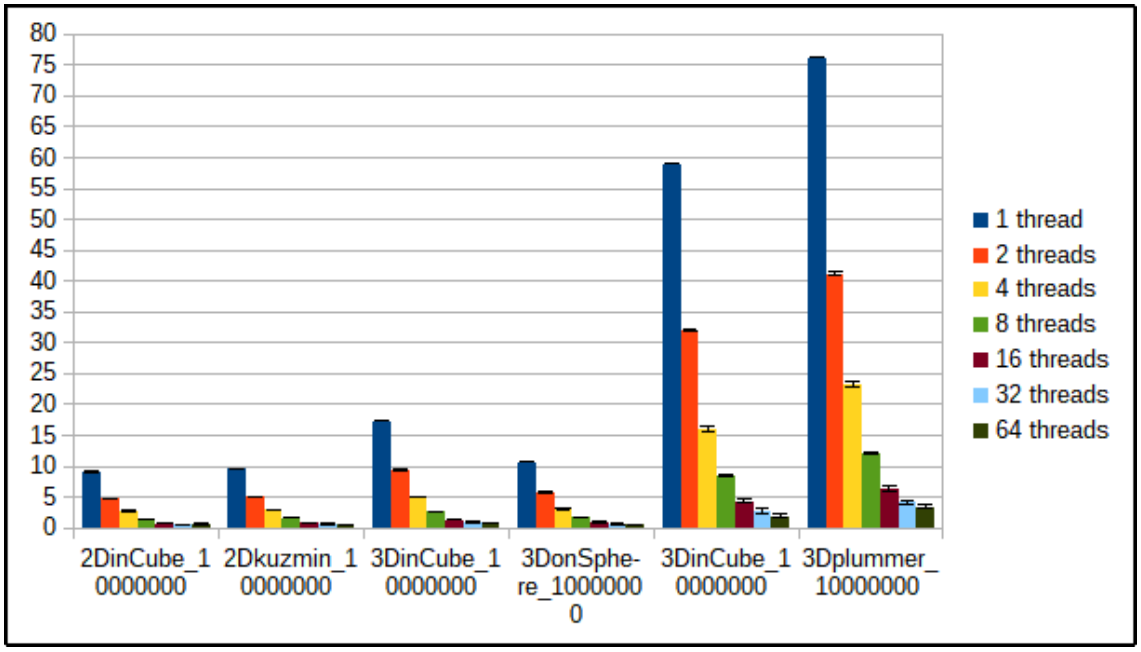


Figure A.276: Execution times (in minutes) of Nearest Neighbors ran on AMD 32-64 using SBLCWS with WShare (**first version**).

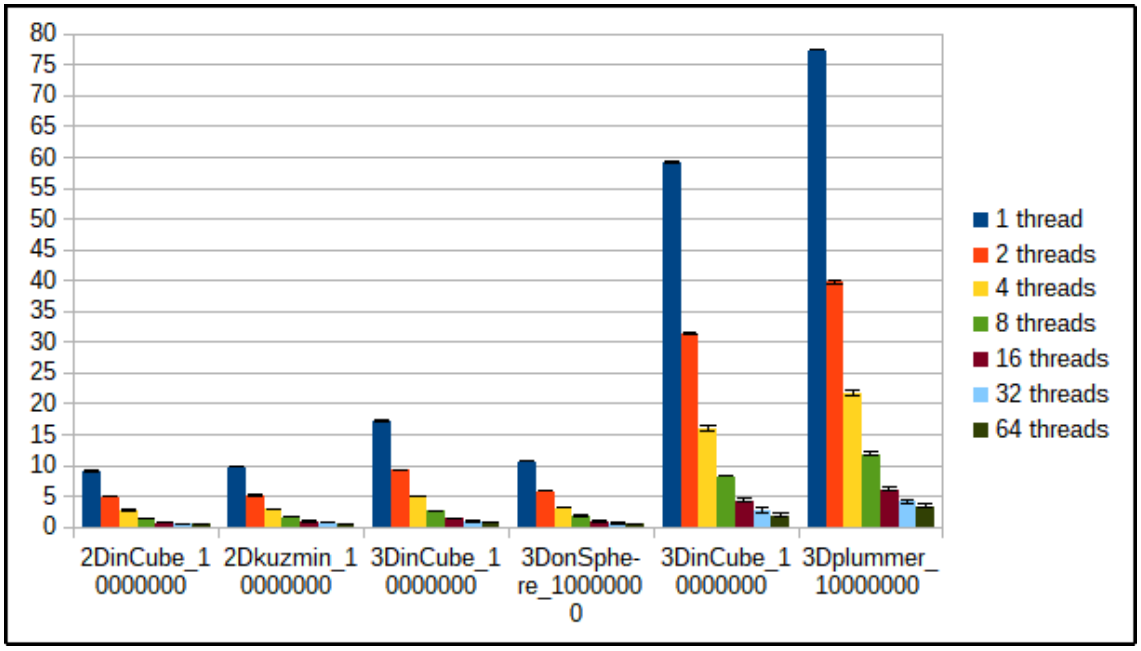


Figure A.277: Execution times (in minutes) of Nearest Neighbors ran on AMD 32-64 using SBLCWS with WShare (**second version**).

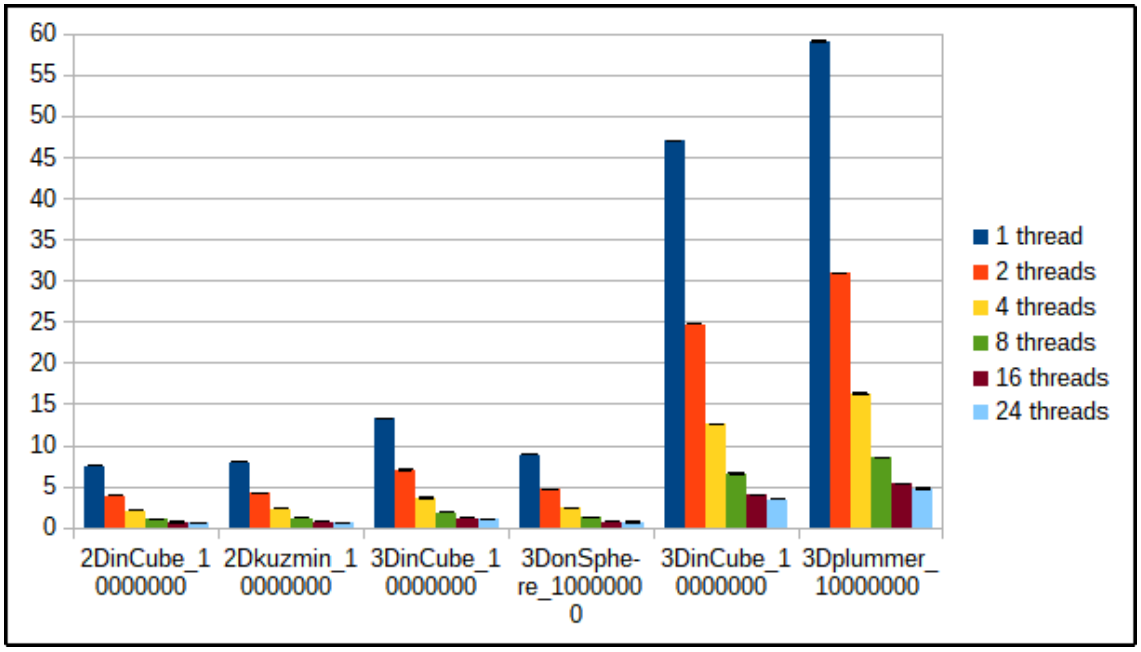


Figure A.278: Execution times (in minutes) of Nearest Neighbors ran on Intel 16-16 using WSteal

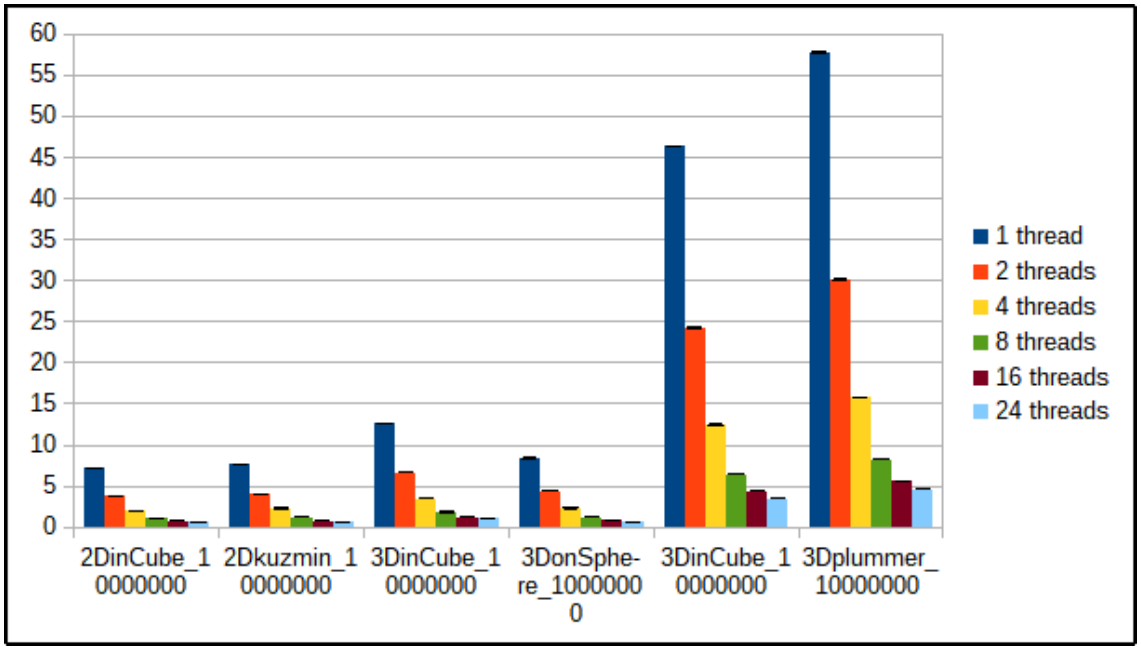


Figure A.279: Execution times (in minutes) of Nearest Neighbors ran on Intel 16-16 using LCWS

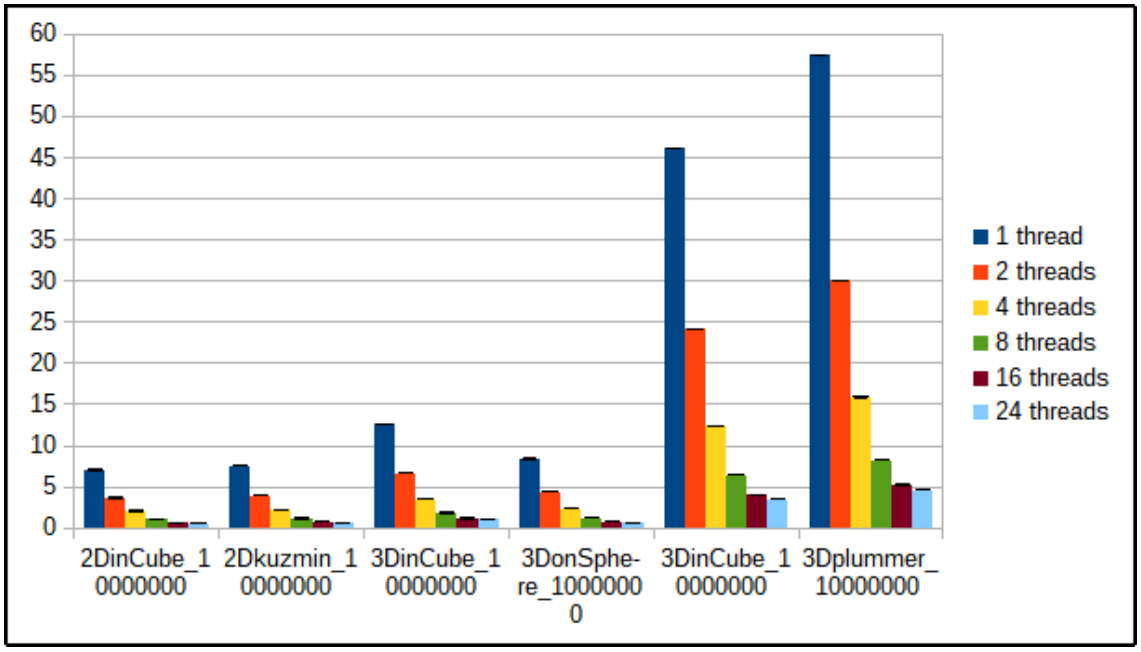


Figure A.280: Execution times (in minutes) of Nearest Neighbors ran on Intel 16-16 using SBLCWS (**first version**)

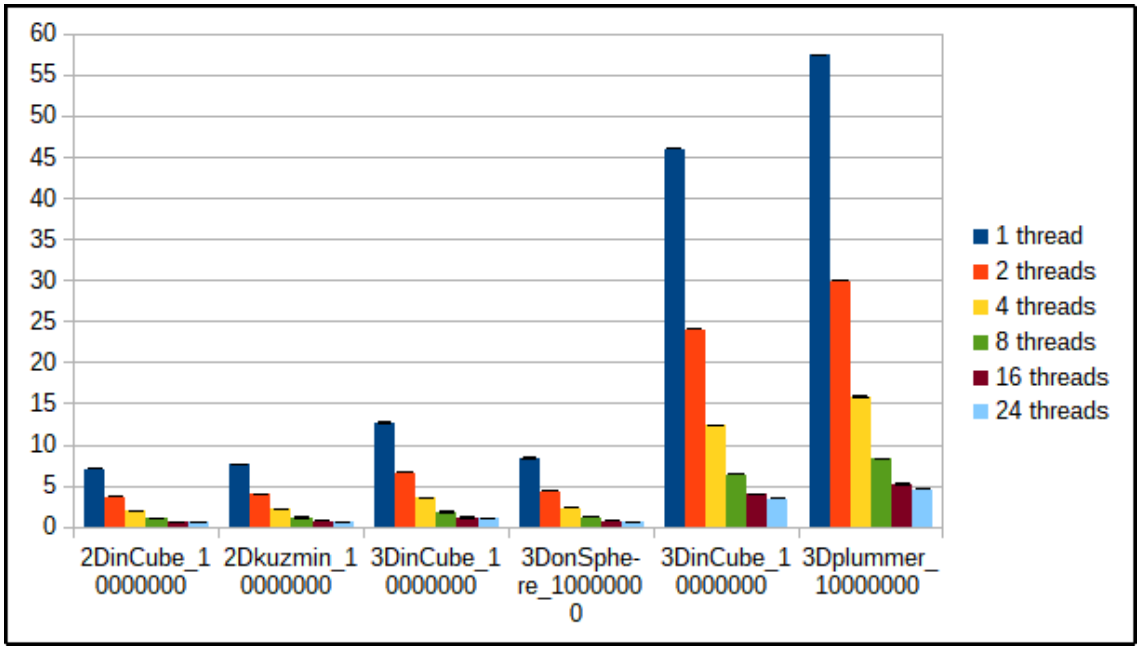


Figure A.281: Execution times (in minutes) of Nearest Neighbors ran on Intel 16-16 using SBLCWS (**second version**)

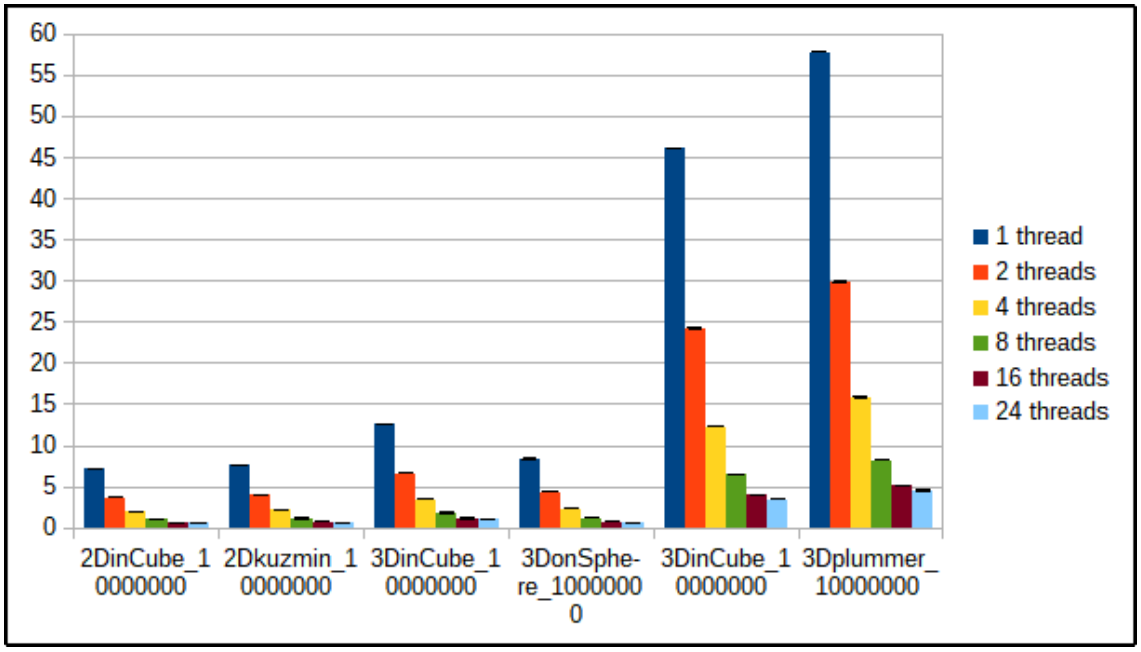


Figure A.282: Execution times (in minutes) of Nearest Neighbors ran on Intel 16-16 using LCWS with WShare (**tit-for-tat**).

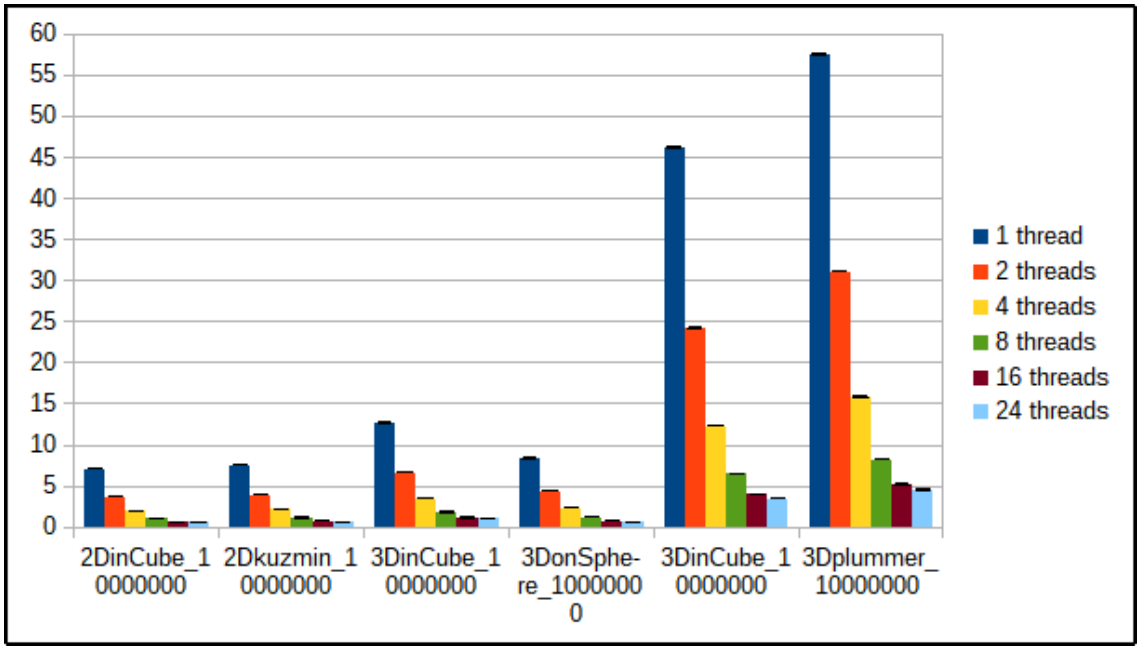


Figure A.283: Execution times (in minutes) of Nearest Neighbors ran on Intel 16-16 using LCWS with WShare (**continuous tit-for-tat**).

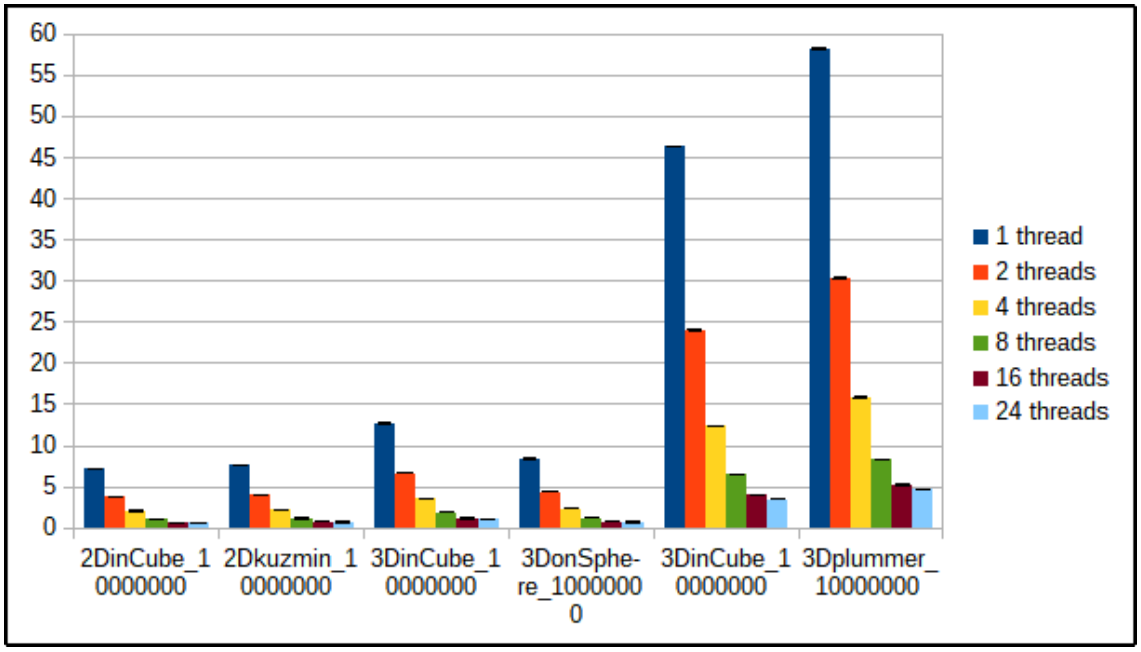


Figure A.284: Execution times (in minutes) of Nearest Neighbors ran on Intel 16-16 using SBLCWS with WShare (**first version**).

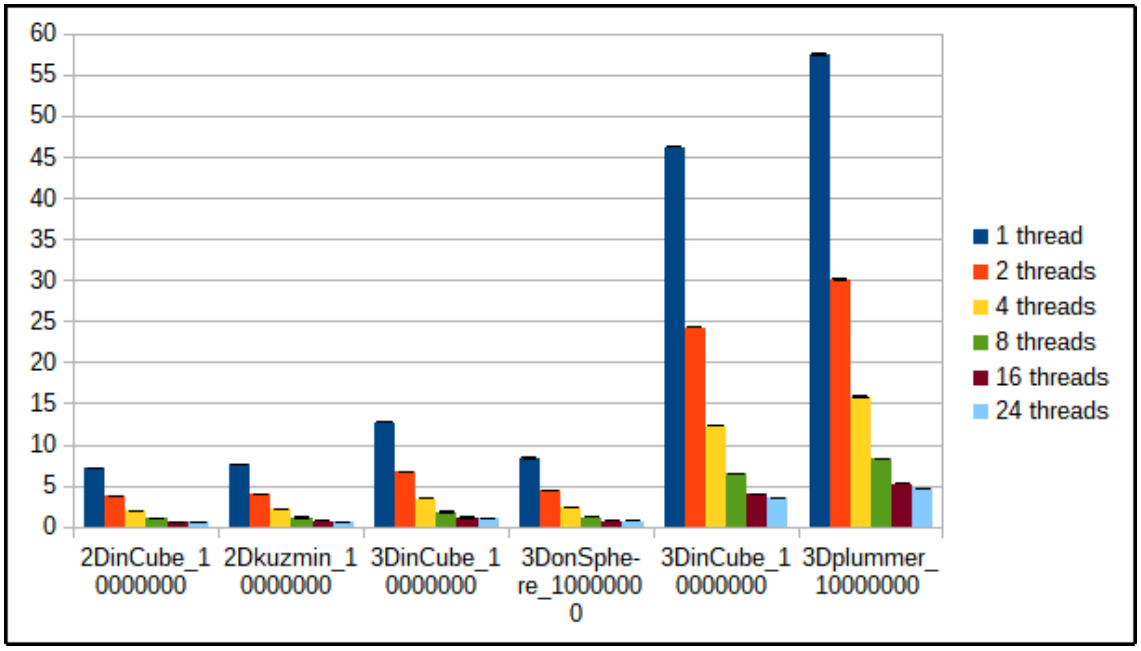


Figure A.285: Execution times (in minutes) of Nearest Neighbors ran on Intel 16-16 using SBLCWS with WShare (**second version**).

A.13 Convex Hull

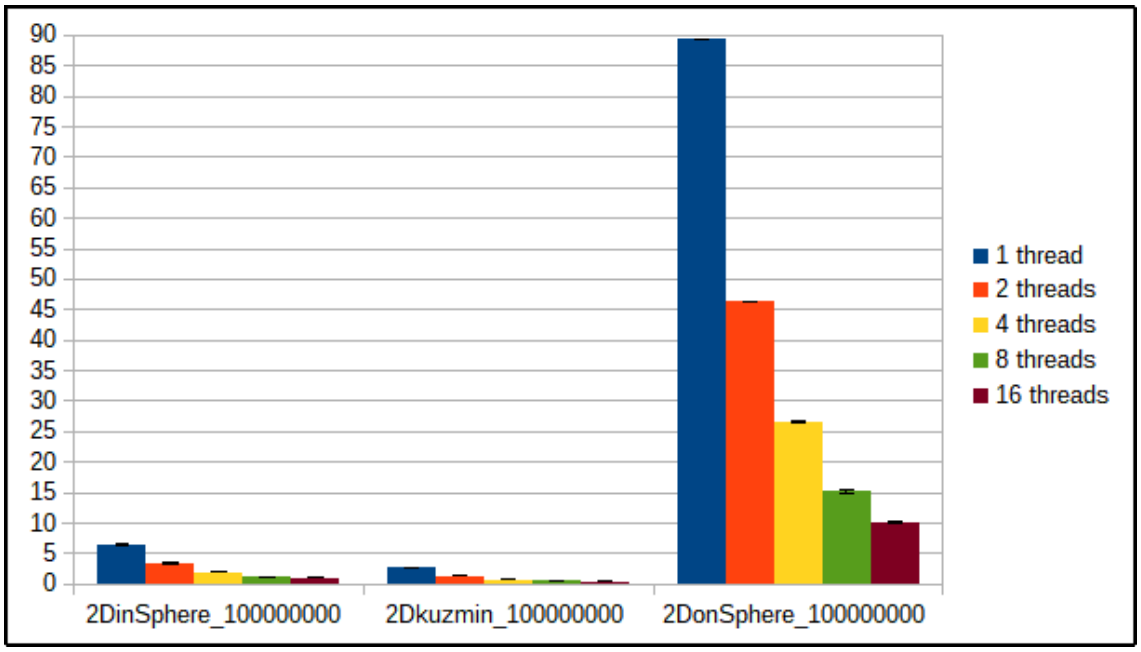


Figure A.286: Execution times (in minutes) of Convex Hull ran on Intel 12-24 using WSteal

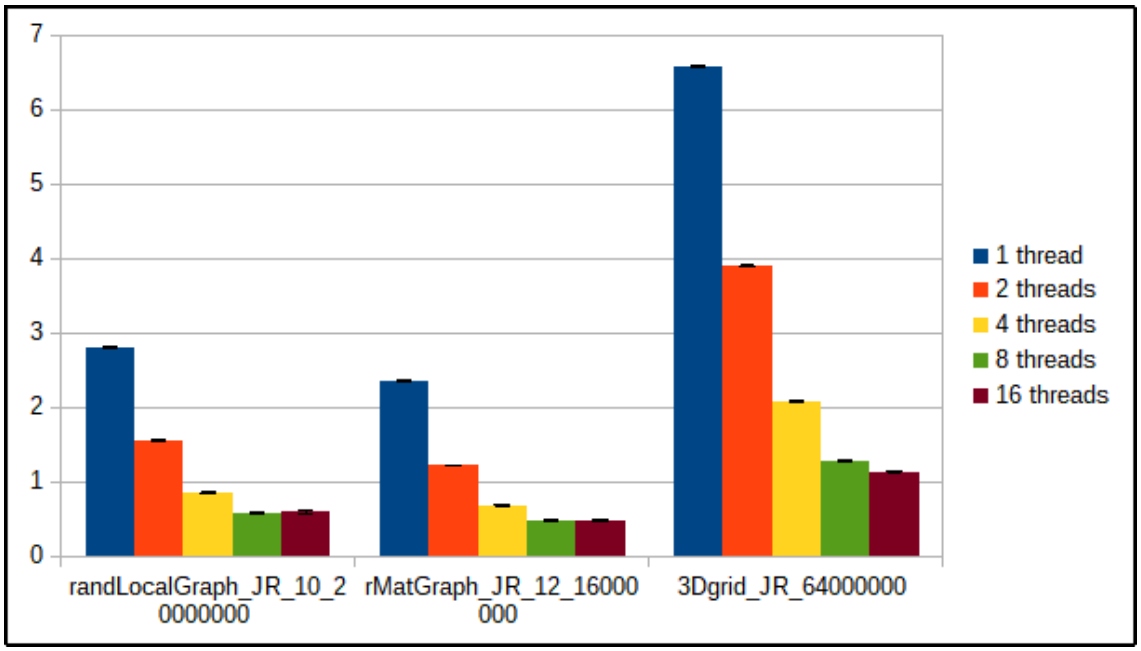


Figure A.287: Execution times (in minutes) of Convex Hull ran on Intel 12-24 using LCWS

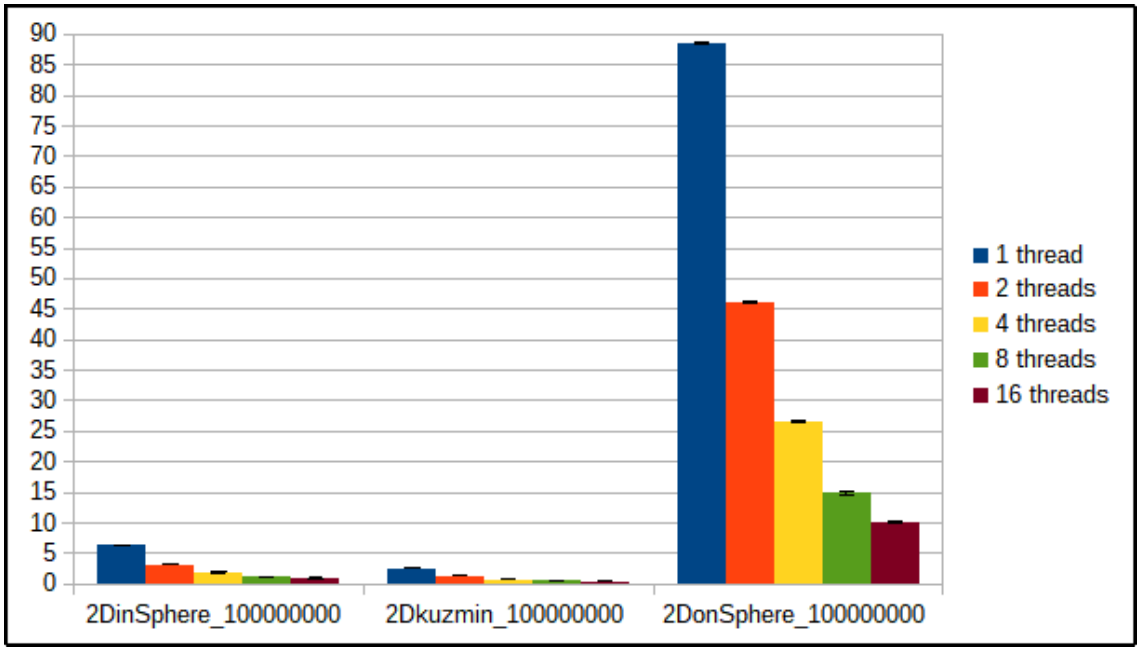


Figure A.288: Execution times (in minutes) of Convex Hull ran on Intel 12-24 using SBLCWS (**first version**)

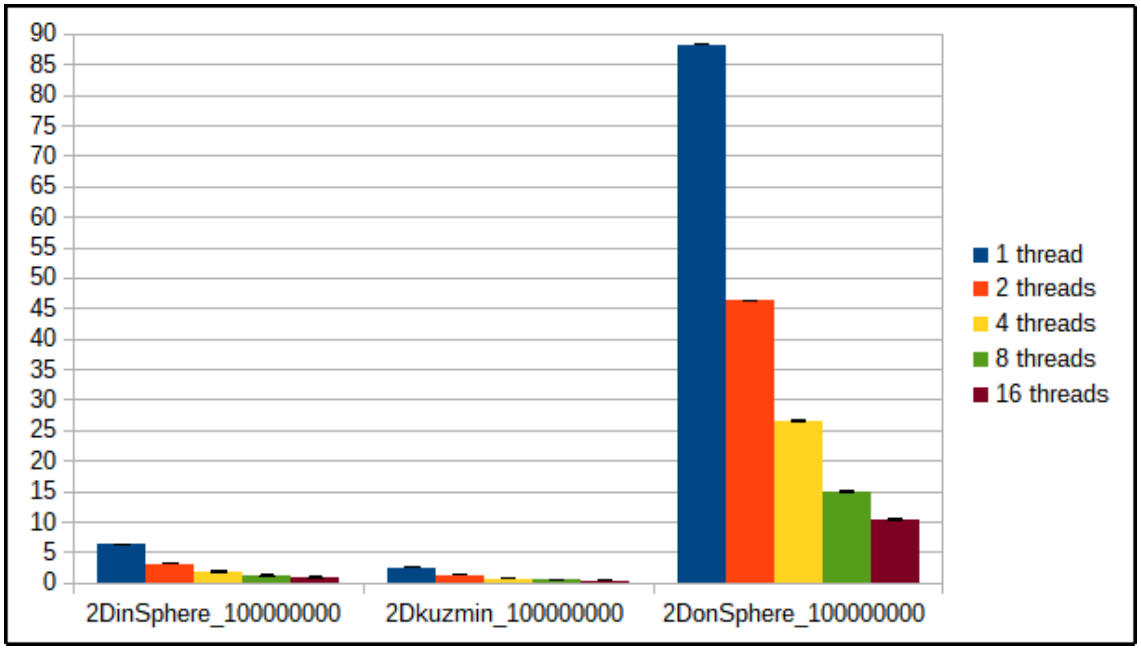


Figure A.289: Execution times (in minutes) of Convex Hull ran on Intel 12-24 using SBLCWS (**second version**)

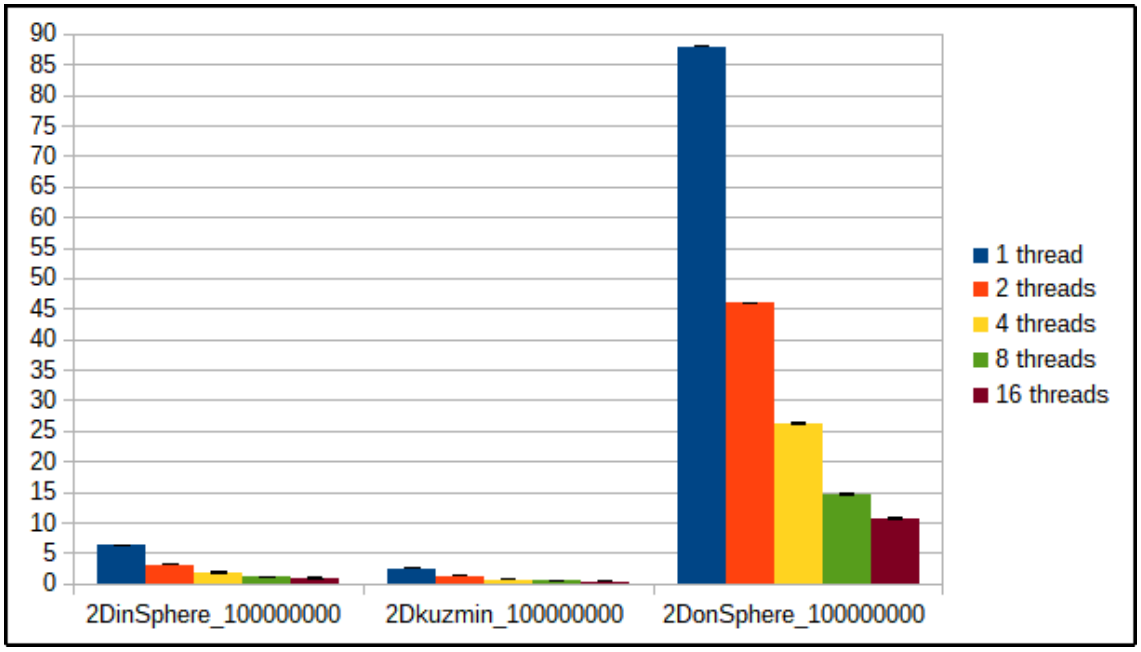


Figure A.290: Execution times (in minutes) of Convex Hull ran on Intel 12-24 using LCWS with WShare (**tit-for-tat**).

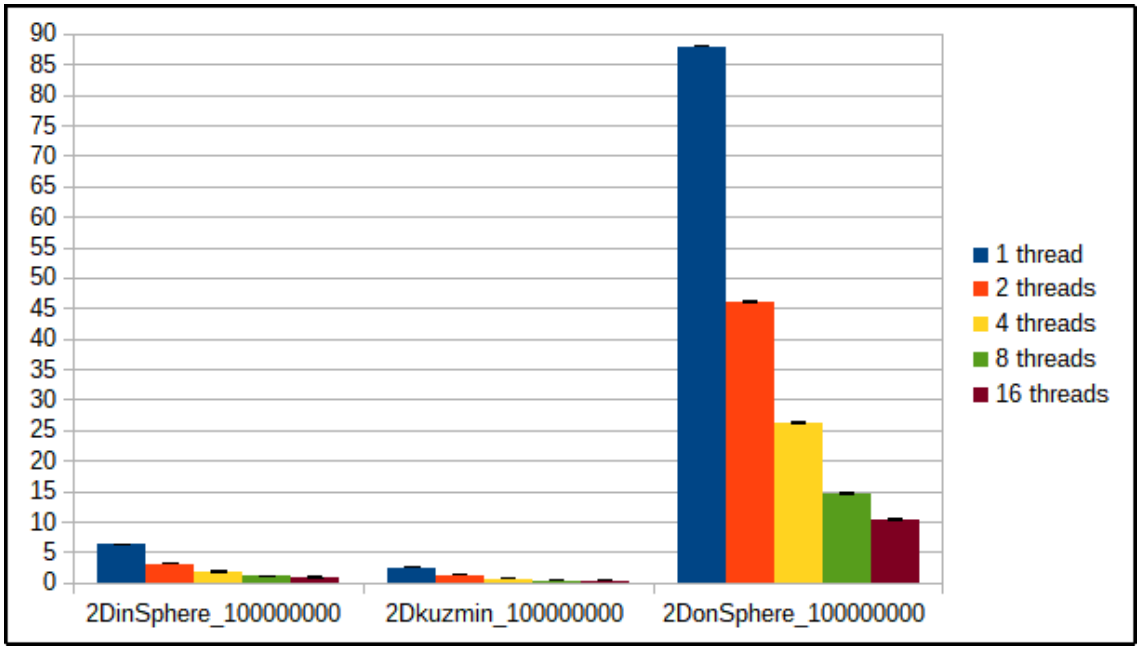


Figure A.291: Execution times (in minutes) of Convex Hull ran on Intel 12-24 using LCWS with WShare (**continuous tit-for-tat**).

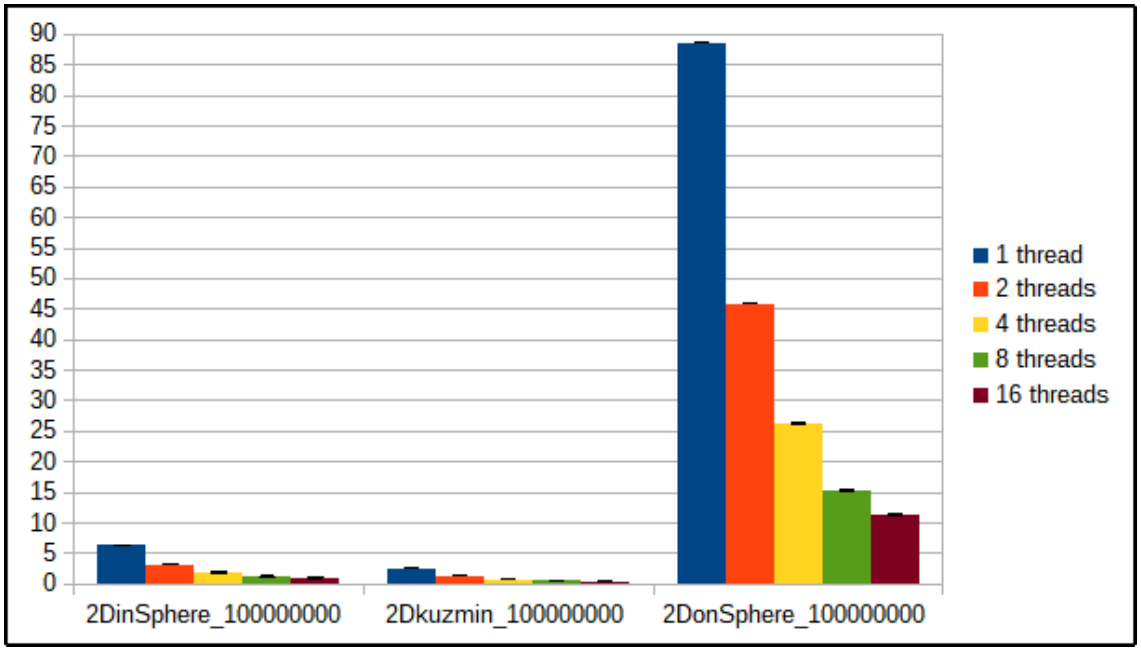


Figure A.292: Execution times (in minutes) of Convex Hull ran on Intel 12-24 using SBLCWS with WShare (**first version**).

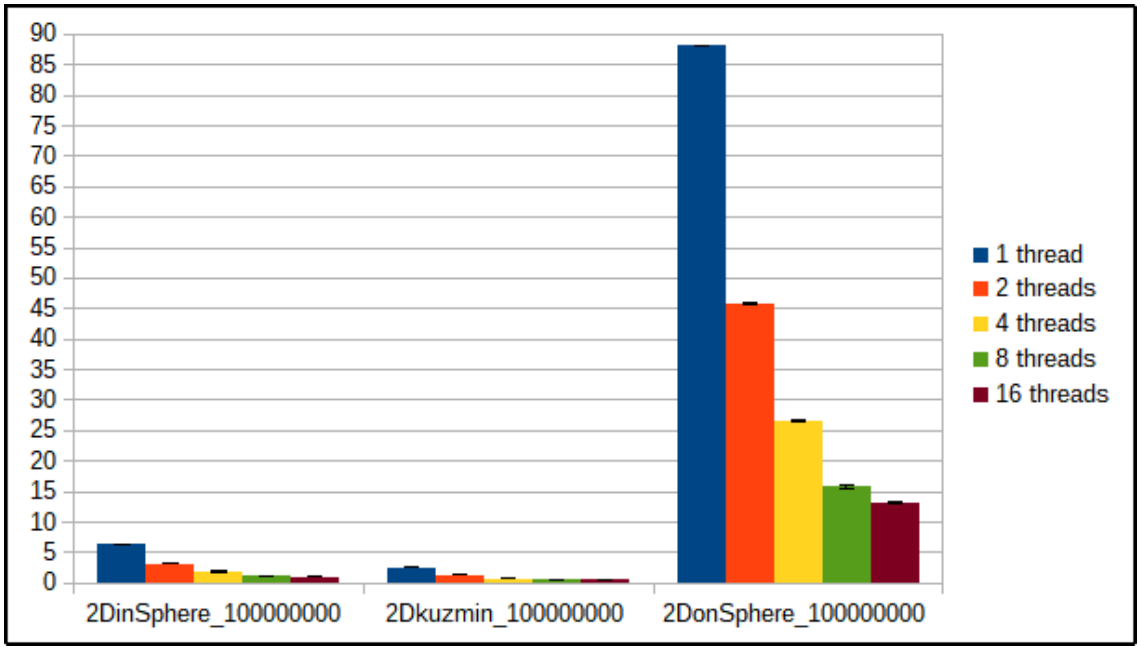


Figure A.293: Execution times (in minutes) of Convex Hull ran on Intel 12-24 using SBLCWS with WShare (**second version**).

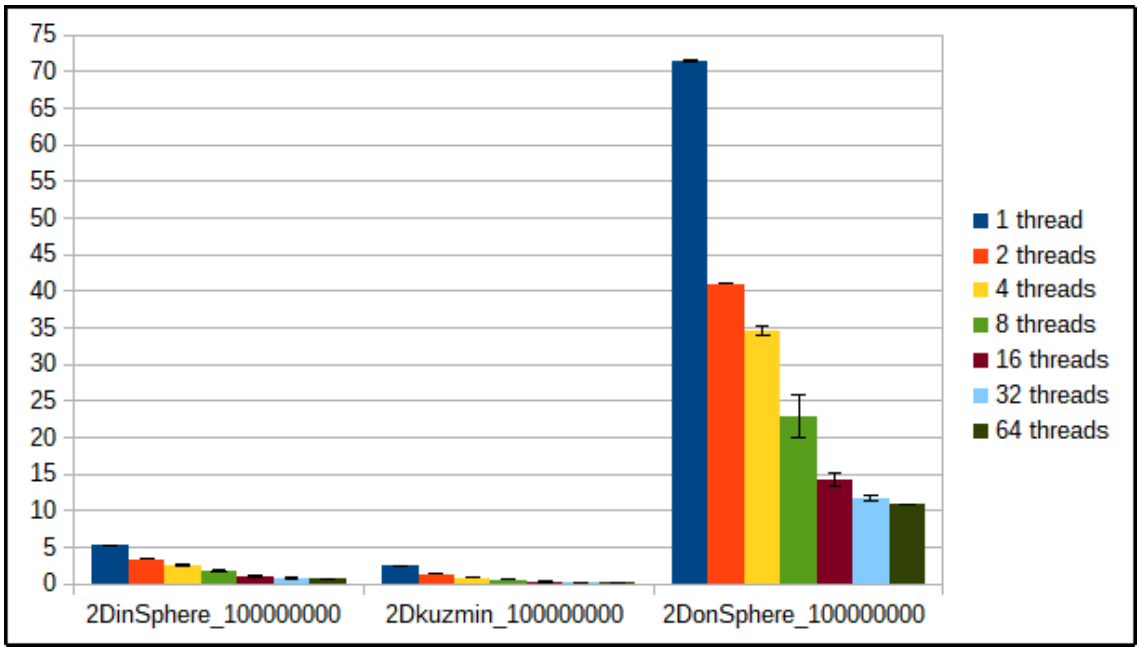


Figure A.294: Execution times (in minutes) of Convex Hull ran on AMD 32-64 using WSteal

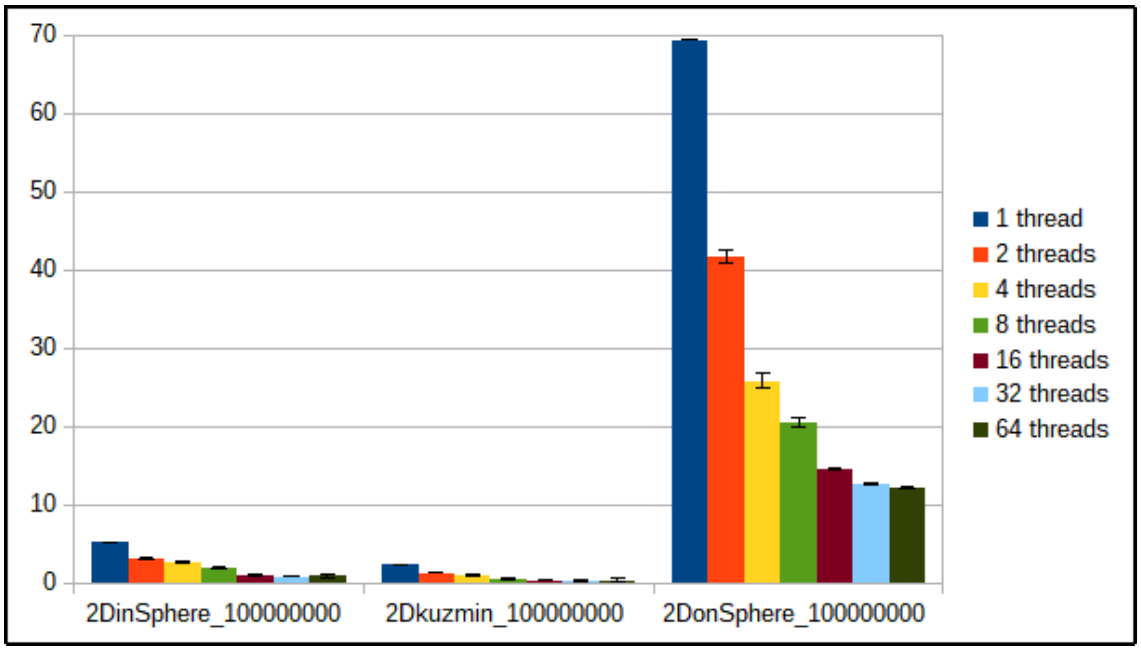


Figure A.295: Execution times (in minutes) of Convex Hull ran on AMD 32-64 using LCWS

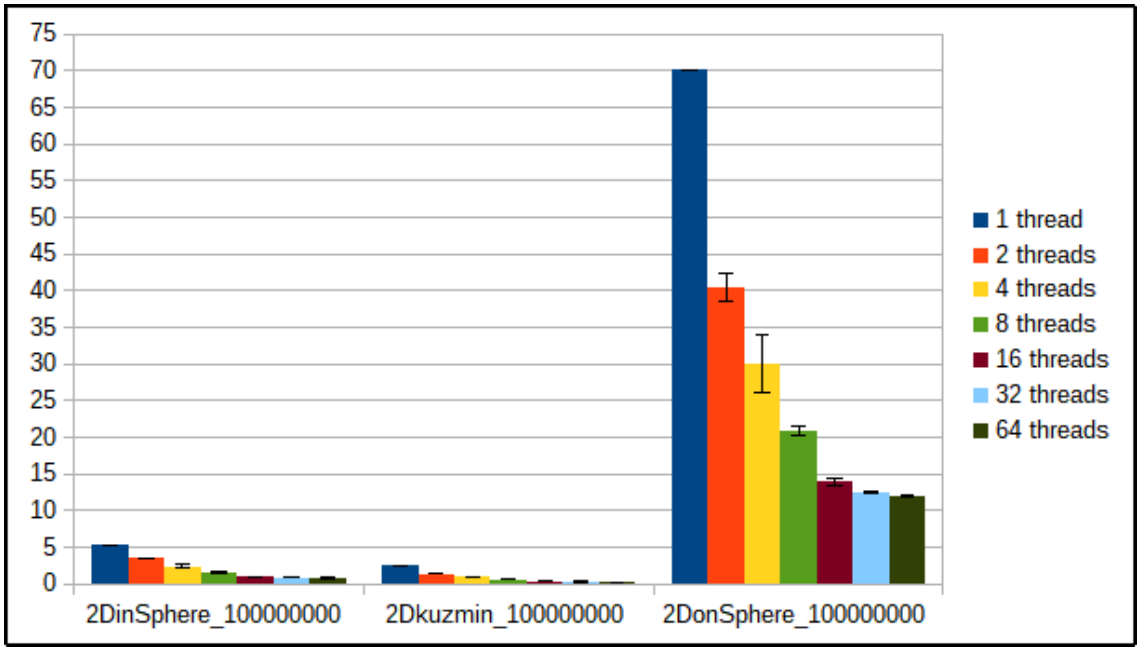


Figure A.296: Execution times (in minutes) of Convex Hull ran on AMD 32-64 using SBLCWS (**first version**)

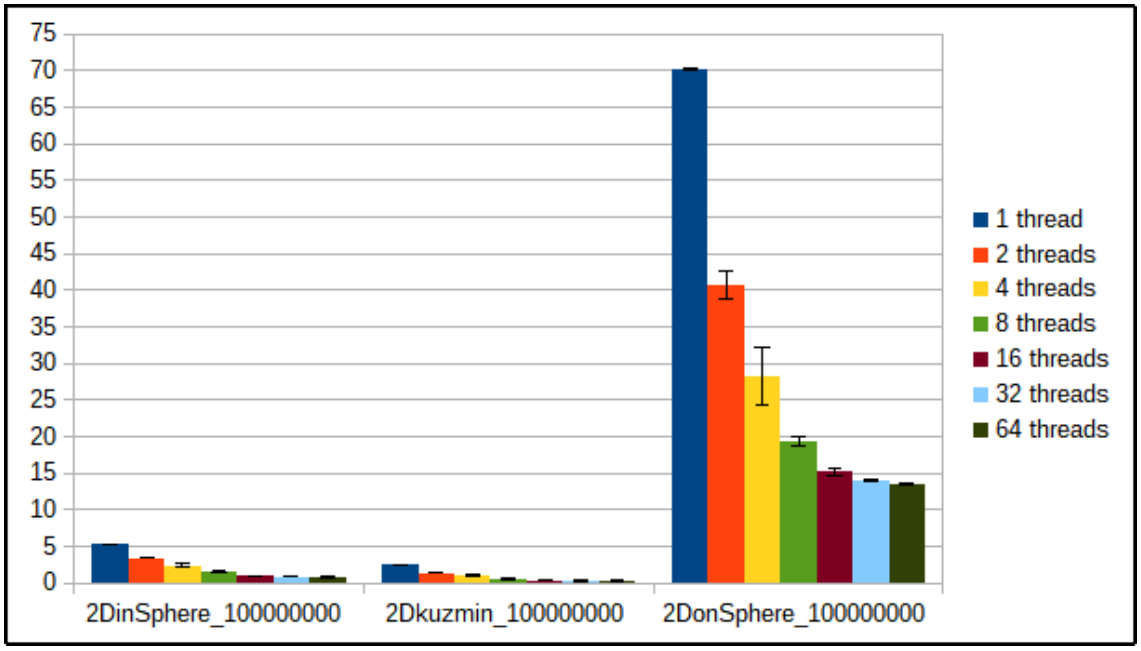


Figure A.297: Execution times (in minutes) of Convex Hull ran on AMD 32-64 using SBLCWS (**second version**)

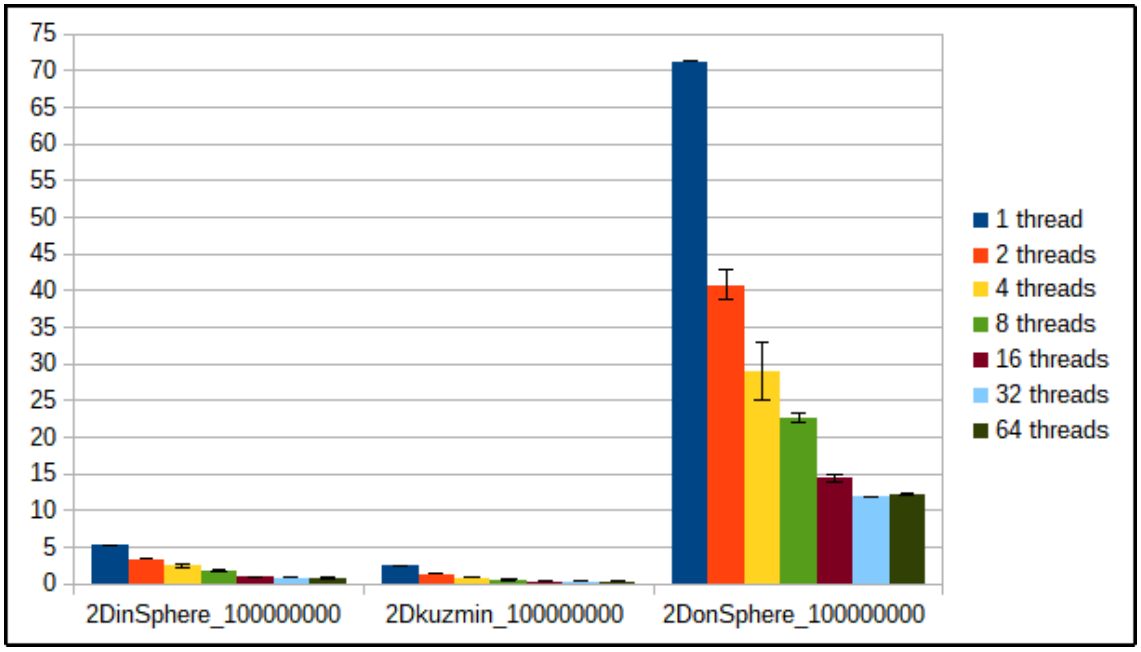


Figure A.298: Execution times (in minutes) of Convex Hull ran on AMD 32-64 using LCWS with WShare (**tit-for-tat**).

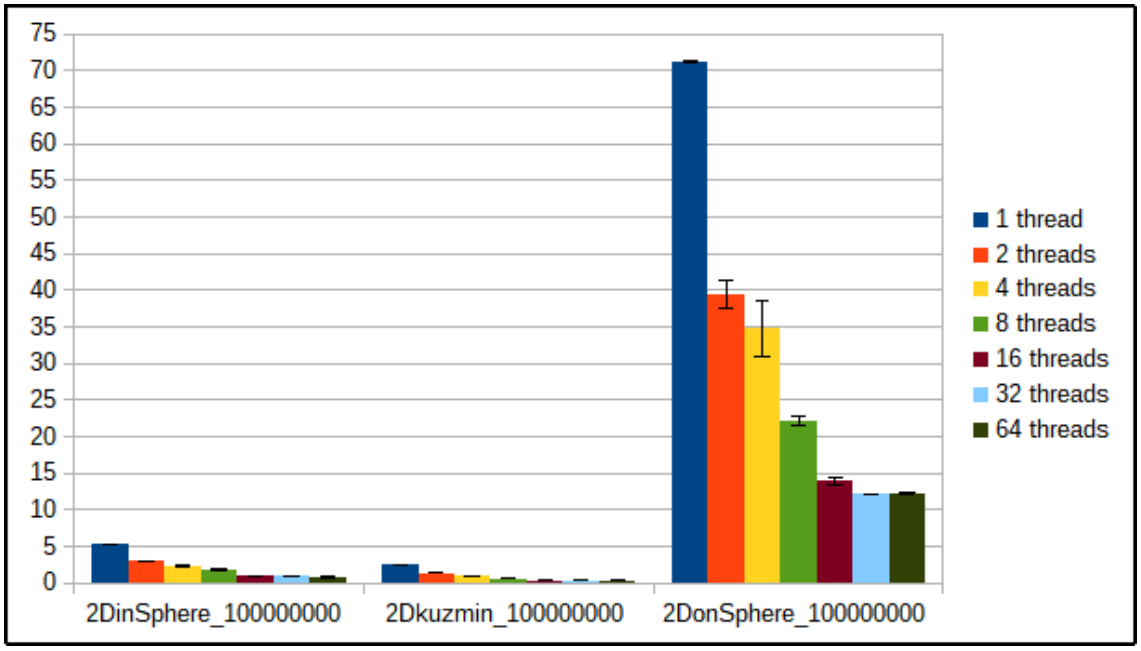


Figure A.299: Execution times (in minutes) of Convex Hull ran on AMD 32-64 using LCWS with WShare (**continuous tit-for-tat**).

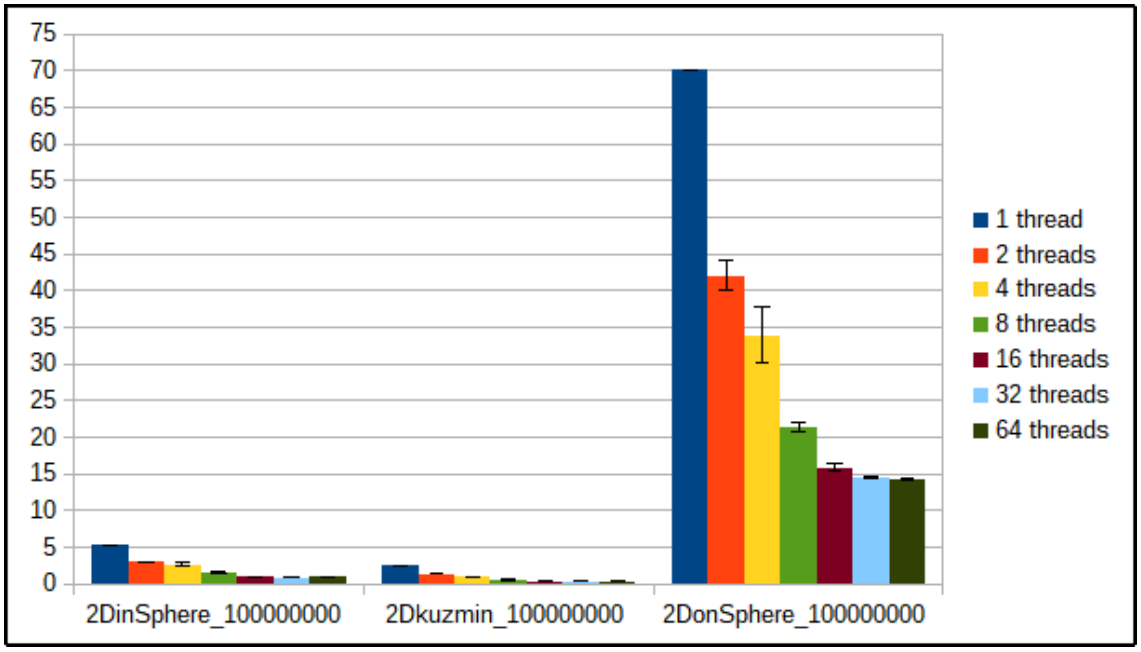


Figure A.300: Execution times (in minutes) of Convex Hull ran on AMD 32-64 using SBLCWS with WShare (**first version**).

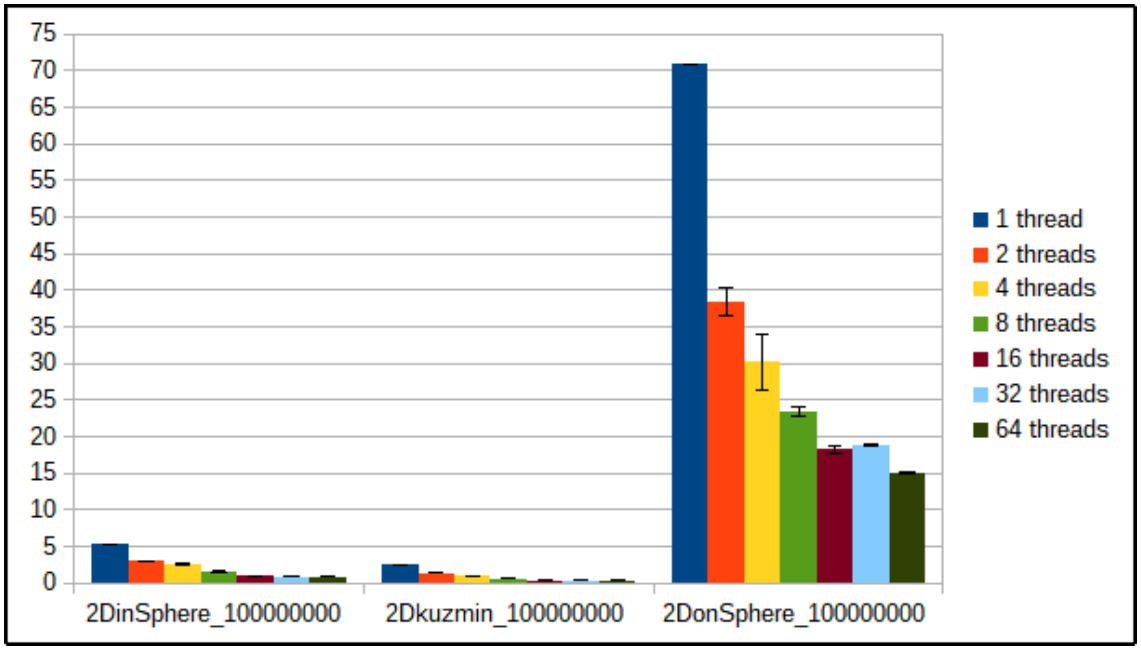


Figure A.301: Execution times (in minutes) of Convex Hull ran on AMD 32-64 using SBLCWS with WShare (**second version**).

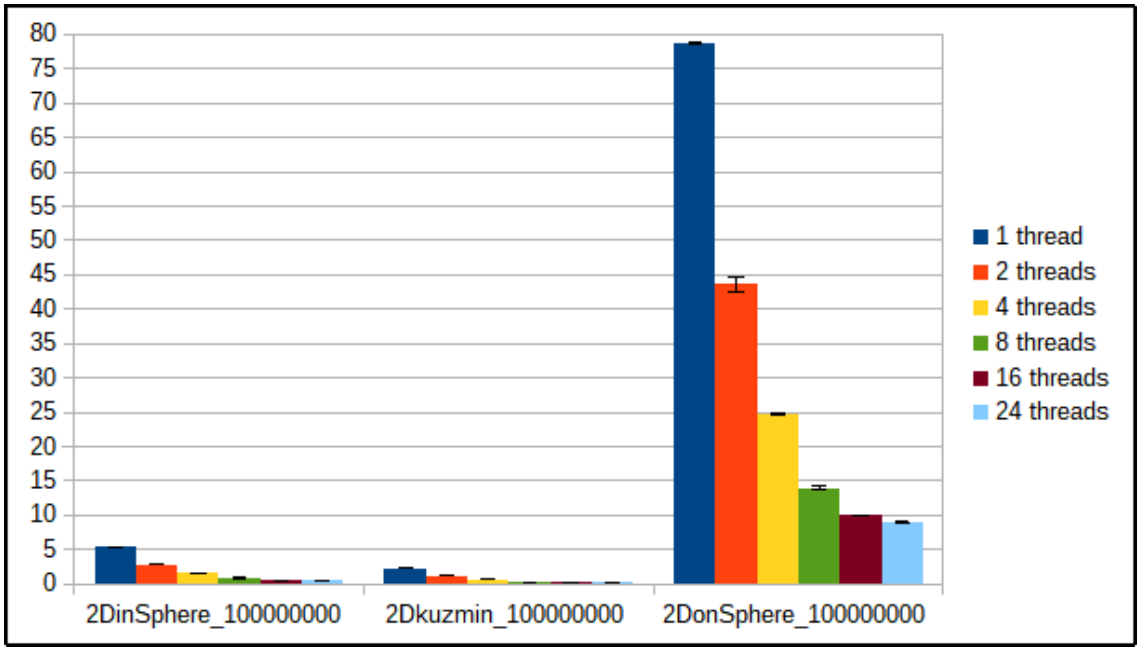


Figure A.302: Execution times (in minutes) of Convex Hull ran on Intel 16-16 using WSteal

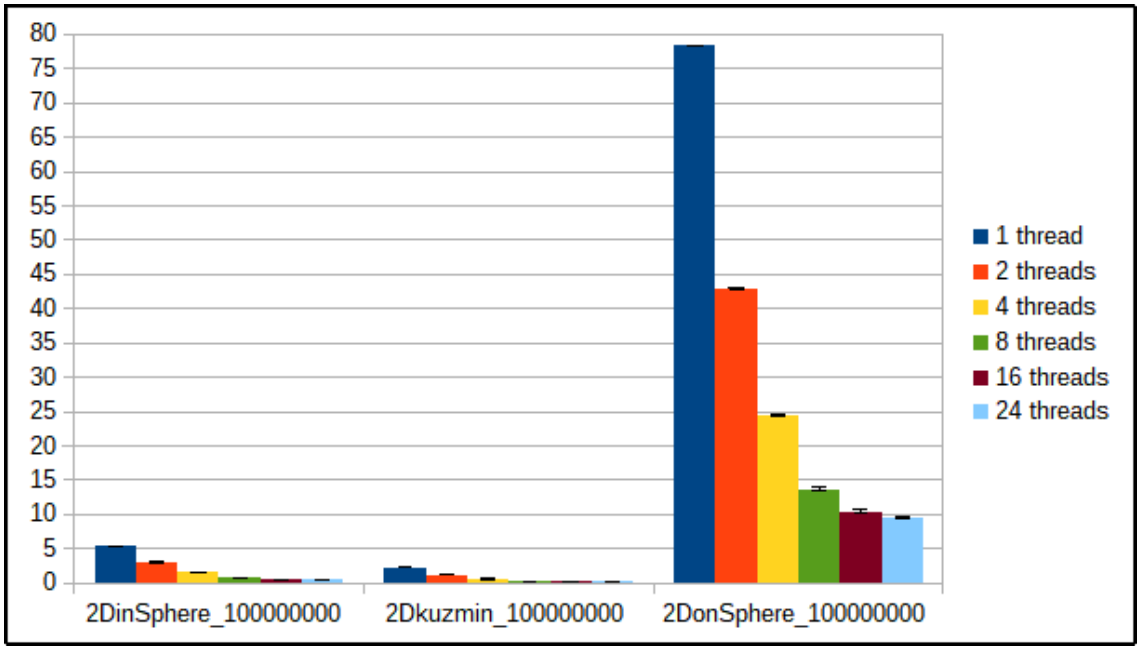


Figure A.303: Execution times (in minutes) of Convex Hull ran on Intel 16-16 using LCWS

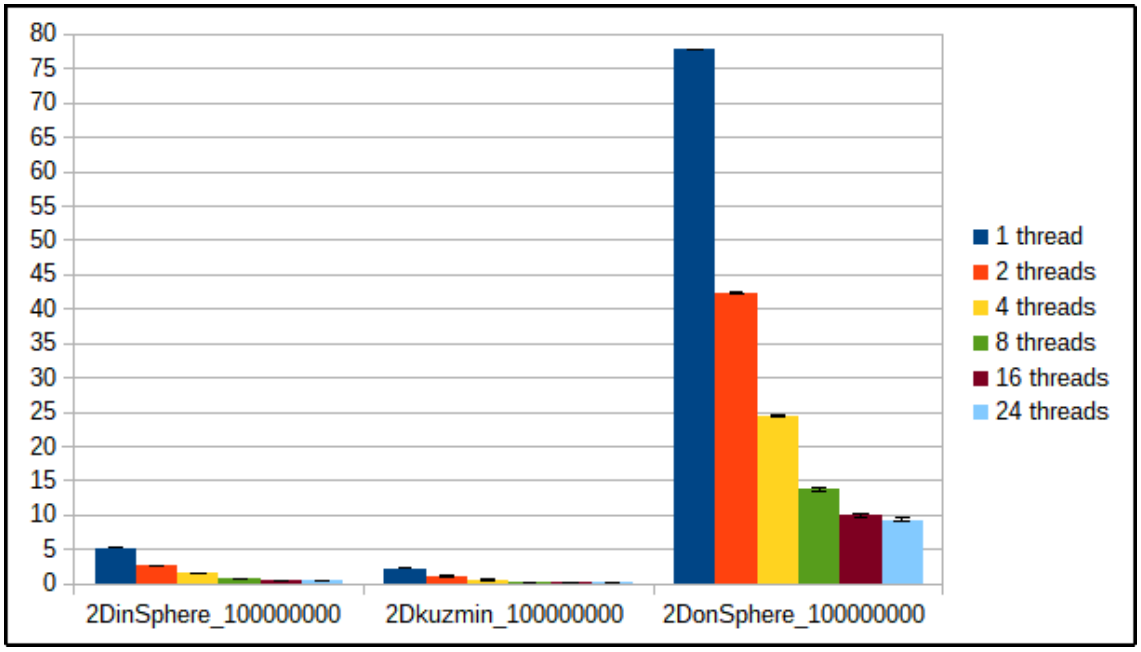


Figure A.304: Execution times (in minutes) of Convex Hull ran on Intel 16-16 using SBLCWS (**first version**)

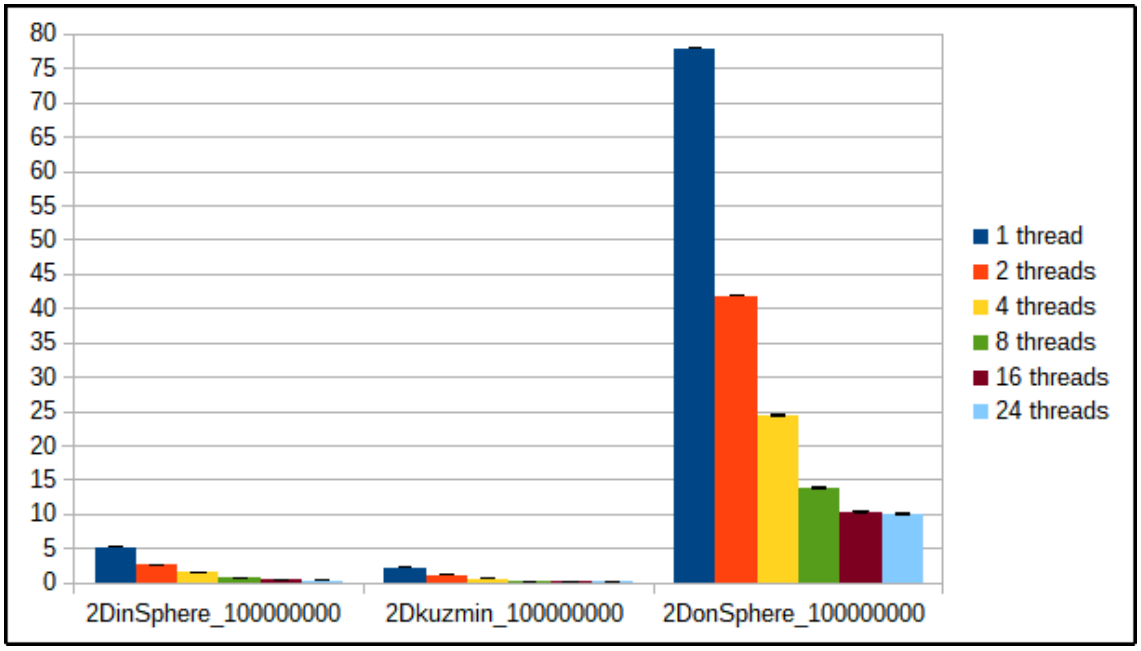


Figure A.305: Execution times (in minutes) of Convex Hull ran on Intel 16-16 using SBLCWS (**second version**)

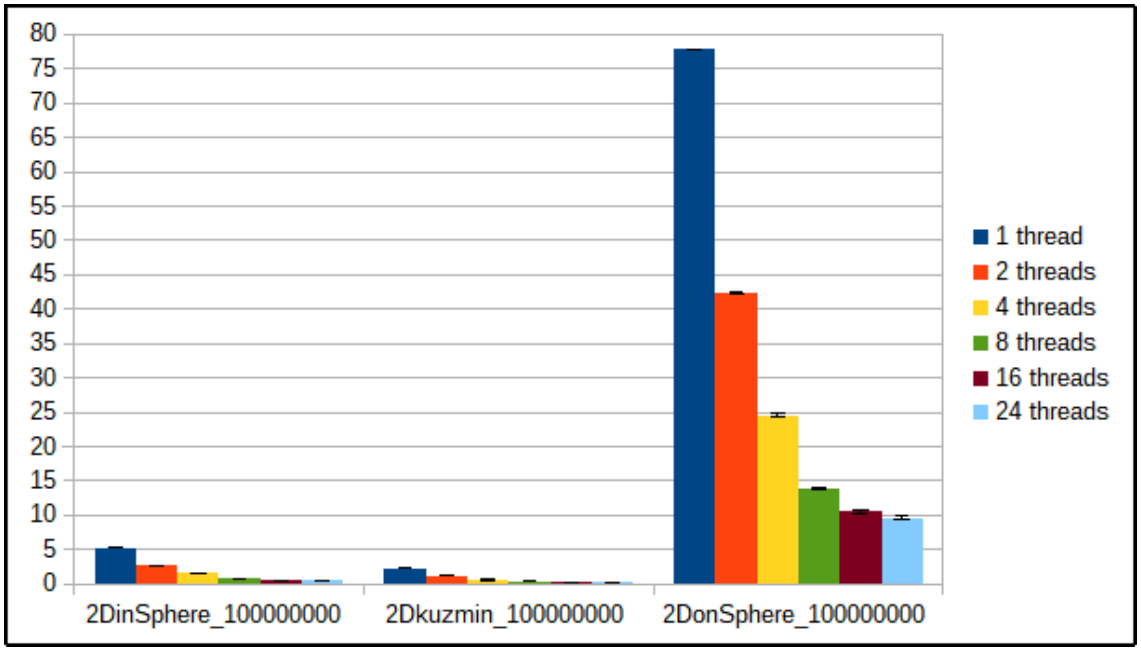


Figure A.306: Execution times (in minutes) of Convex Hull ran on Intel 16-16 using LCWS with WShare (**tit-for-tat**).

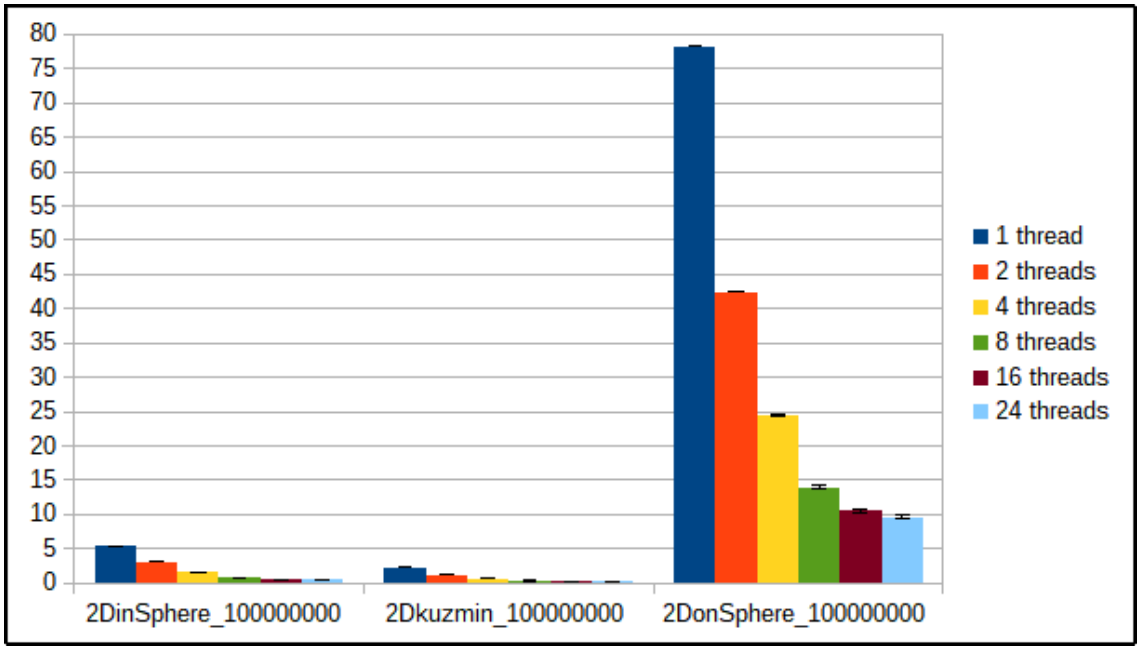


Figure A.307: Execution times (in minutes) of Convex Hull ran on Intel 16-16 using LCWS with WShare (**continuous tit-for-tat**).

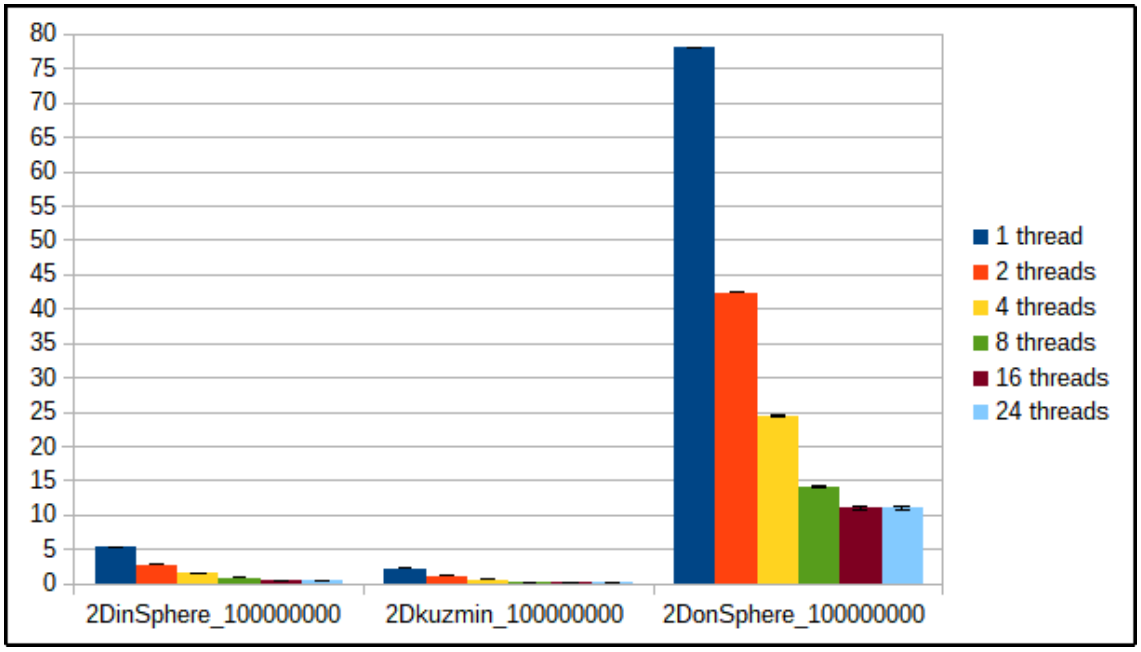


Figure A.308: Execution times (in minutes) of Convex Hull ran on Intel 16-16 using SBLCWS with WShare (**first version**).

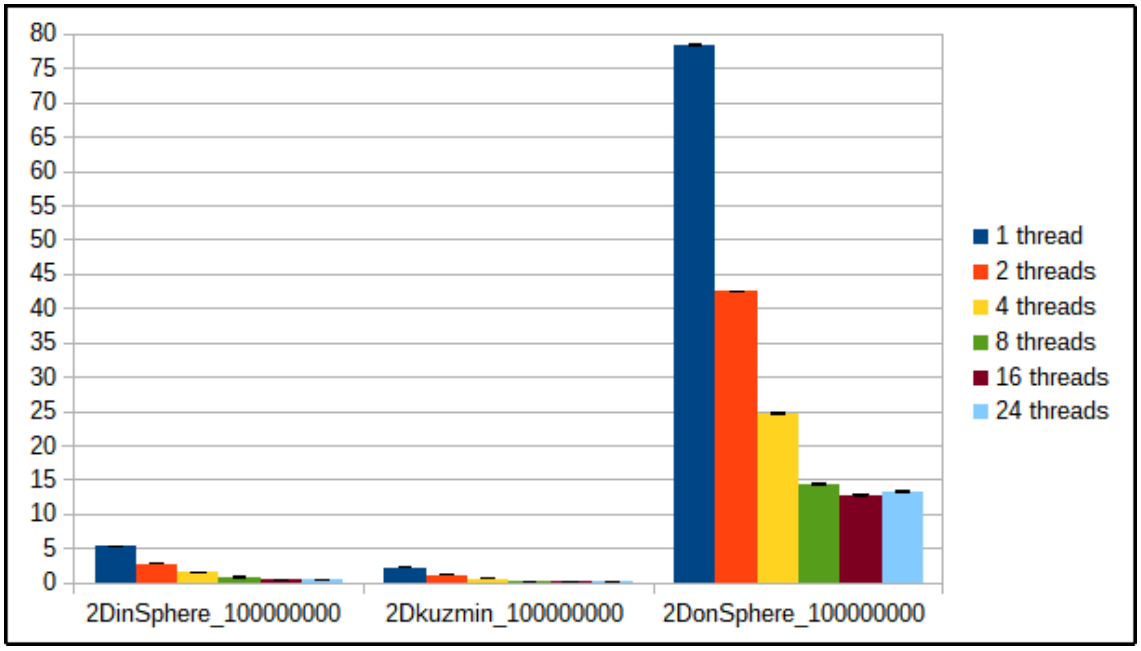


Figure A.309: Execution times (in minutes) of Convex Hull ran on Intel 16-16 using SBLCWS with WShare (**second version**).

A.14 Delaunay Triangulation

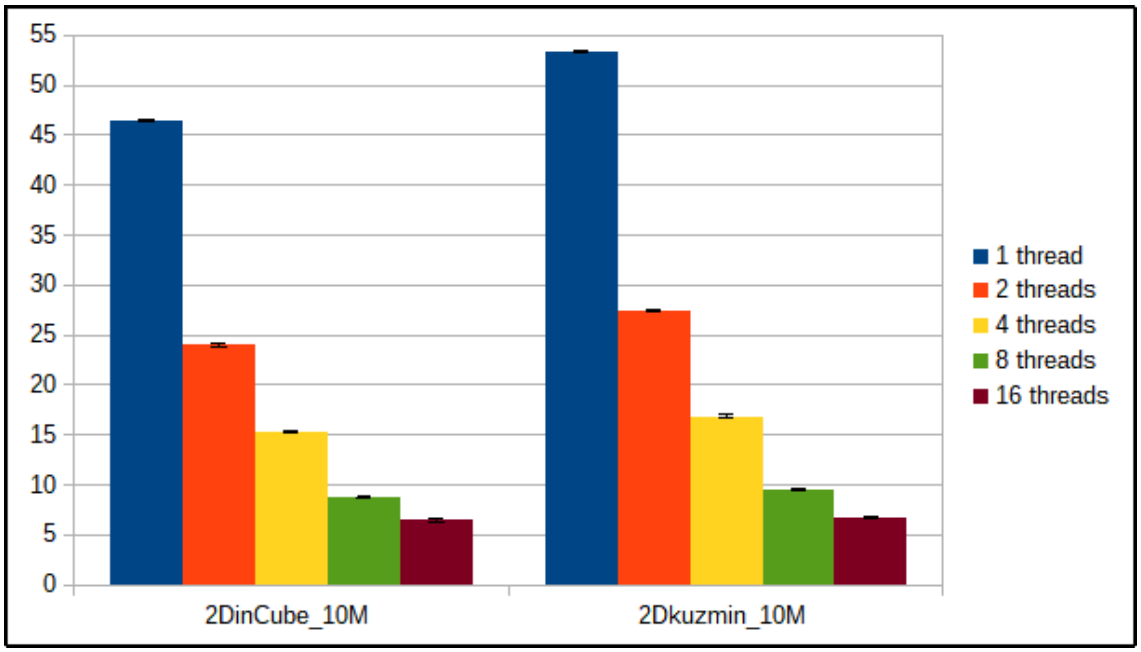


Figure A.310: Execution times (in minutes) of Delaunay Triangulation ran on Intel 12-24 using WSteal

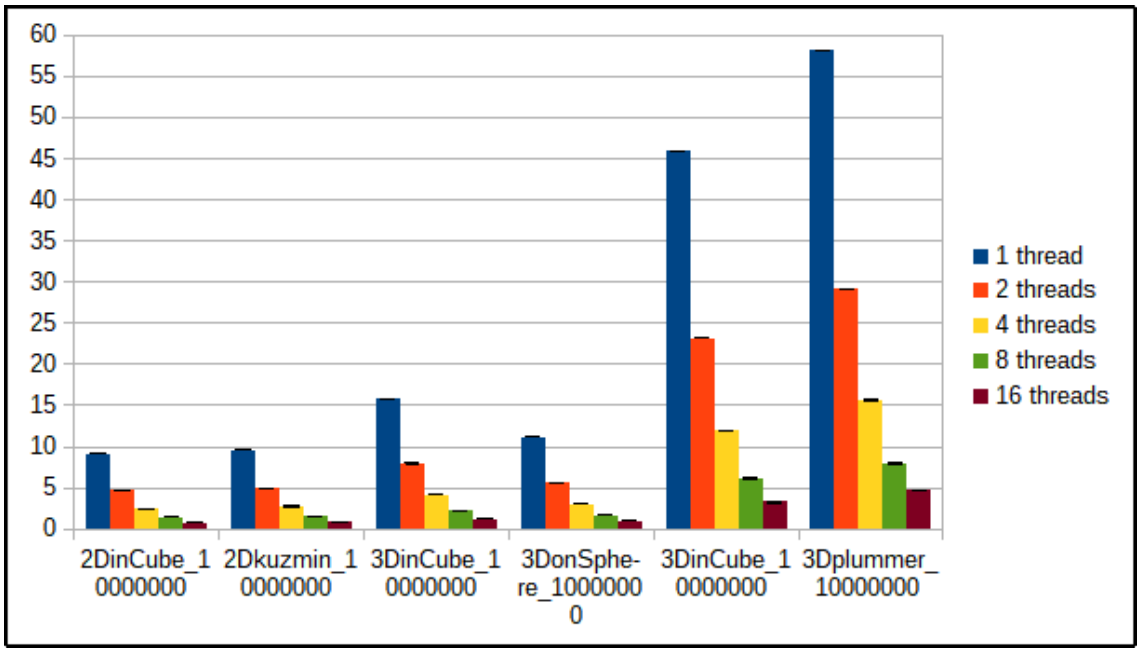


Figure A.311: Execution times (in minutes) of Delaunay Triangulation ran on Intel 12-24 using LCWS

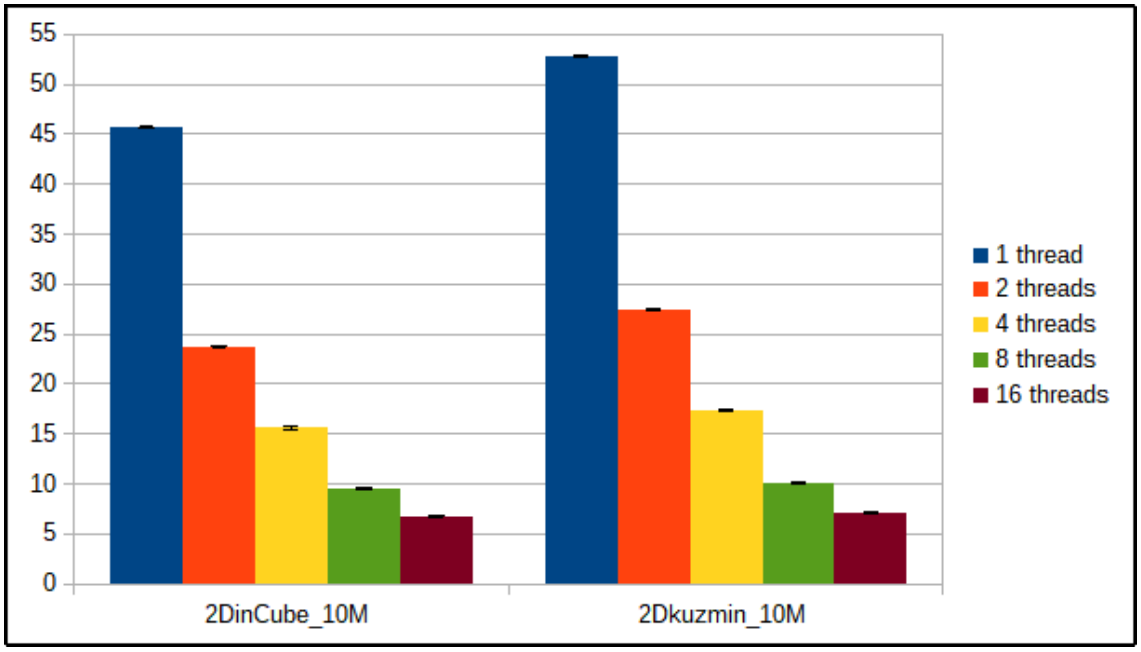


Figure A.312: Execution times (in minutes) of Delaunay Triangulation ran on Intel 12-24 using SBLCWS (**first version**)

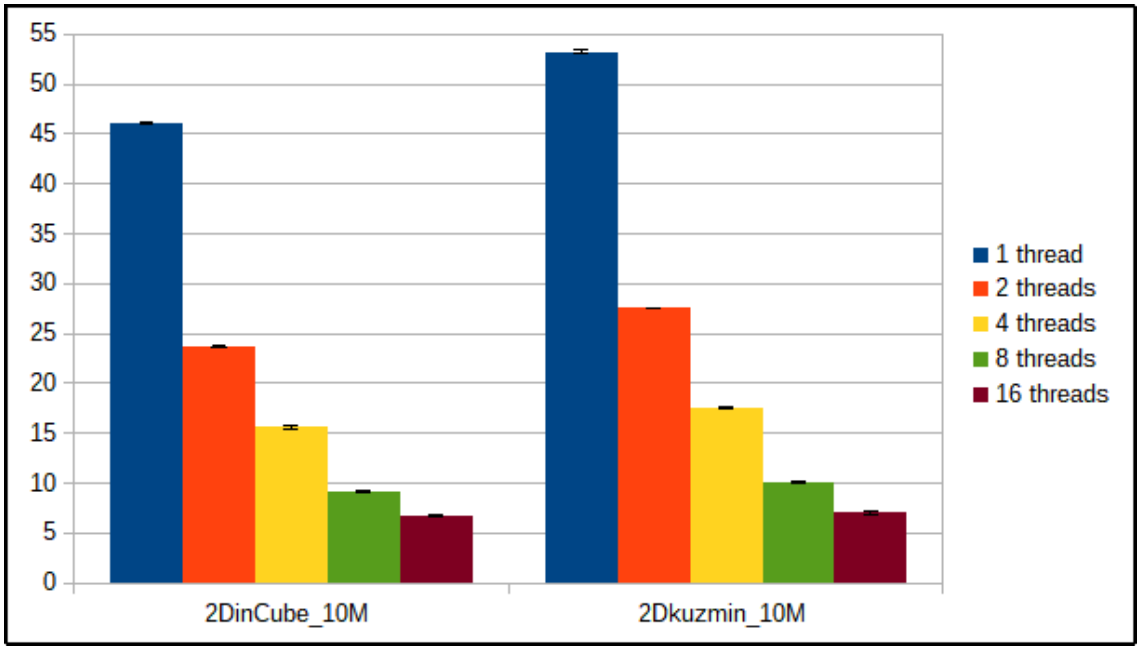


Figure A.313: Execution times (in minutes) of Delaunay Triangulation ran on Intel 12-24 using SBLCWS (**second version**)

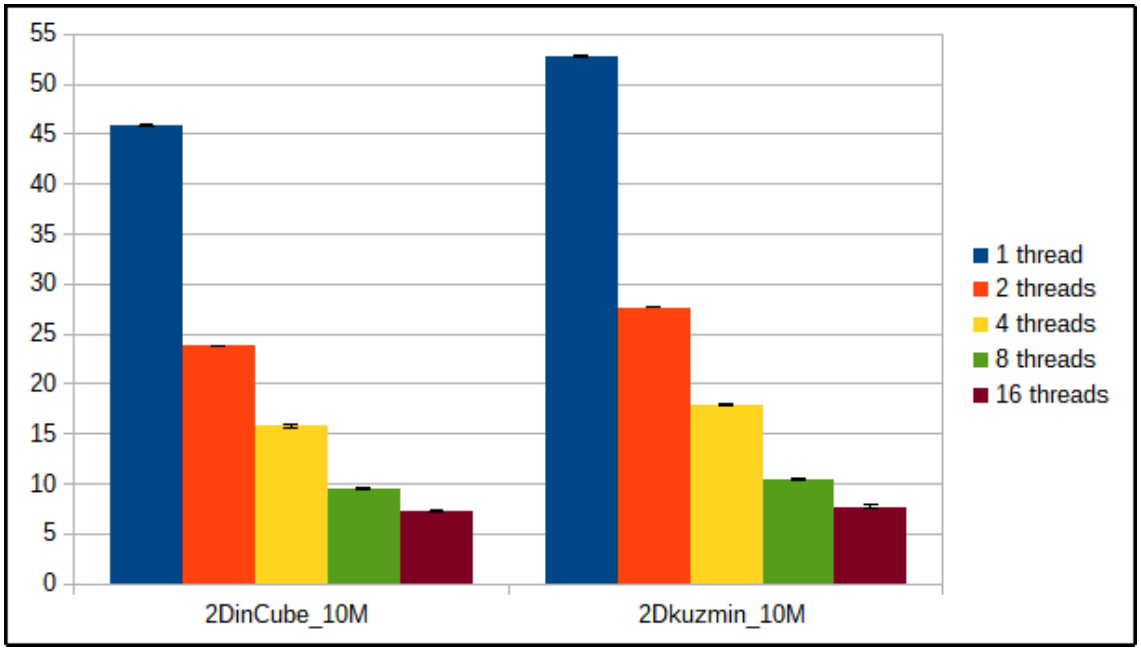


Figure A.314: Execution times (in minutes) of Delaunay Triangulation ran on Intel 12-24 using LCWS with WShare (**tit-for-tat**).

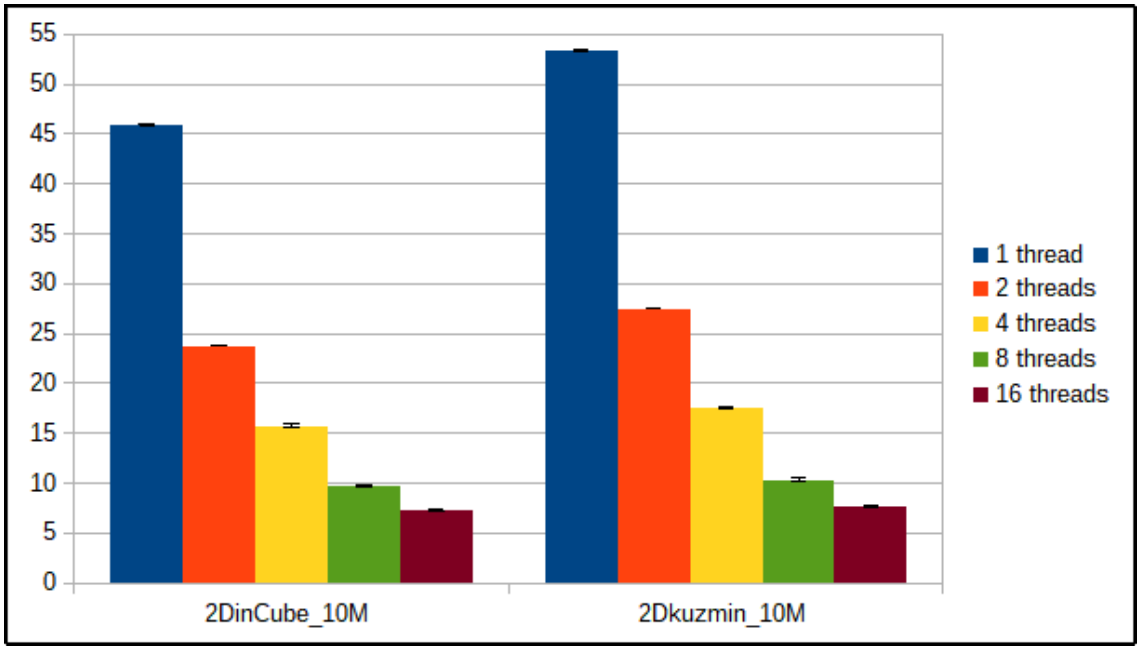


Figure A.315: Execution times (in minutes) of Delaunay Triangulation ran on Intel 12-24 using LCWS with WShare (**continuous tit-for-tat**).

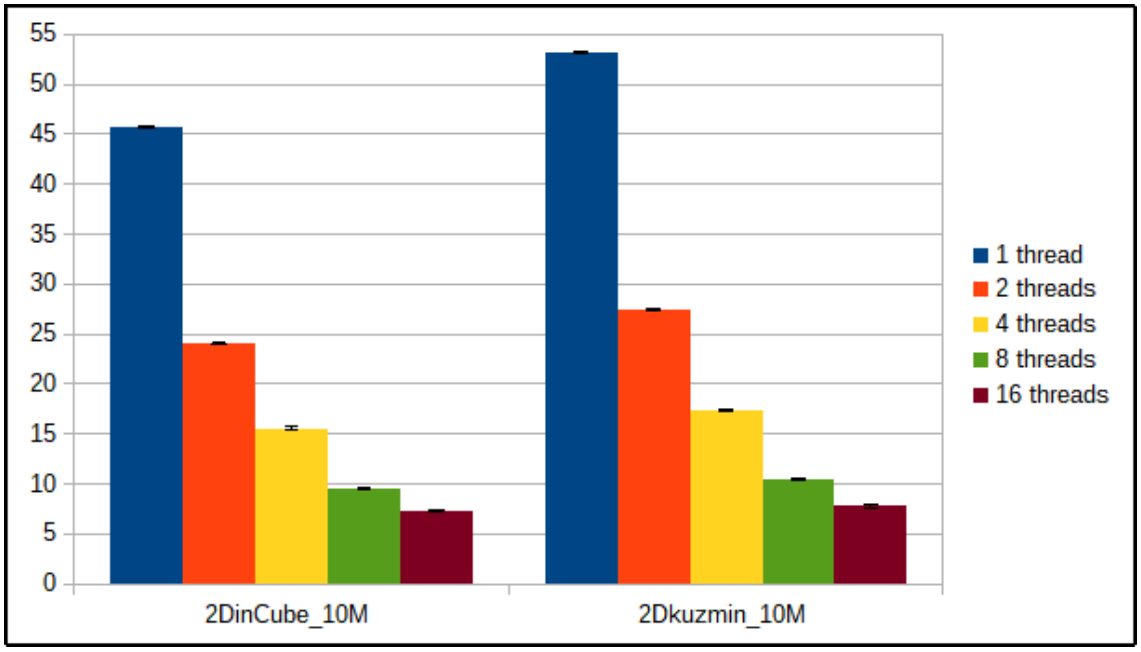


Figure A.316: Execution times (in minutes) of Delaunay Triangulation ran on Intel 12-24 using SBLCWS with WShare (**first version**).

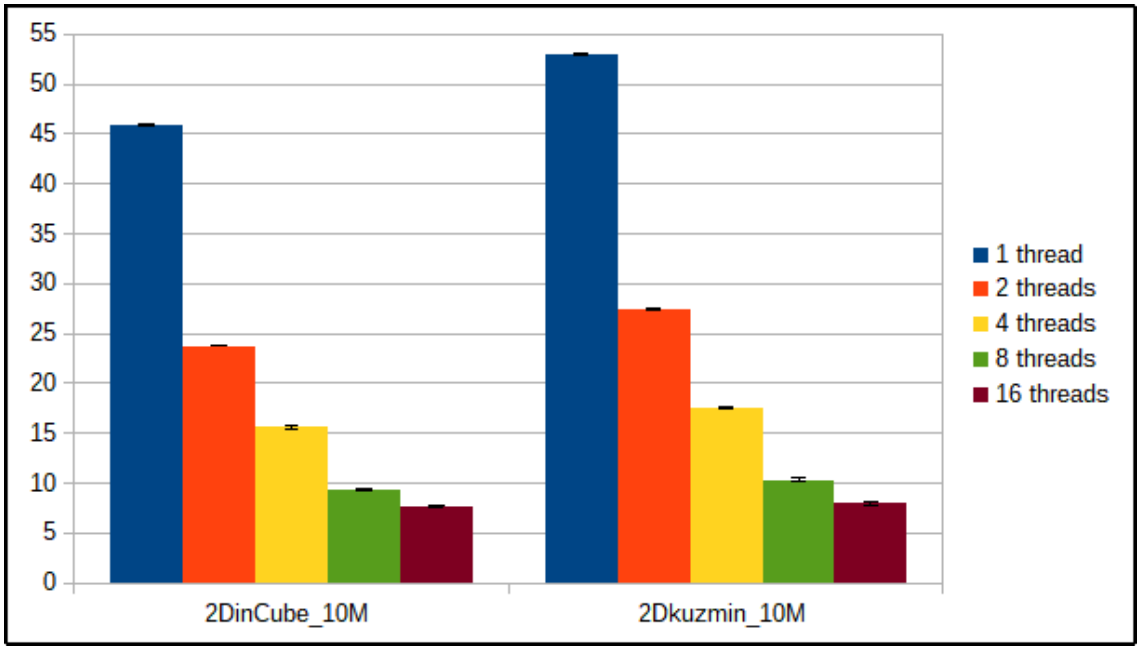


Figure A.317: Execution times (in minutes) of Delaunay Triangulation ran on Intel 12-24 using SBLCWS with WShare (**second version**).

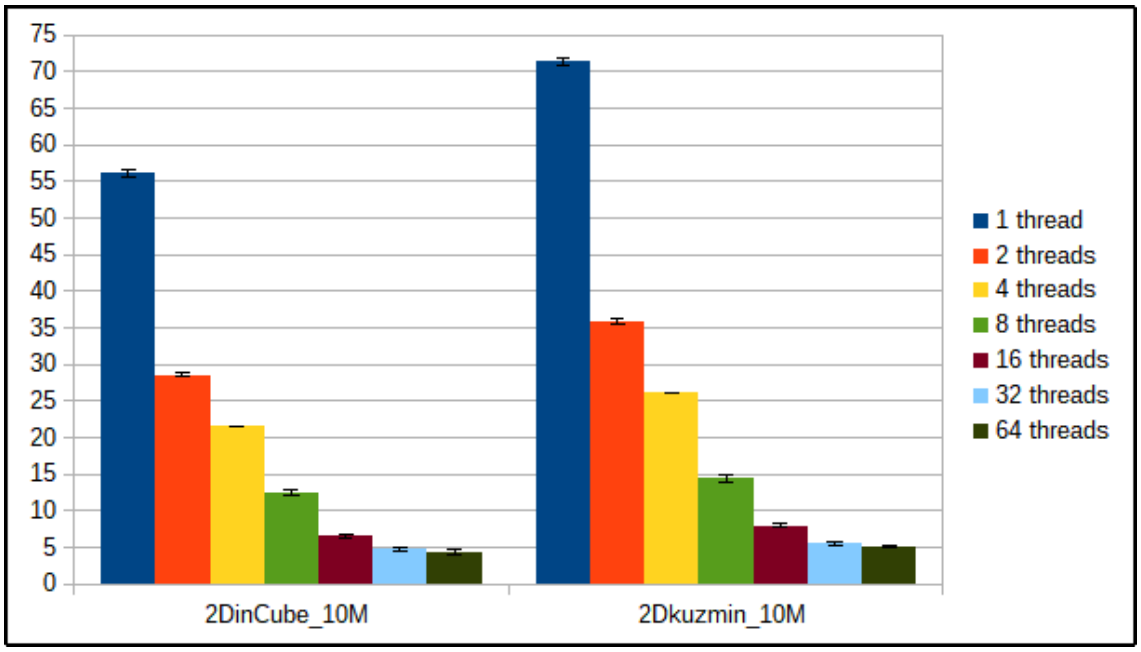


Figure A.318: Execution times (in minutes) of Delaunay Triangulation ran on AMD 32-64 using WSteal

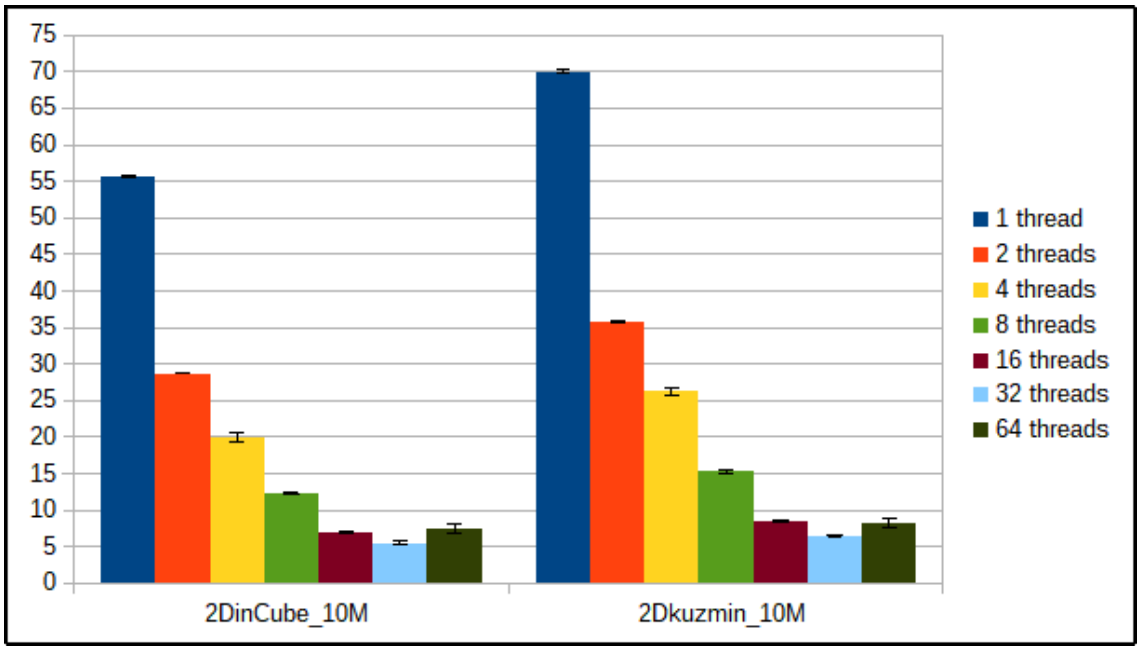


Figure A.319: Execution times (in minutes) of Delaunay Triangulation ran on AMD 32-64 using LCWS

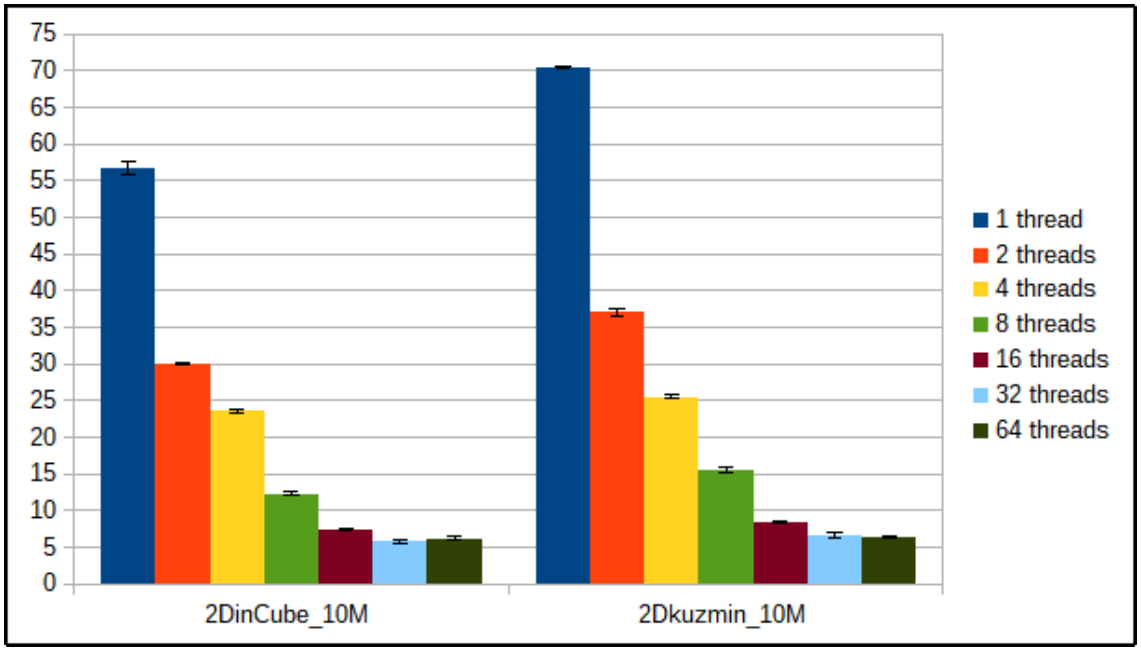


Figure A.320: Execution times (in minutes) of Delaunay Triangulation ran on AMD 32-64 using SBLCWS (**first version**)

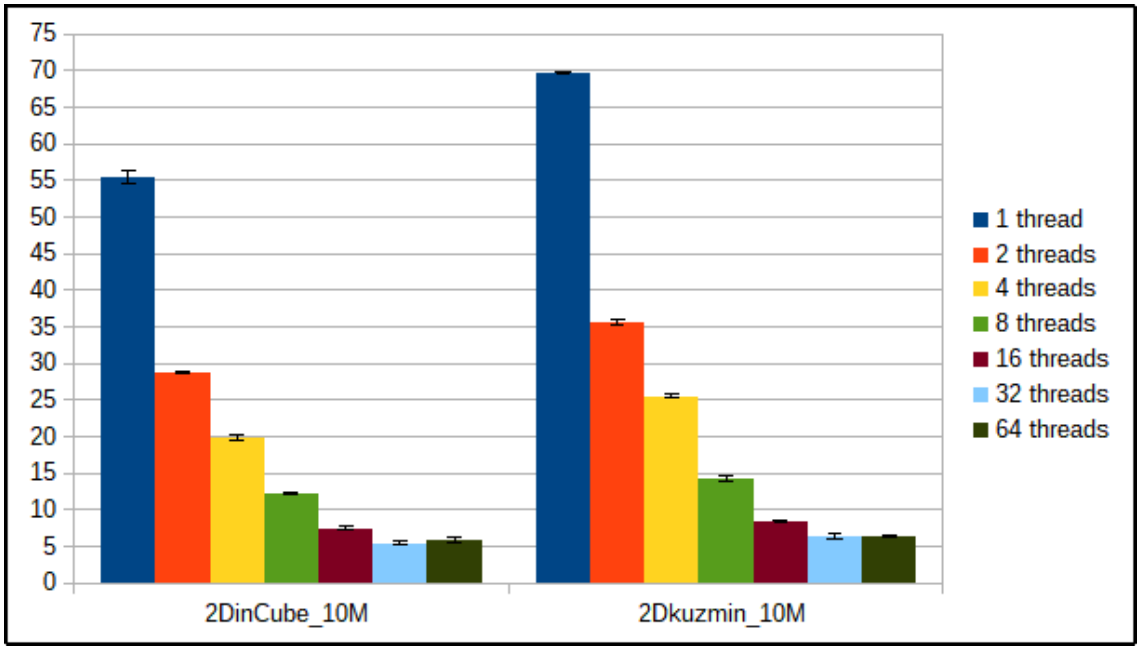


Figure A.321: Execution times (in minutes) of Delaunay Triangulation ran on AMD 32-64 using SBLCWS (**second version**)

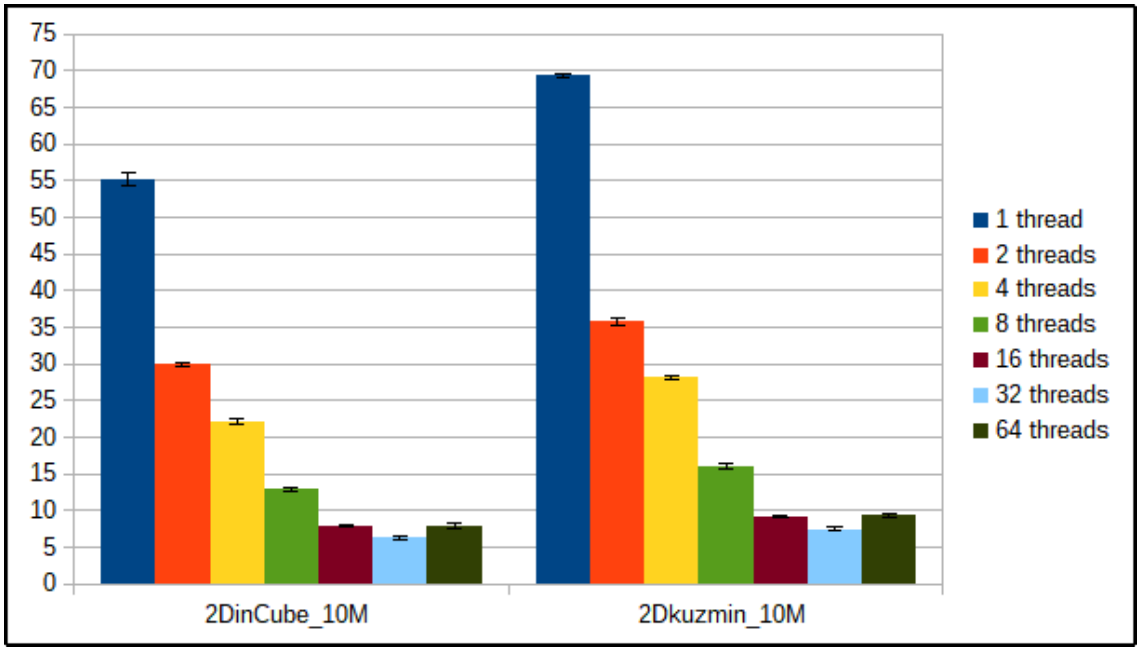


Figure A.322: Execution times (in minutes) of Delaunay Triangulation ran on AMD 32-64 using LCWS with WShare (**tit-for-tat**).

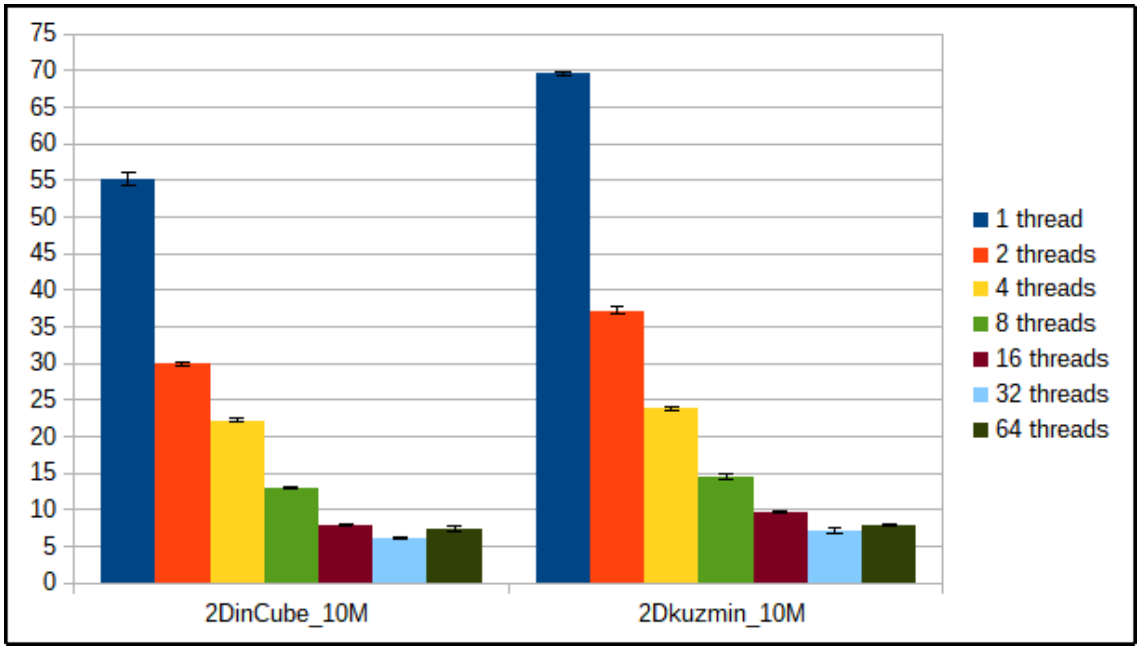


Figure A.323: Execution times (in minutes) of Delaunay Triangulation ran on AMD 32-64 using LCWS with WShare (**continuous tit-for-tat**).

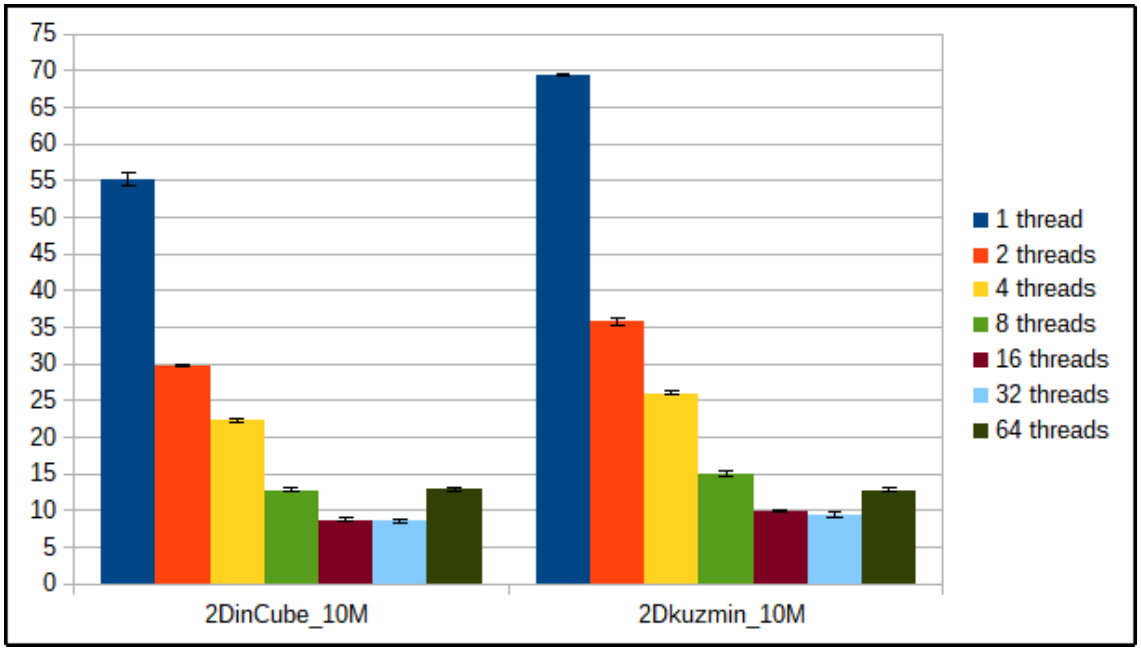


Figure A.324: Execution times (in minutes) of Delaunay Triangulation ran on AMD 32-64 using SBLCWS with WShare (**first version**).

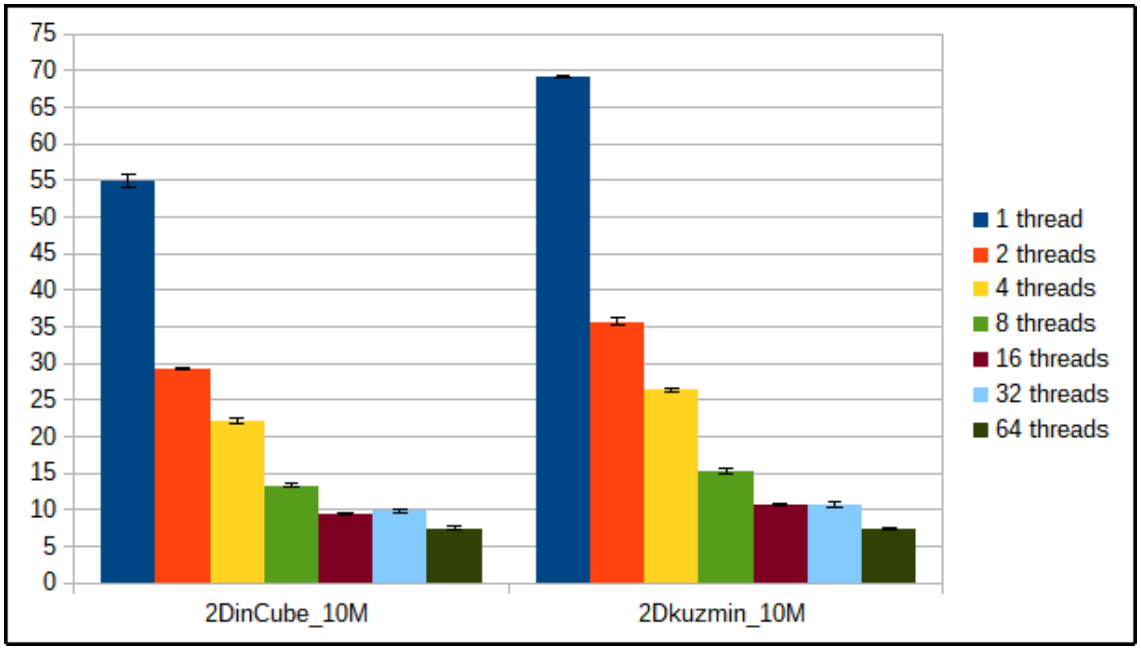


Figure A.325: Execution times (in minutes) of Delaunay Triangulation ran on AMD 32-64 using SBLCWS with WShare (**second version**).

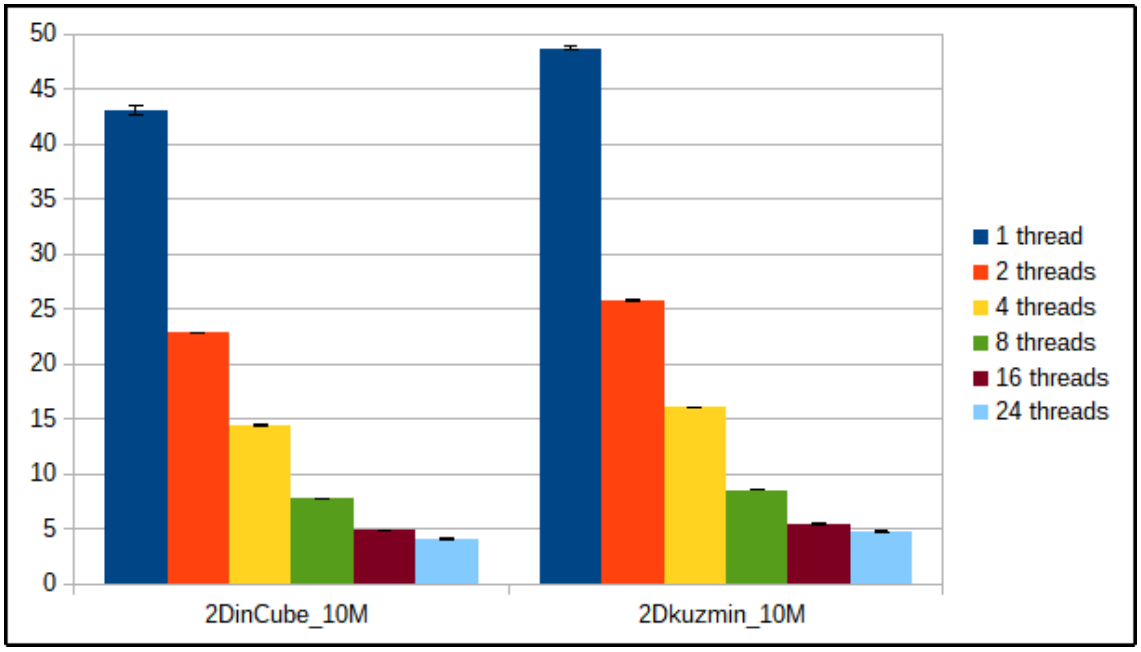


Figure A.326: Execution times (in minutes) of Delaunay Triangulation ran on Intel 16-16 using WSteal

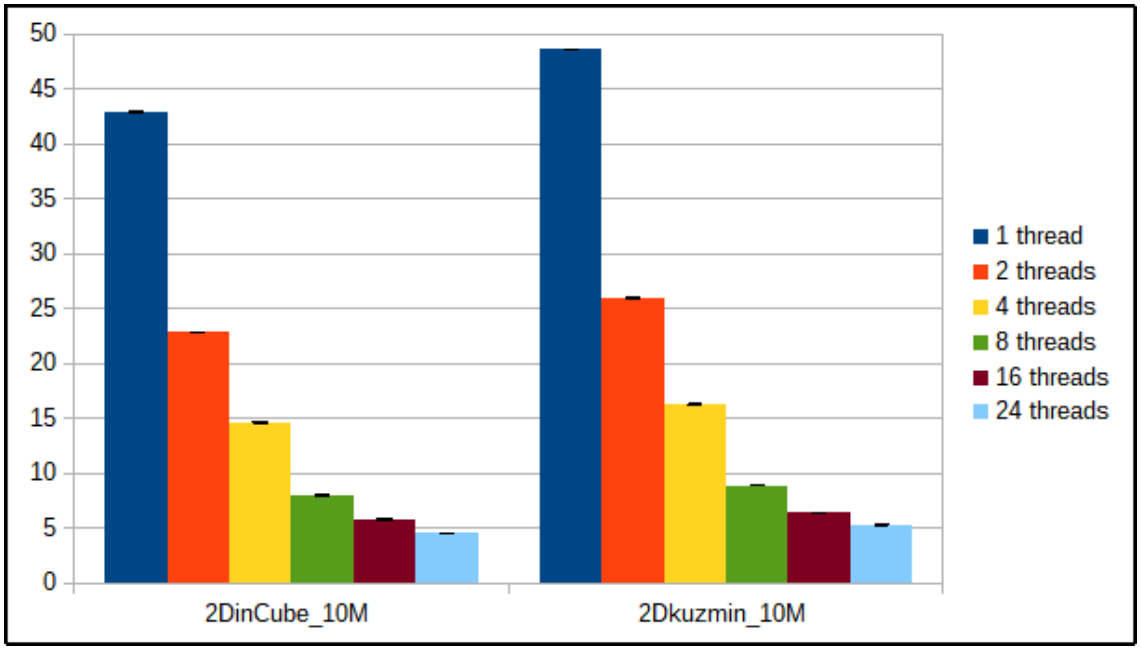


Figure A.327: Execution times (in minutes) of Delaunay Triangulation ran on Intel 16-16 using LCWS

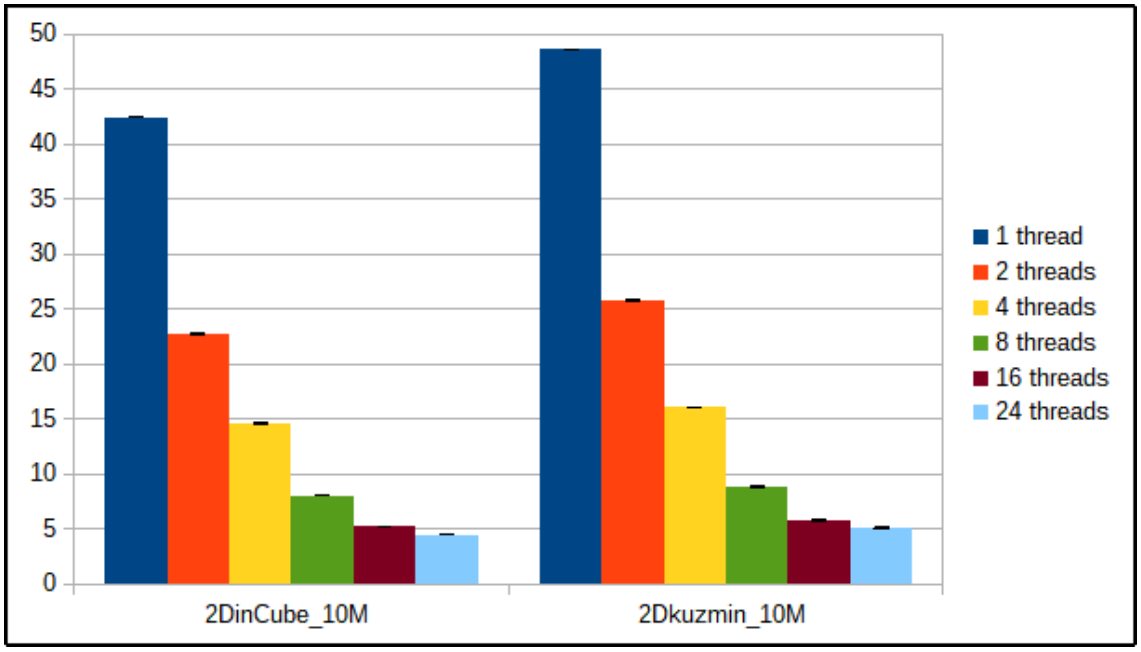


Figure A.328: Execution times (in minutes) of Delaunay Triangulation ran on Intel 16-16 using SBLCWS (**first version**)

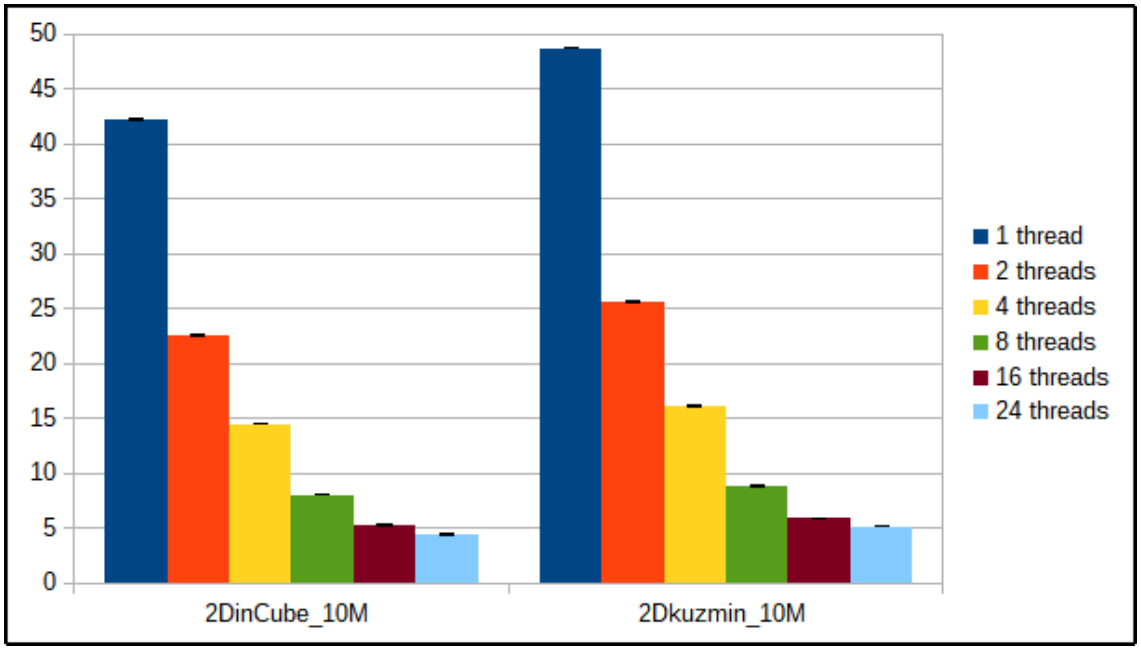


Figure A.329: Execution times (in minutes) of Delaunay Triangulation ran on Intel 16-16 using SBLCWS (**second version**)

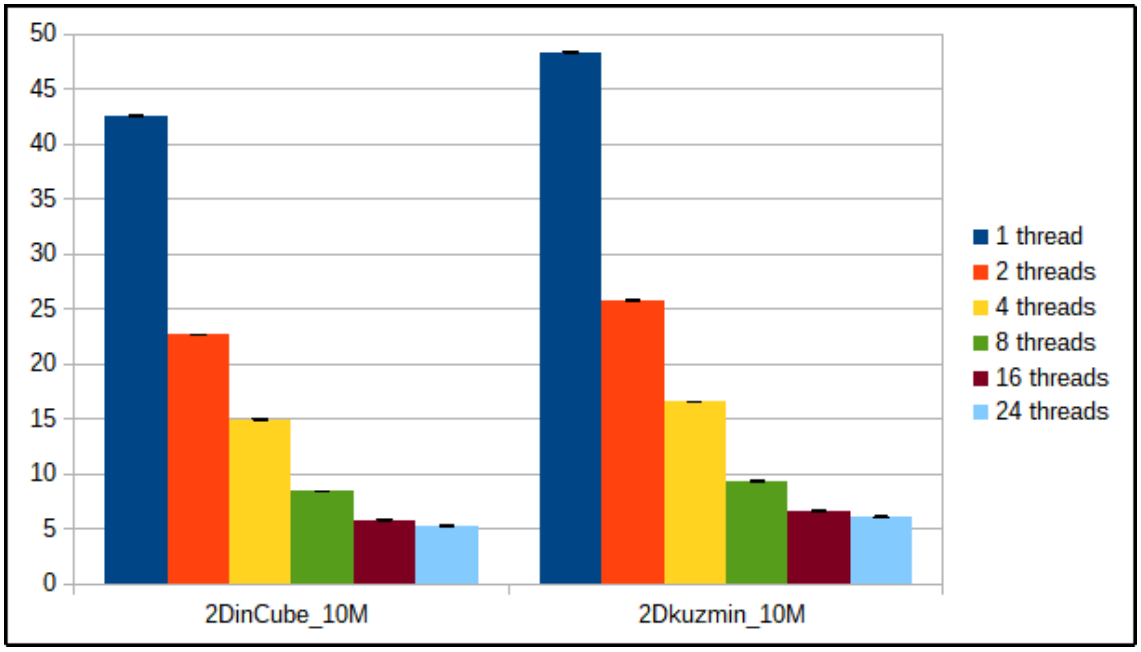


Figure A.330: Execution times (in minutes) of Delaunay Triangulation ran on Intel 16-16 using LCWS with WShare (**tit-for-tat**).

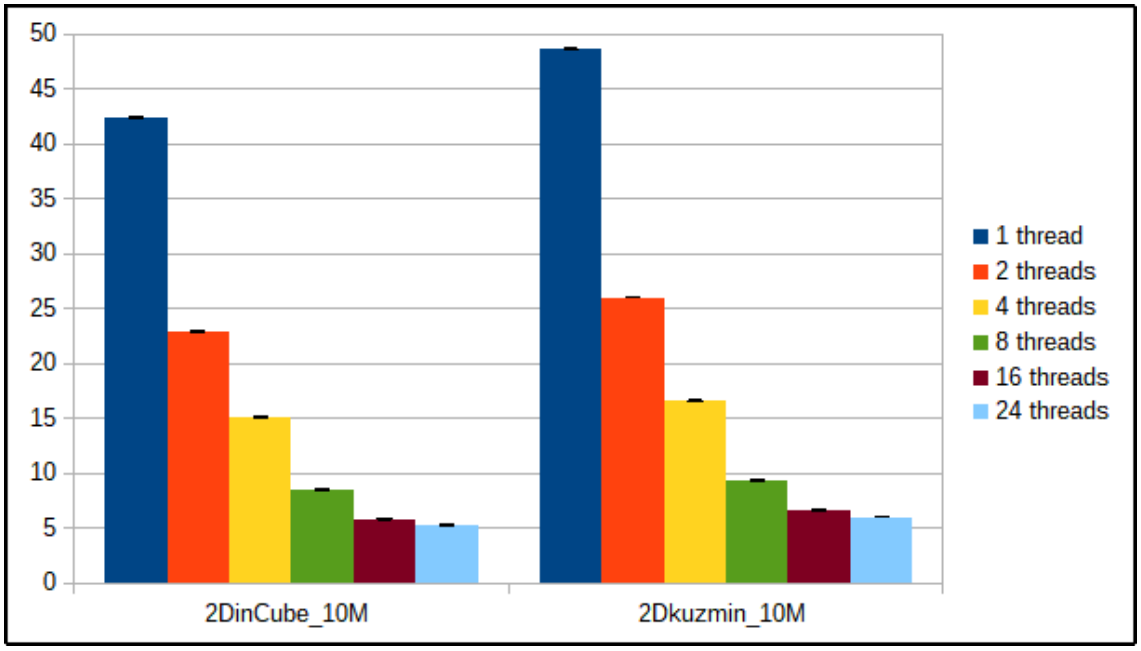


Figure A.331: Execution times (in minutes) of Delaunay Triangulation ran on Intel 16-16 using LCWS with WShare (**continuous tit-for-tat**).

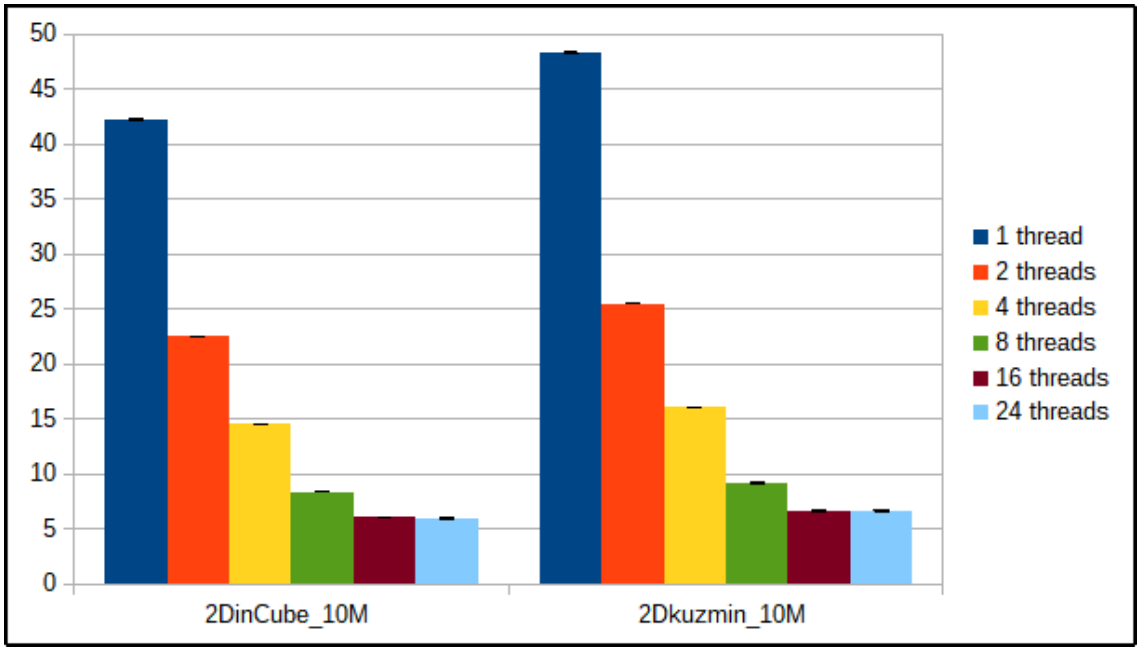


Figure A.332: Execution times (in minutes) of Delaunay Triangulation ran on Intel 16-16 using SBLCWS with WShare (**first version**).

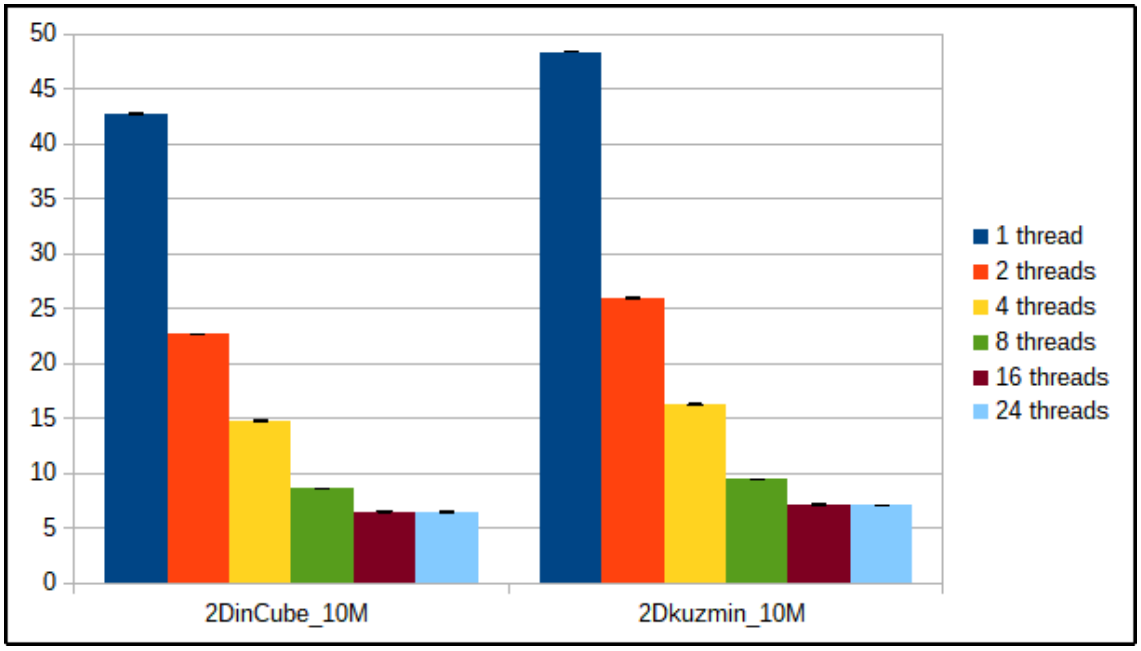


Figure A.333: Execution times (in minutes) of Delaunay Triangulation ran on Intel 16-16 using SBLCWS with WShare (**second version**).

A.15 Delaunay Refine

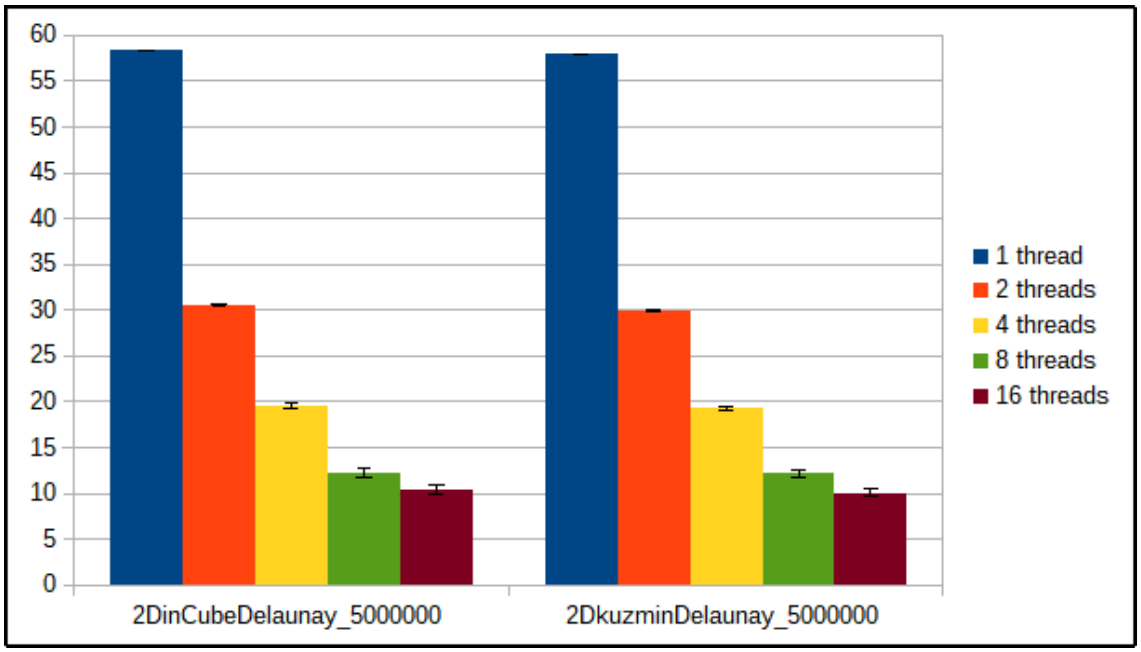


Figure A.334: Execution times (in minutes) of Delaunay Refine ran on Intel 12-24 using WSteal

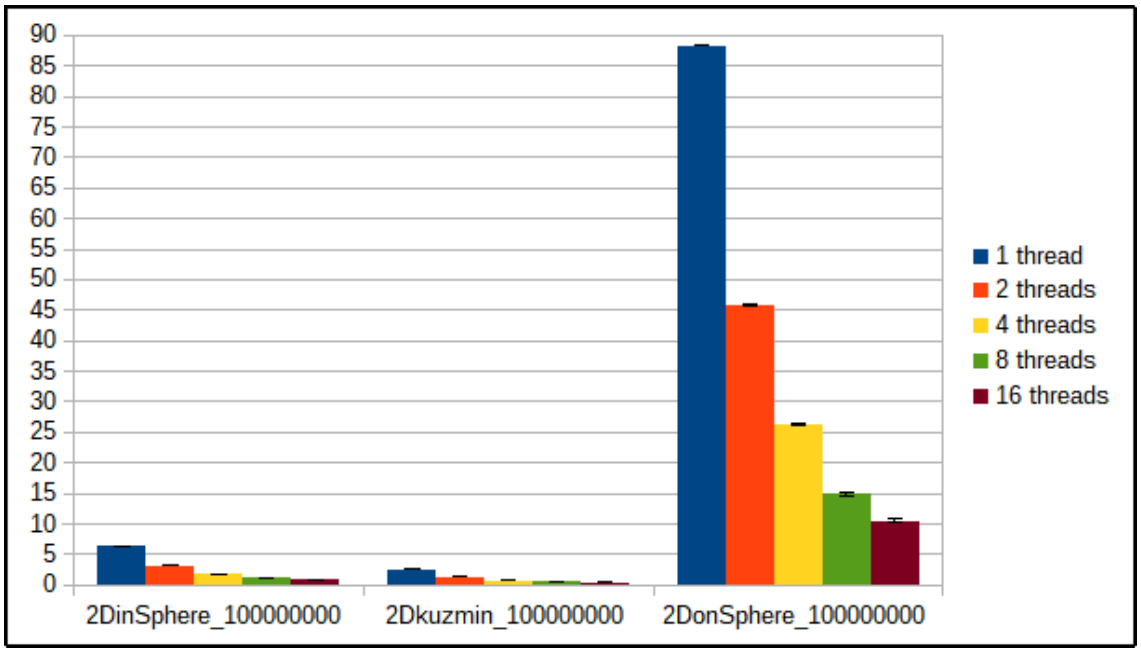


Figure A.335: Execution times (in minutes) of Delaunay Refine ran on Intel 12-24 using LCWS

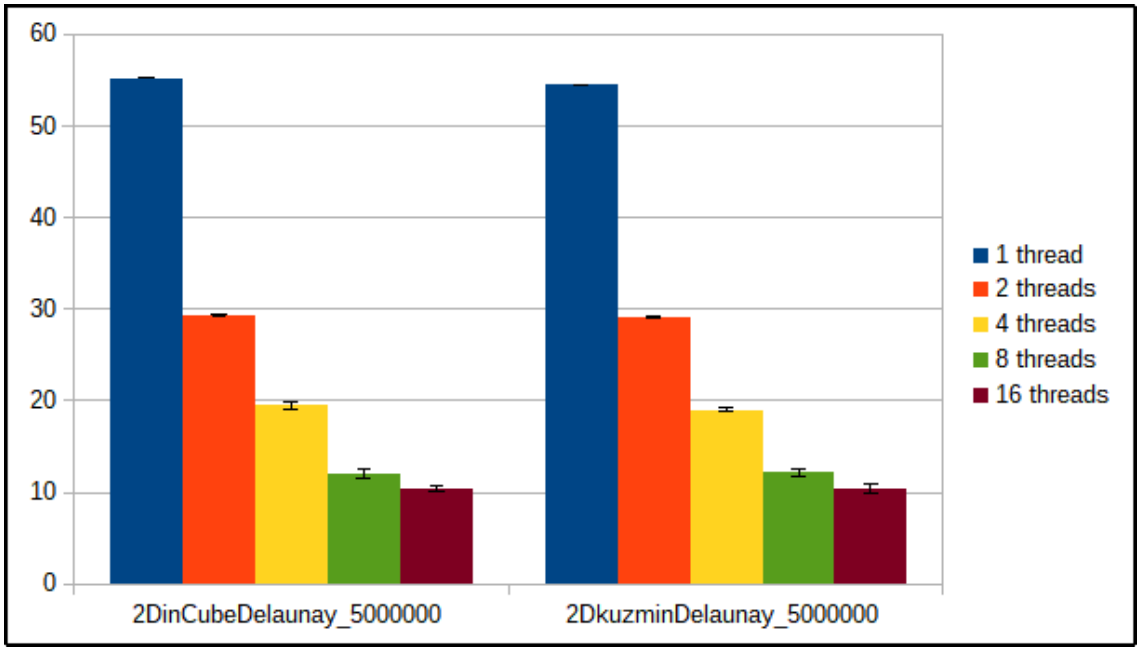


Figure A.336: Execution times (in minutes) of Delaunay Refine ran on Intel 12-24 using SBLCWS (**first version**)

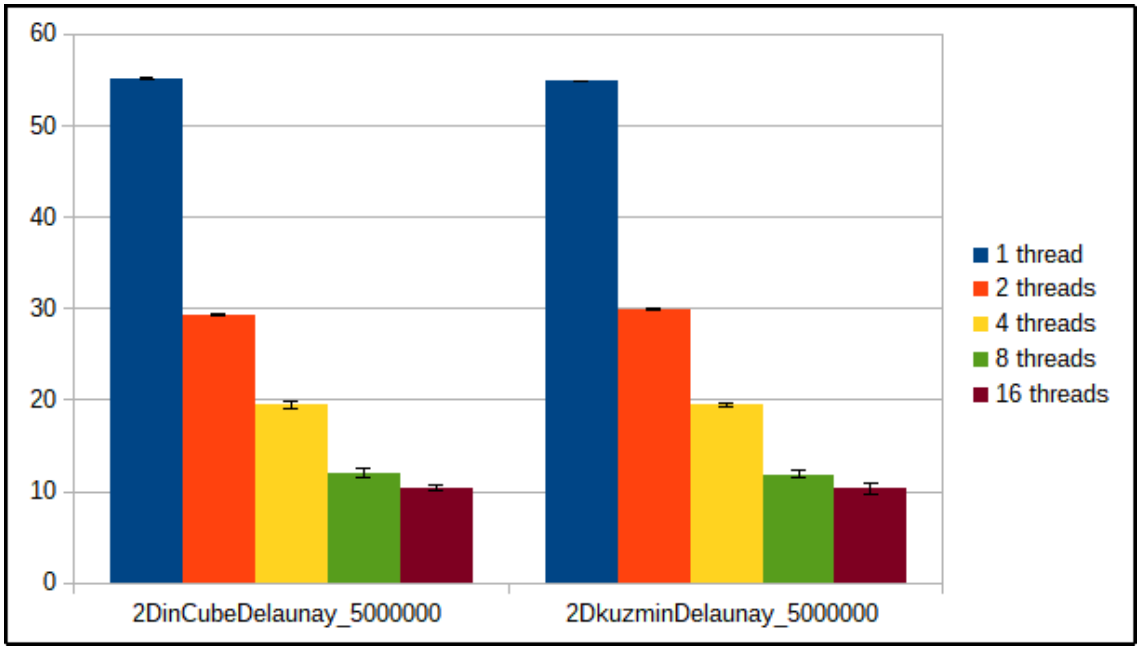


Figure A.337: Execution times (in minutes) of Delaunay Refine ran on Intel 12-24 using SBLCWS (**second version**)

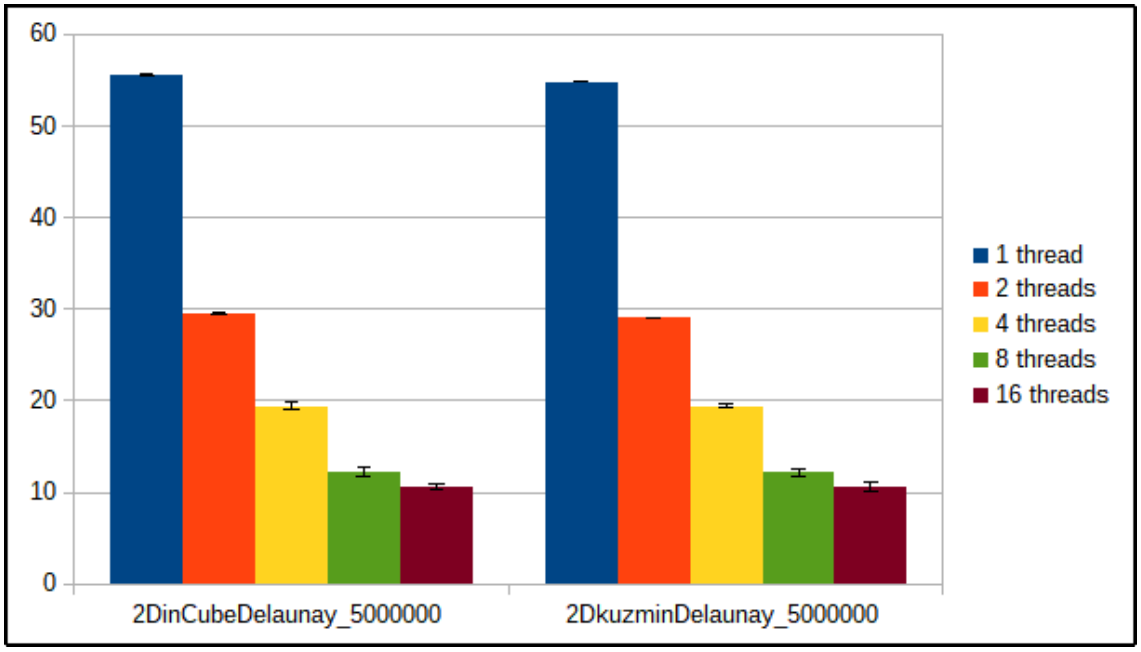


Figure A.338: Execution times (in minutes) of Delaunay Refine ran on Intel 12-24 using LCWS with WShare (**tit-for-tat**).

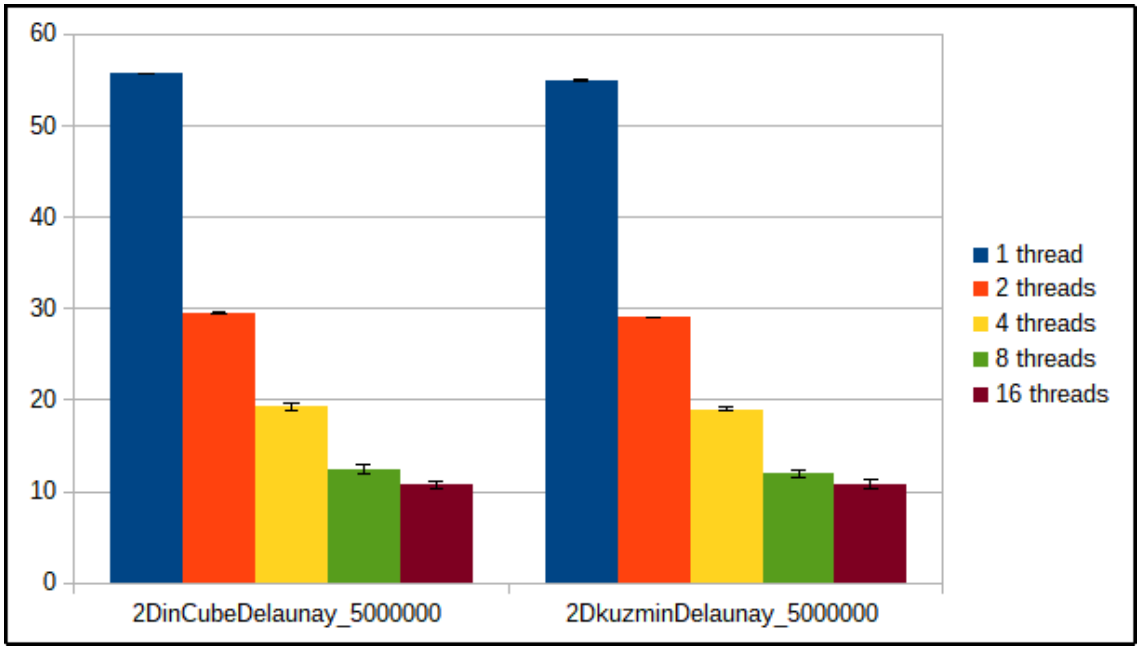


Figure A.339: Execution times (in minutes) of Delaunay Refine ran on Intel 12-24 using LCWS with WShare (**continuous tit-for-tat**).

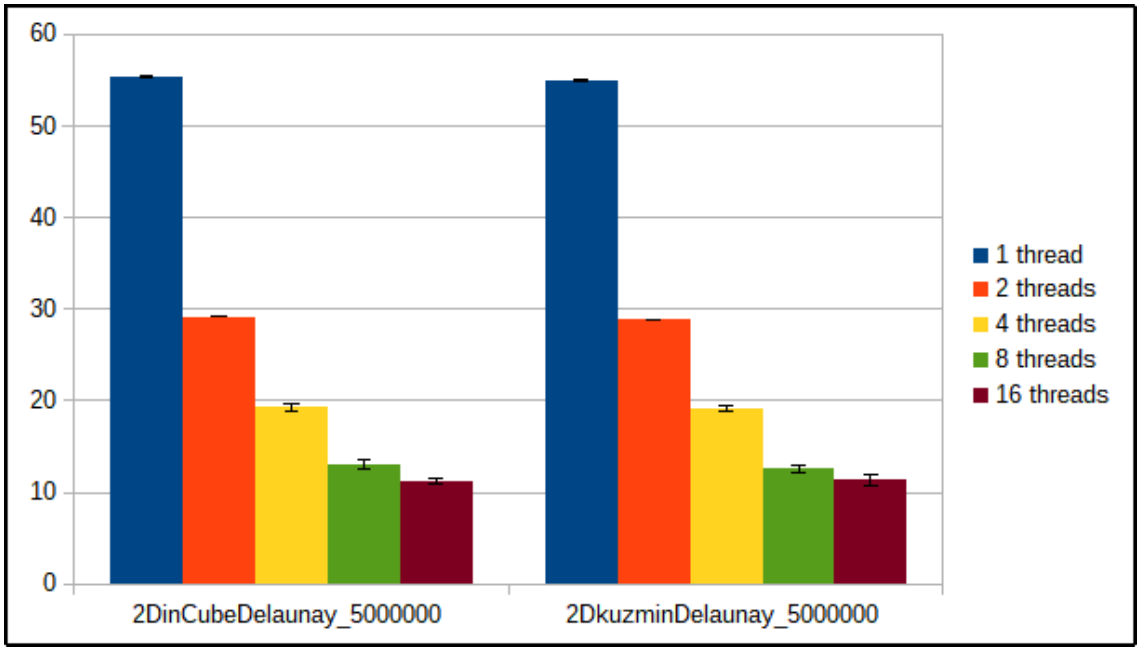


Figure A.340: Execution times (in minutes) of Delaunay Refine ran on Intel 12-24 using SBLCWS with WShare (**first version**).

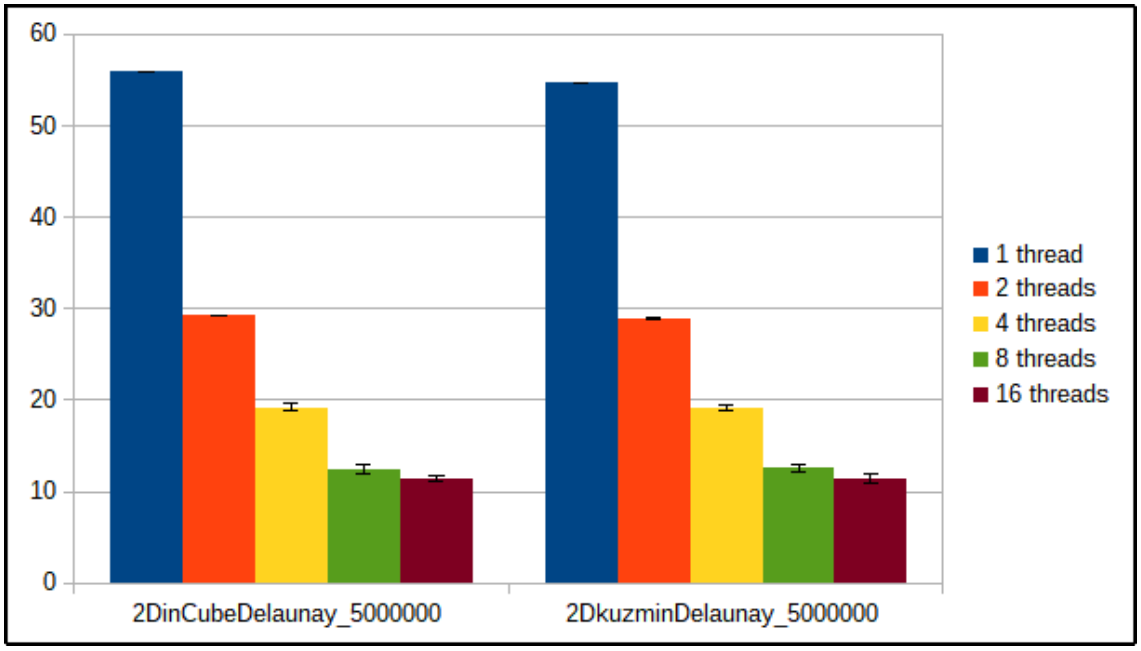


Figure A.341: Execution times (in minutes) of Delaunay Refine ran on Intel 12-24 using SBLCWS with WShare (**second version**).

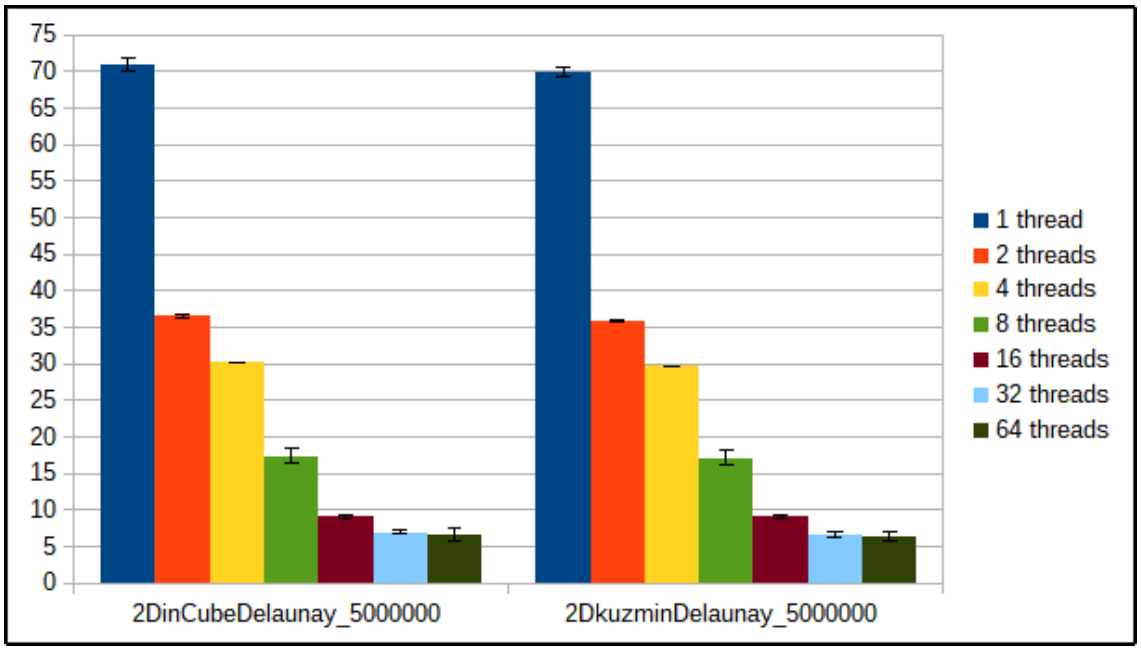


Figure A.342: Execution times (in minutes) of Delaunay Refine ran on AMD 32-64 using WSteal

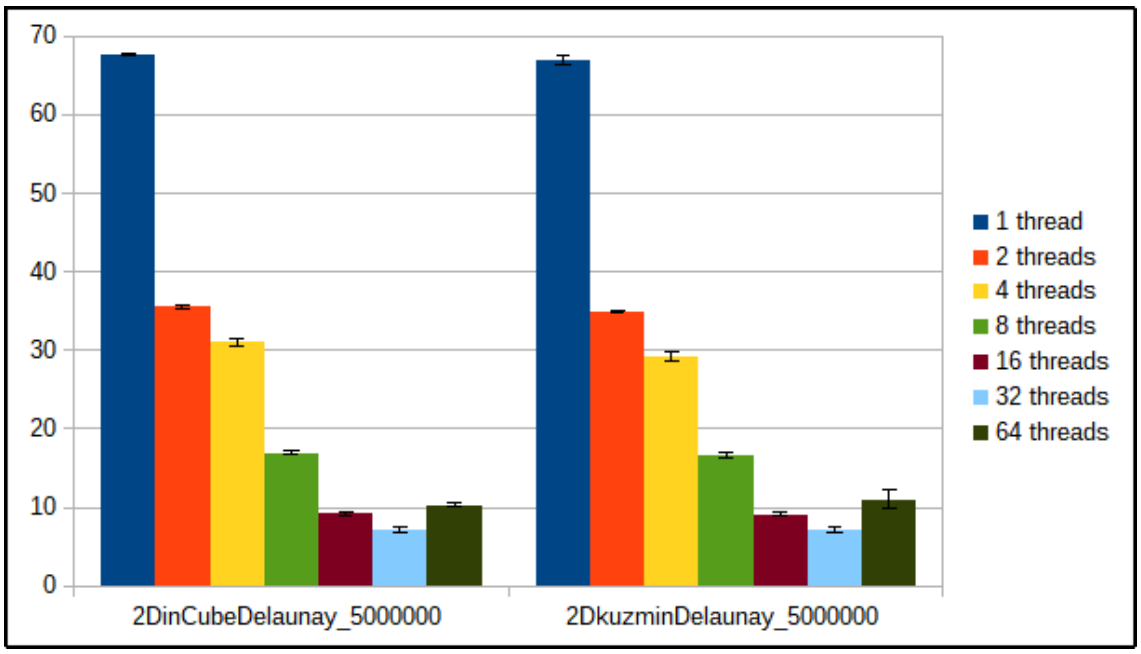


Figure A.343: Execution times (in minutes) of Delaunay Refine ran on AMD 32-64 using LCWS

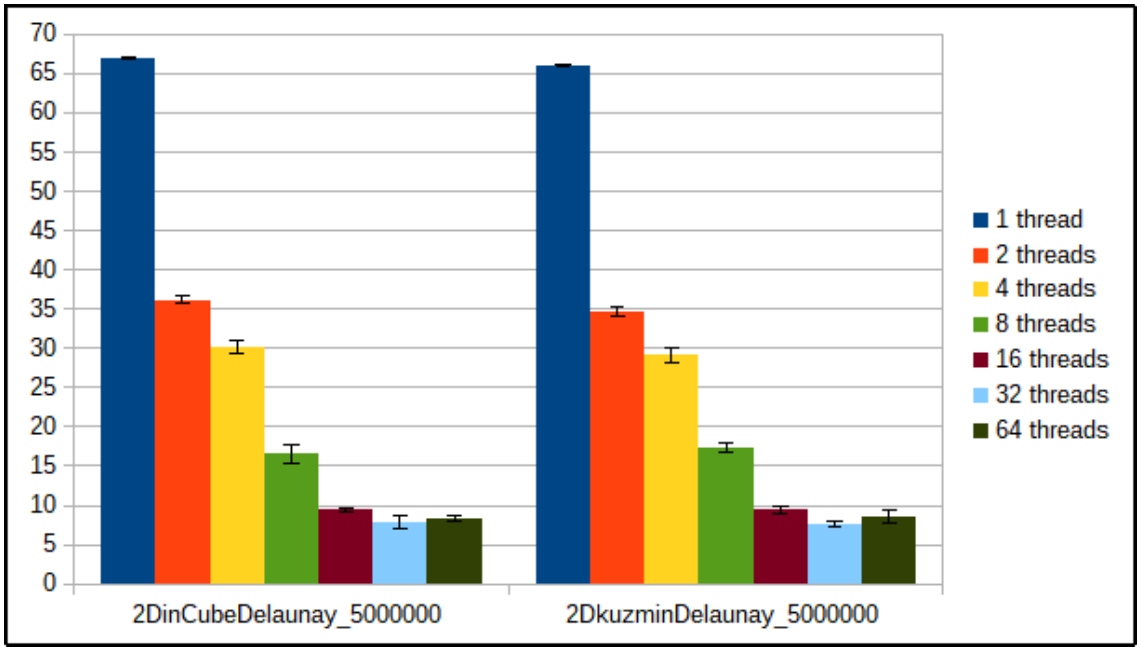


Figure A.344: Execution times (in minutes) of Delaunay Refine ran on AMD 32-64 using SBLCWS (**first version**)

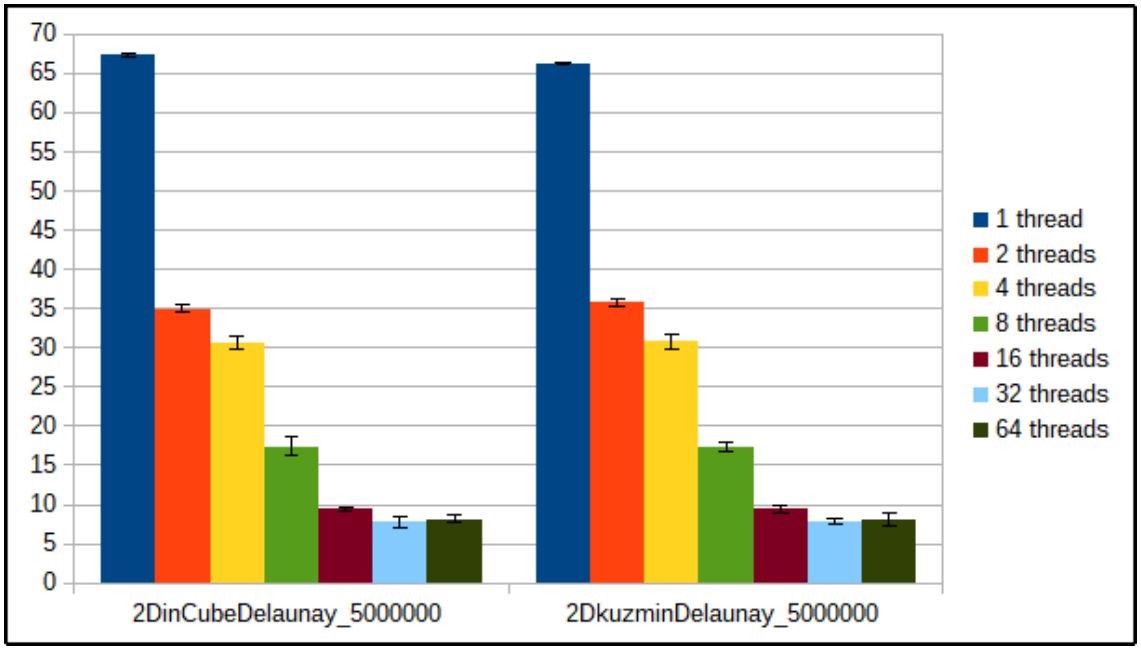


Figure A.345: Execution times (in minutes) of Delaunay Refine ran on AMD 32-64 using SBLCWS (**second version**)

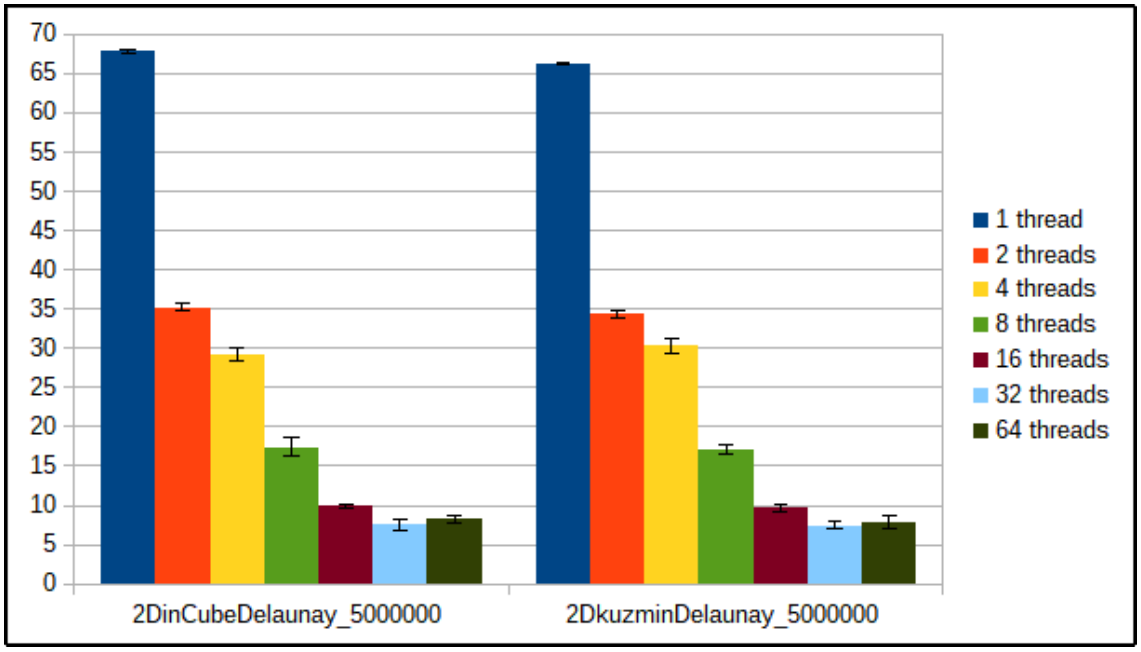


Figure A.346: Execution times (in minutes) of Delaunay Refine ran on AMD 32-64 using LCWS with WShare (**tit-for-tat**).

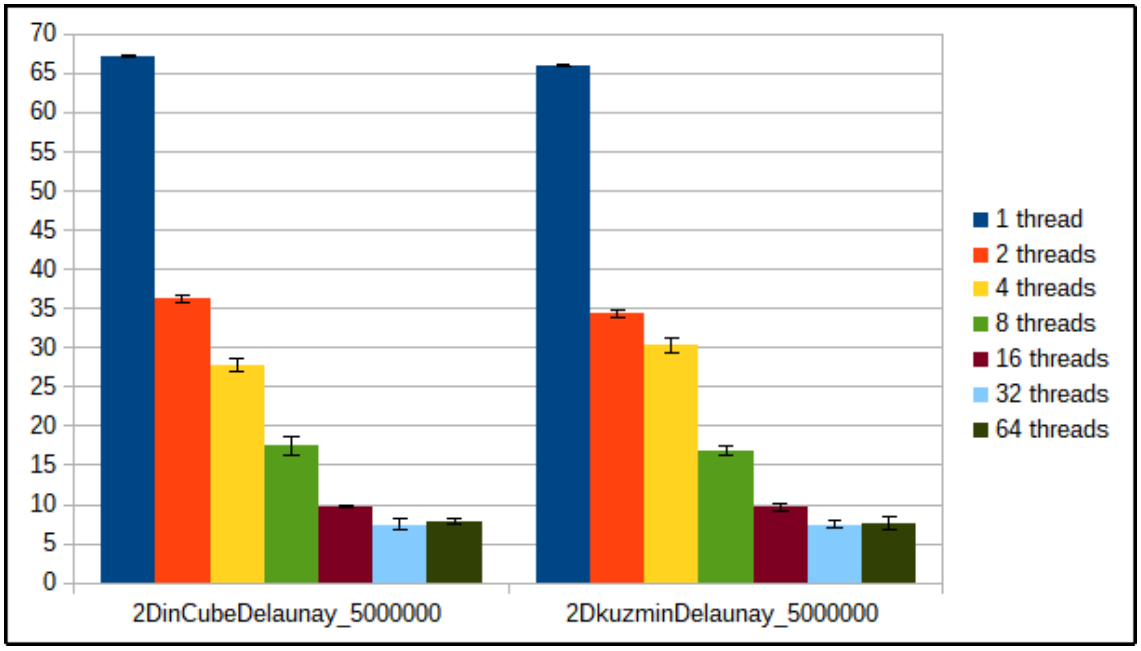


Figure A.347: Execution times (in minutes) of Delaunay Refine ran on AMD 32-64 using LCWS with WShare (**continuous tit-for-tat**).

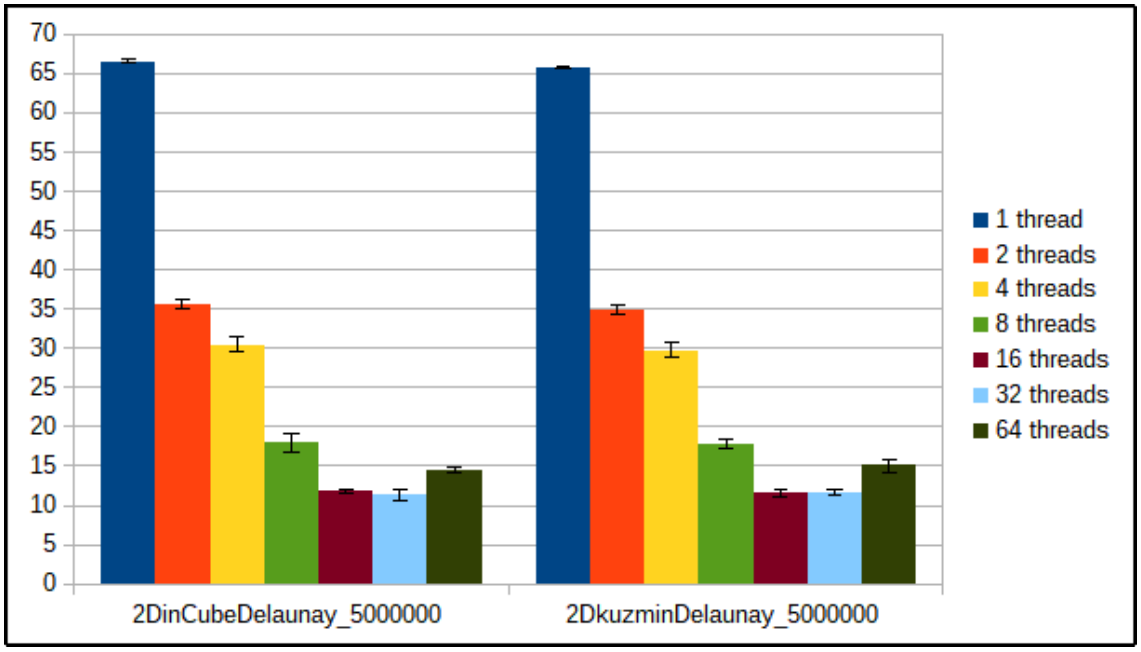


Figure A.348: Execution times (in minutes) of Delaunay Refine ran on AMD 32-64 using SBLCWS with WShare (**first version**).

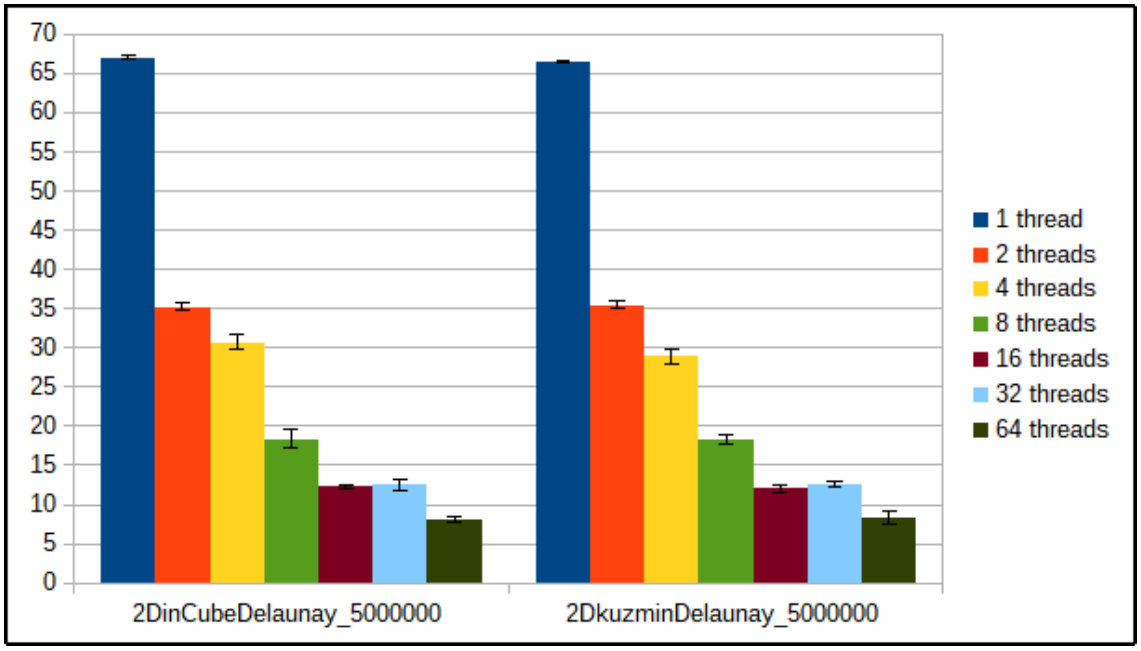


Figure A.349: Execution times (in minutes) of Delaunay Refine ran on AMD 32-64 using SBLCWS with WShare (**second version**).

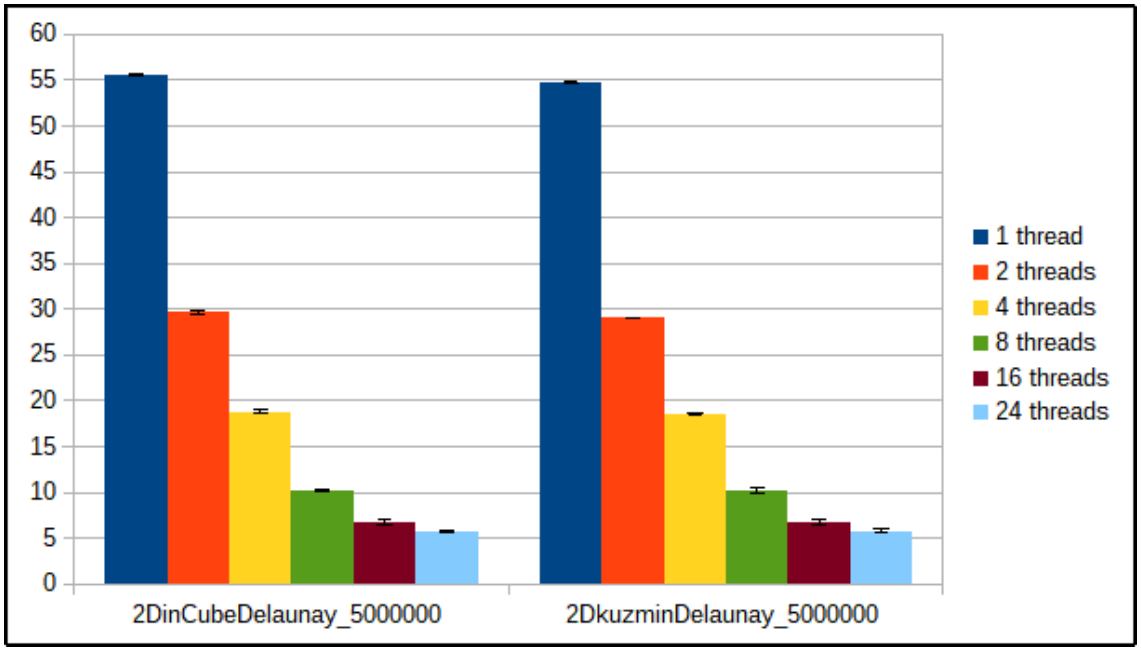


Figure A.350: Execution times (in minutes) of Delaunay Refine ran on Intel 16-16 using WSteal

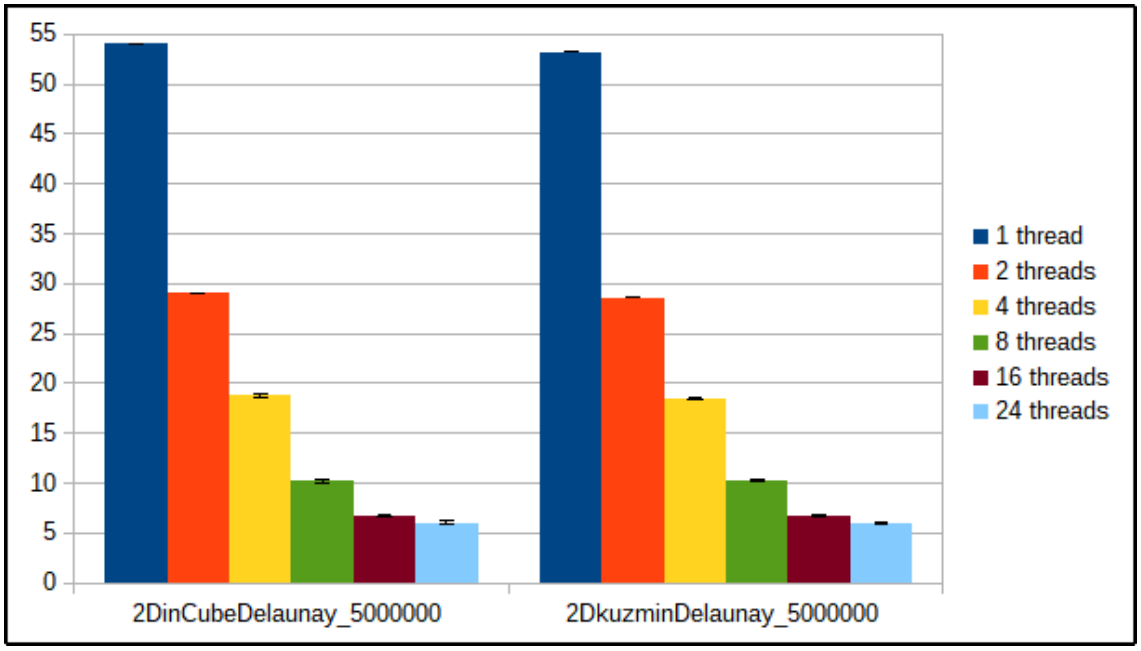


Figure A.351: Execution times (in minutes) of Delaunay Refine ran on Intel 16-16 using LCWS

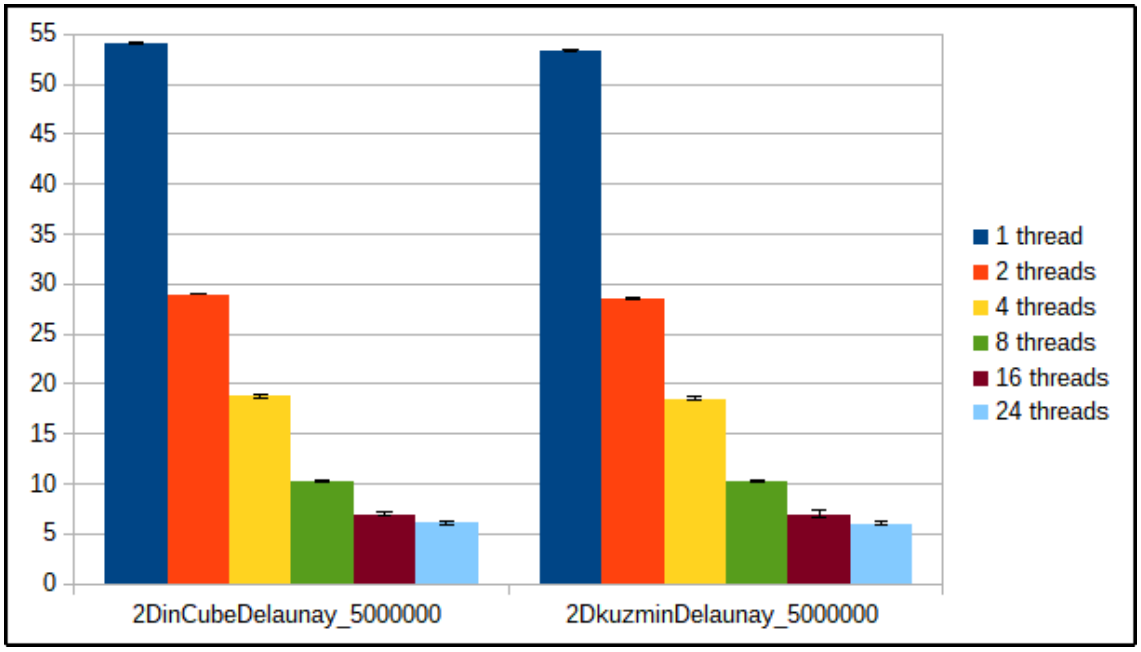


Figure A.352: Execution times (in minutes) of Delaunay Refine ran on Intel 16-16 using SBLCWS (**first version**)

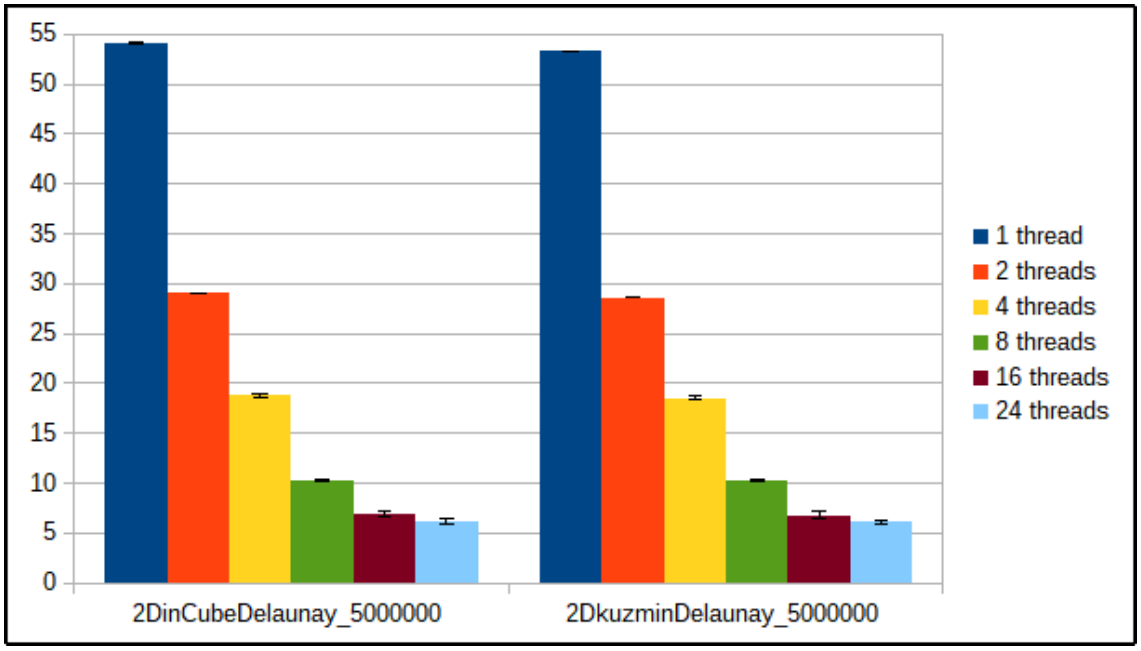


Figure A.353: Execution times (in minutes) of Delaunay Refine ran on Intel 16-16 using SBLCWS (**second version**)

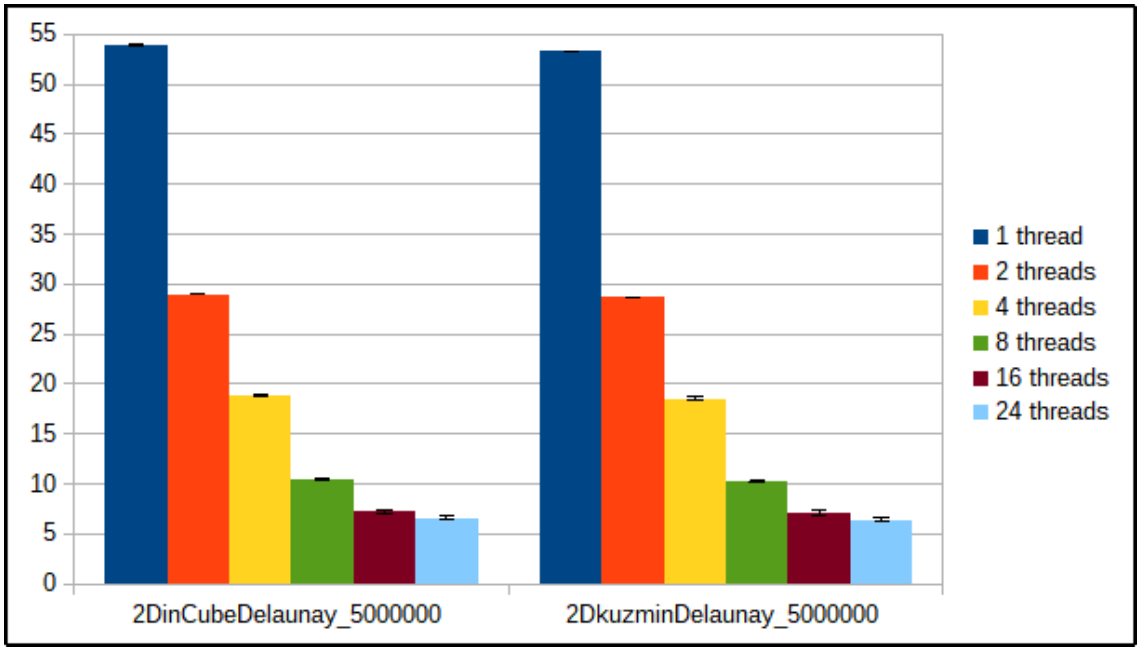


Figure A.354: Execution times (in minutes) of Delaunay Refine ran on Intel 16-16 using LCWS with WShare (**tit-for-tat**).

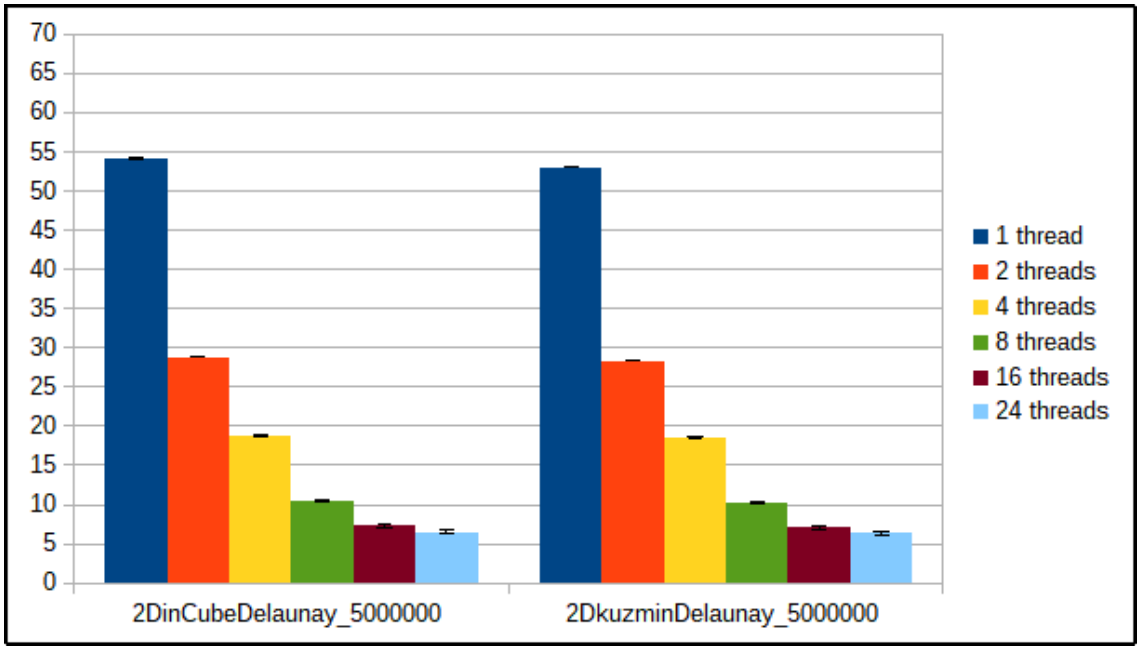


Figure A.355: Execution times (in minutes) of Delaunay Refine ran on Intel 16-16 using LCWS with WShare (**continuous tit-for-tat**).

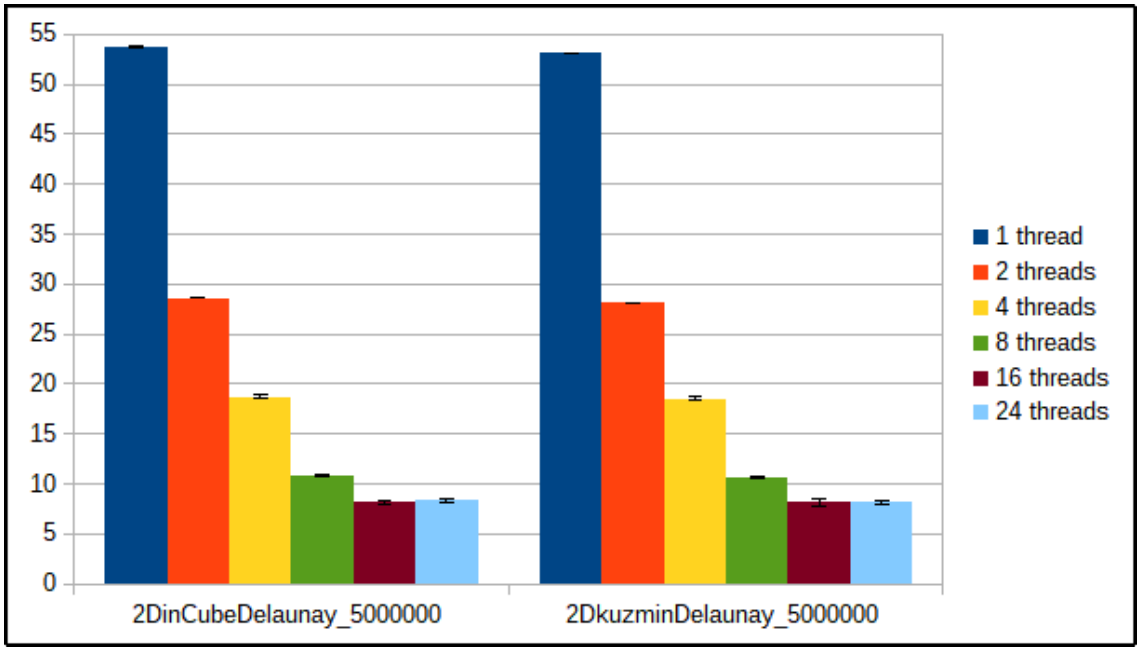


Figure A.356: Execution times (in minutes) of Delaunay Refine ran on Intel 16-16 using SBLCWS with WShare (**first version**).

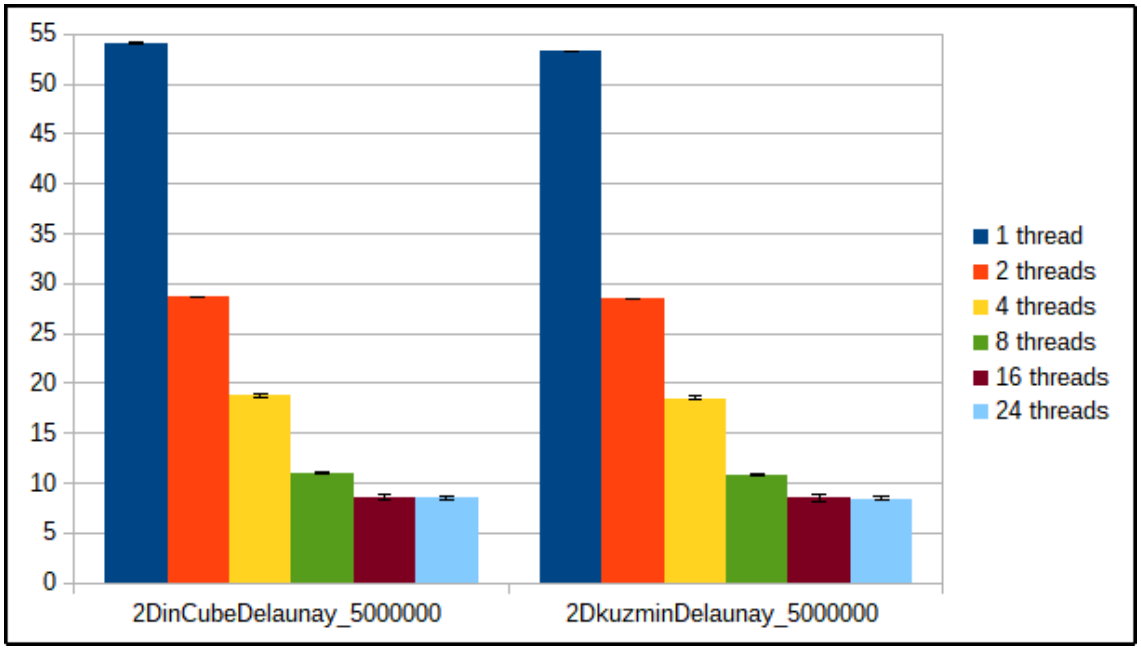


Figure A.357: Execution times (in minutes) of Delaunay Refine ran on Intel 16-16 using SBLCWS with WShare (**second version**).

A.16 Range Query 2D

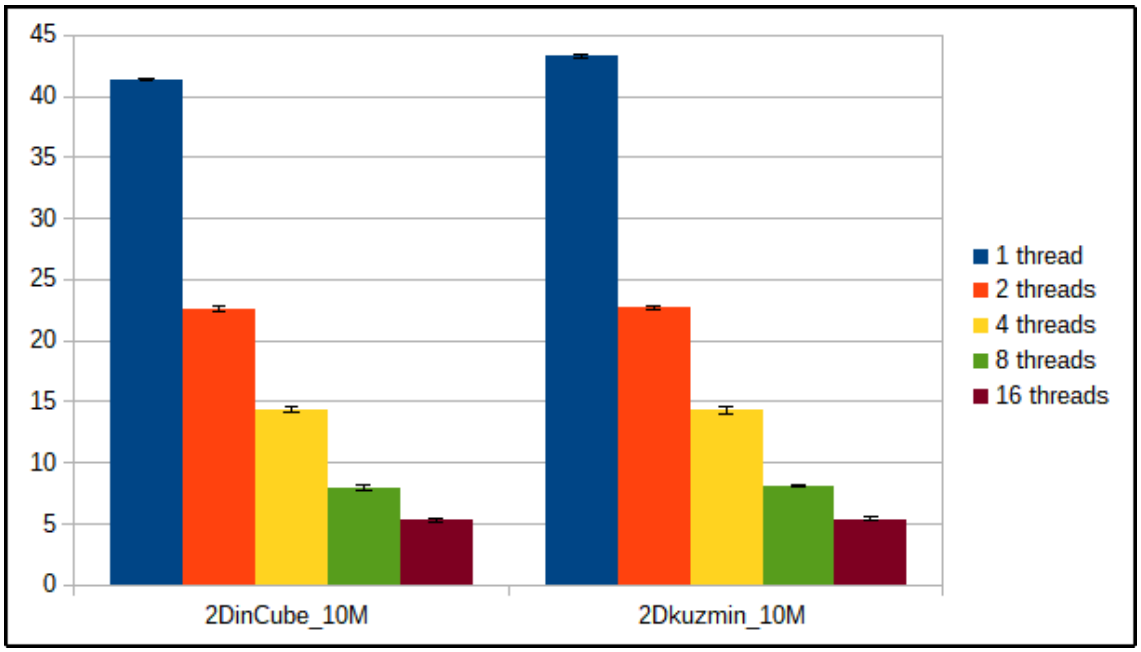


Figure A.358: Execution times (in minutes) of Range Query 2D ran on Intel 12-24 using WSteal

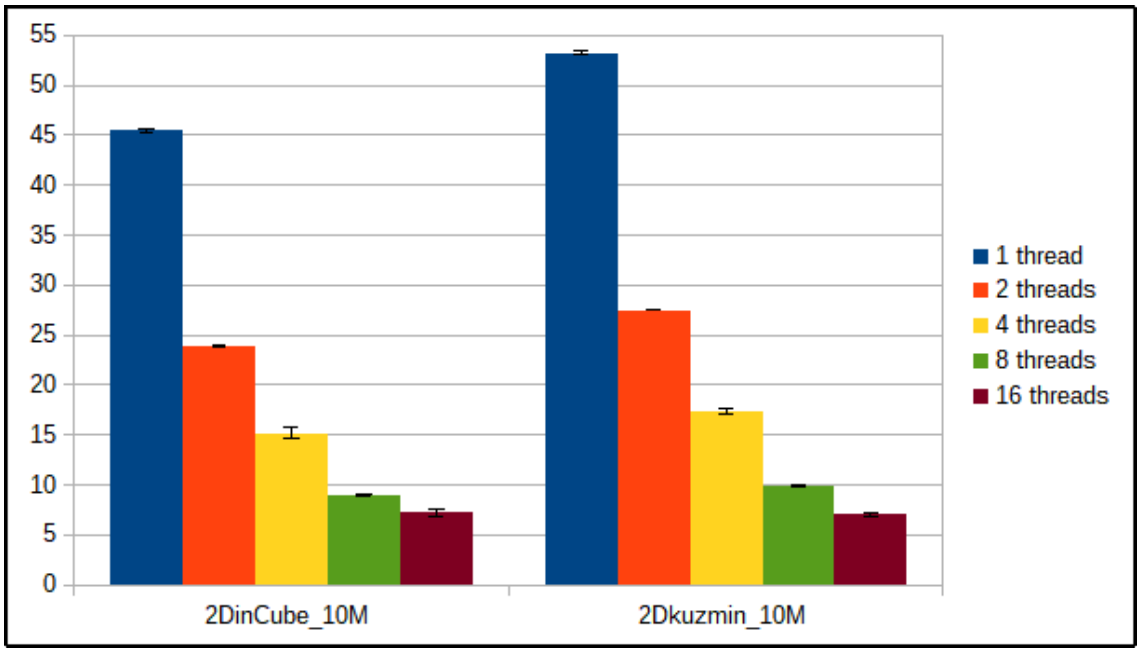


Figure A.359: Execution times (in minutes) of Range Query 2D ran on Intel 12-24 using LCWS

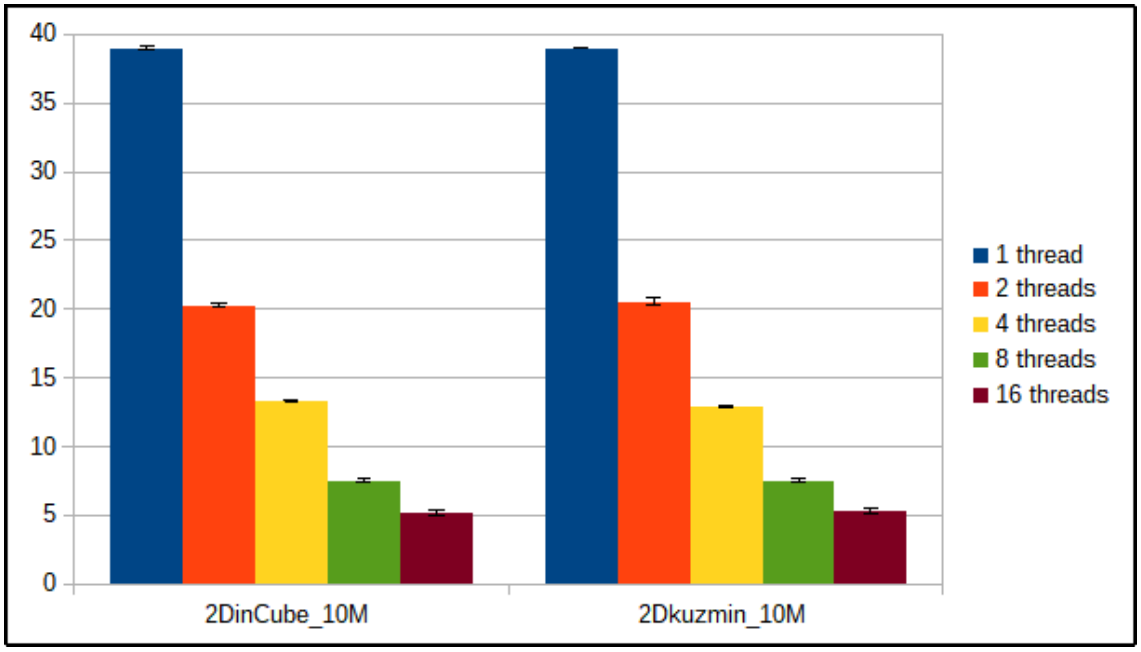


Figure A.360: Execution times (in minutes) of Range Query 2D ran on Intel 12-24 using SBLCWS (**first version**)

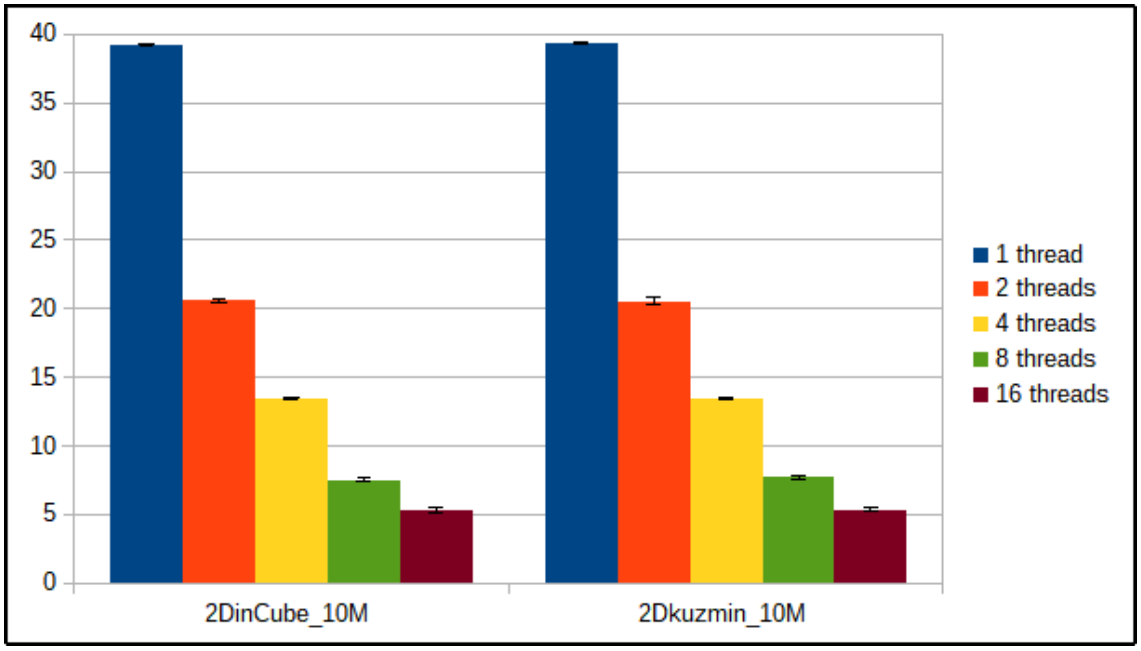


Figure A.361: Execution times (in minutes) of Range Query 2D ran on Intel 12-24 using SBLCWS (**second version**)

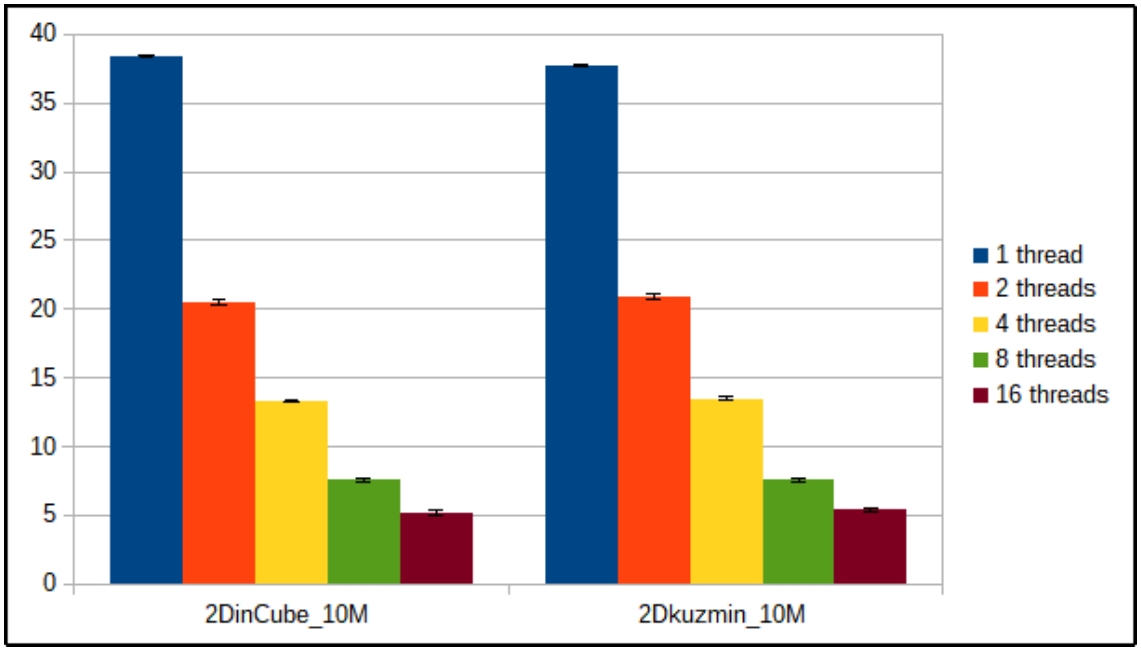


Figure A.362: Execution times (in minutes) of Range Query 2D ran on Intel 12-24 using LCWS with WShare (**tit-for-tat**).

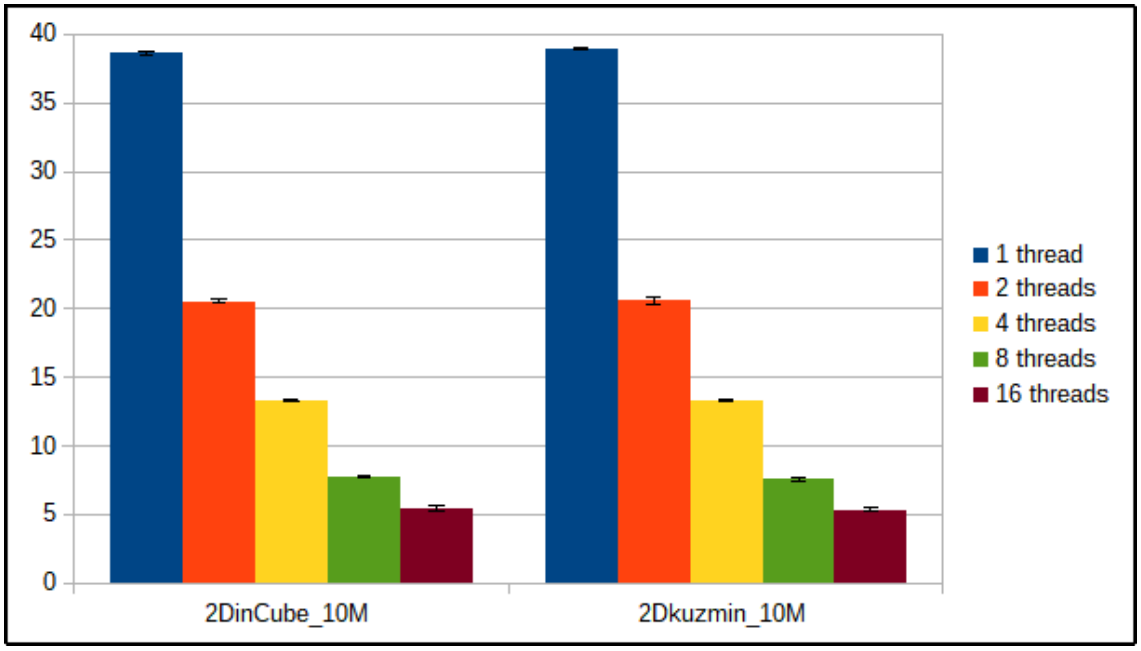


Figure A.363: Execution times (in minutes) of Range Query 2D ran on Intel 12-24 using LCWS with WShare (**continuous tit-for-tat**).

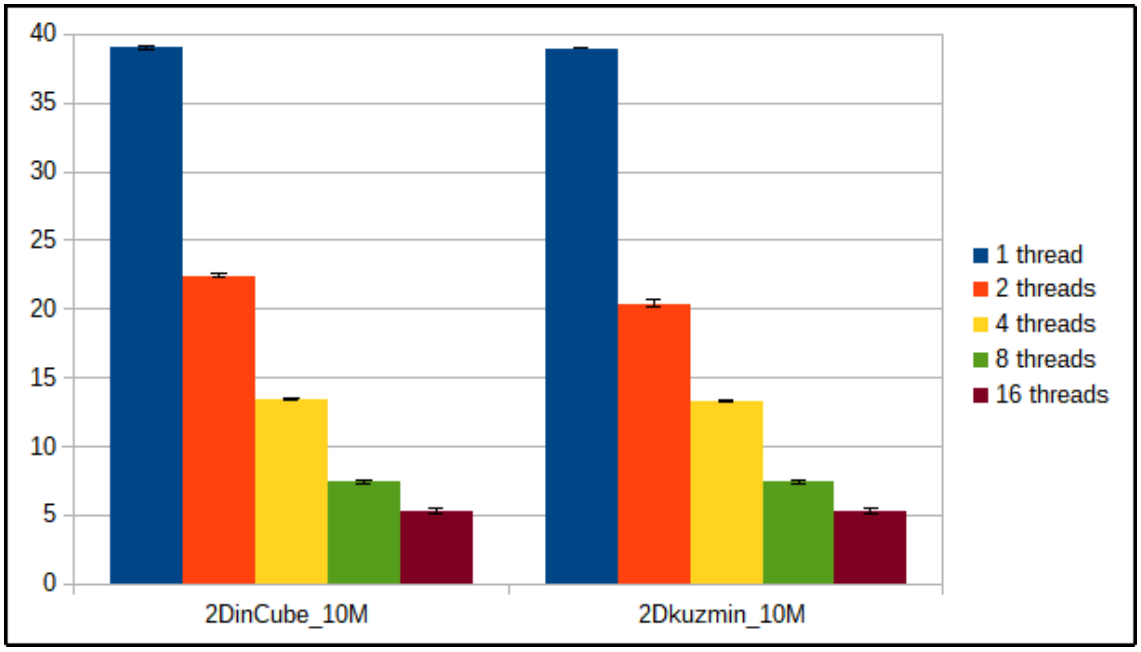


Figure A.364: Execution times (in minutes) of Range Query 2D ran on Intel 12-24 using SBLCWS with WShare (**first version**).

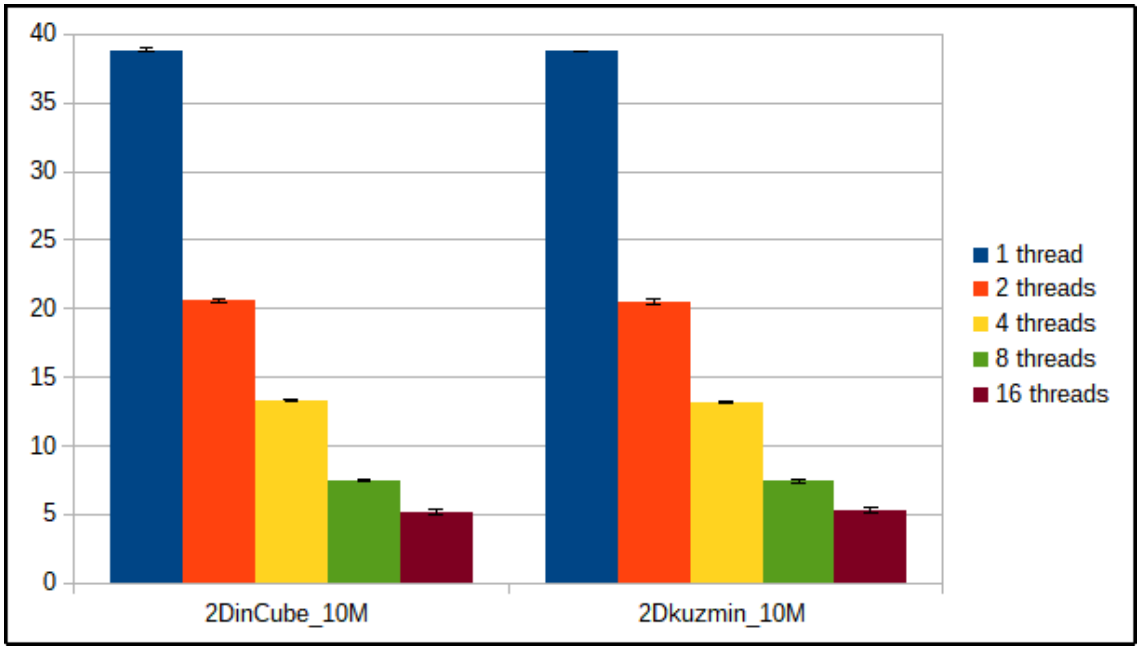


Figure A.365: Execution times (in minutes) of Range Query 2D ran on Intel 12-24 using SBLCWS with WShare (**second version**).

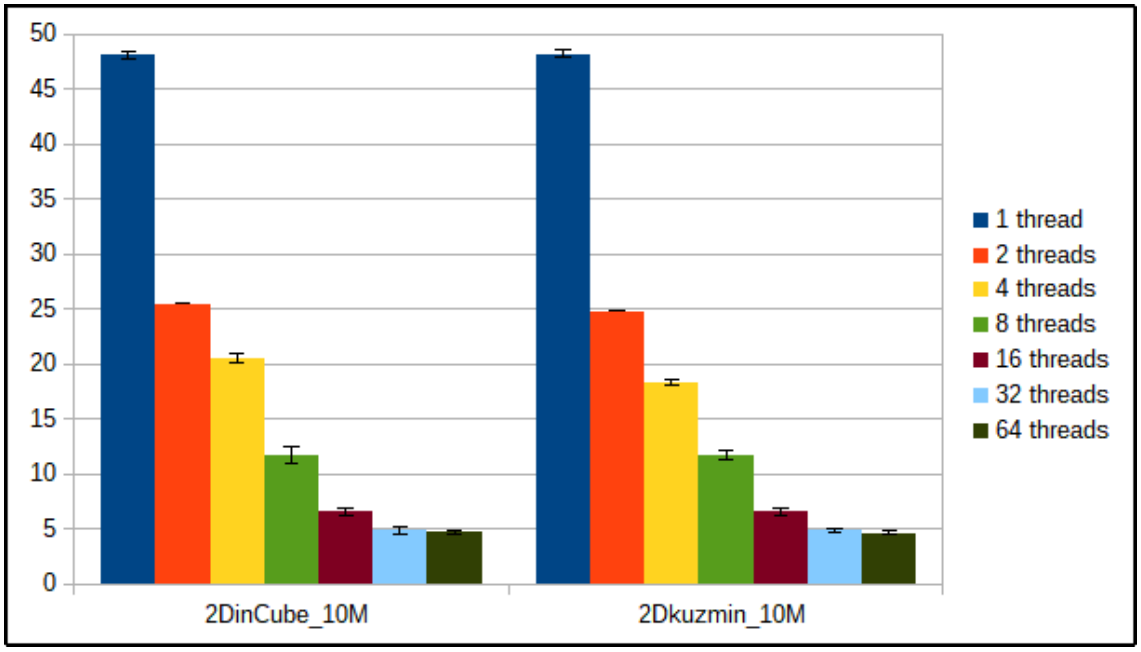


Figure A.366: Execution times (in minutes) of Range Query 2D ran on AMD 32-64 using WSteal

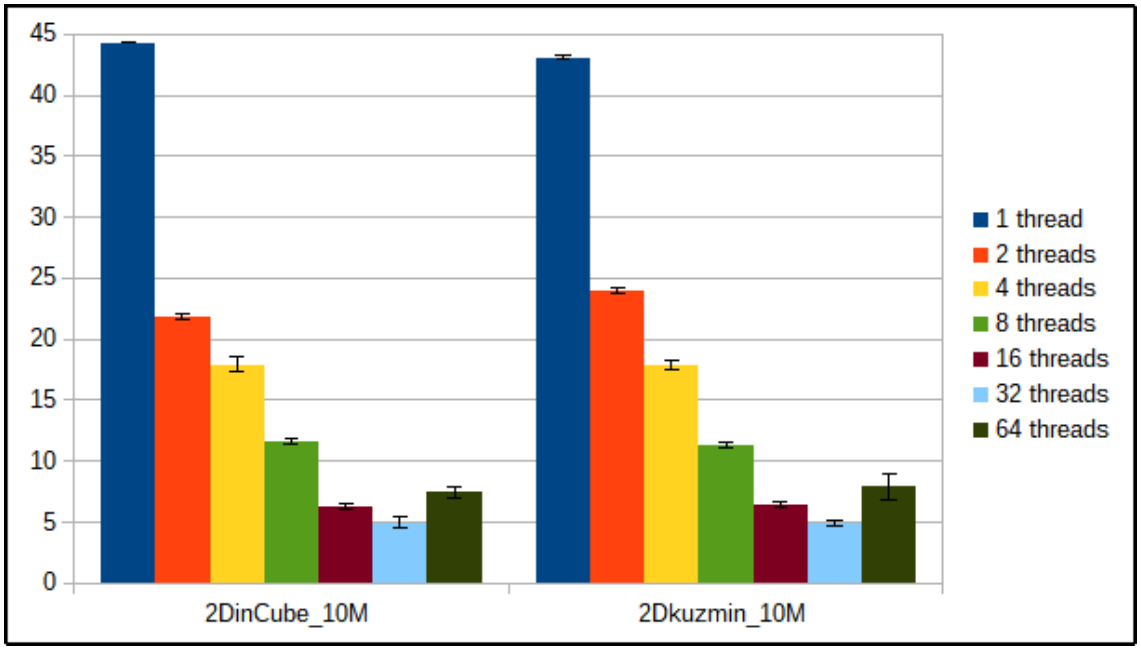


Figure A.367: Execution times (in minutes) of Range Query 2D ran on AMD 32-64 using LCWS

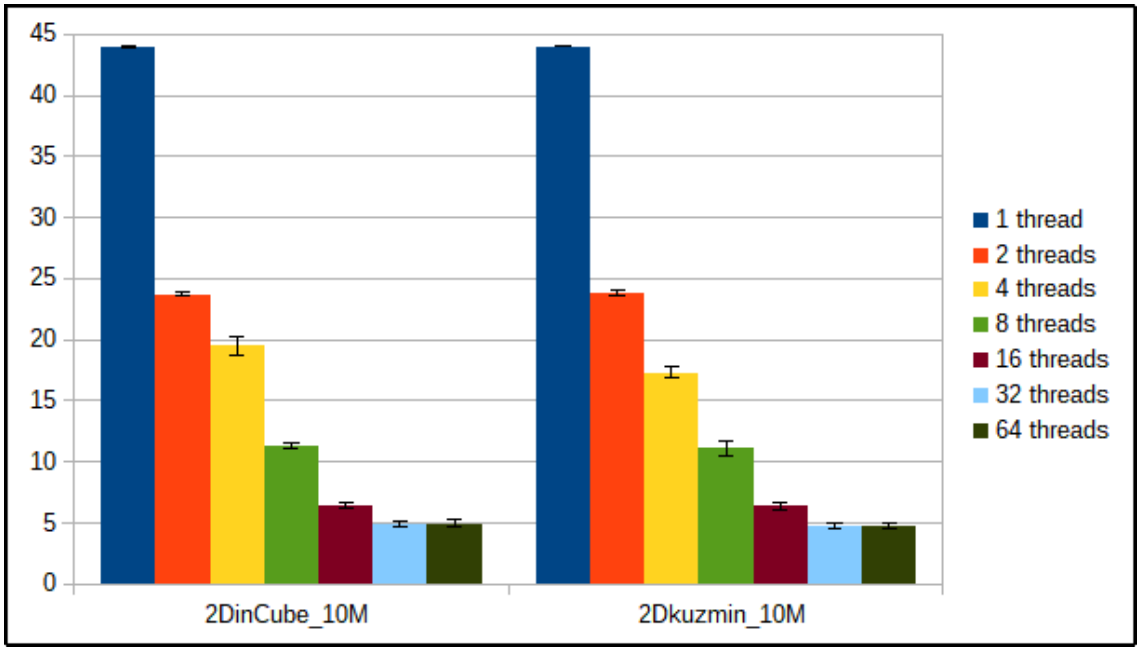


Figure A.368: Execution times (in minutes) of Range Query 2D ran on AMD 32-64 using SBLCWS (**first version**)

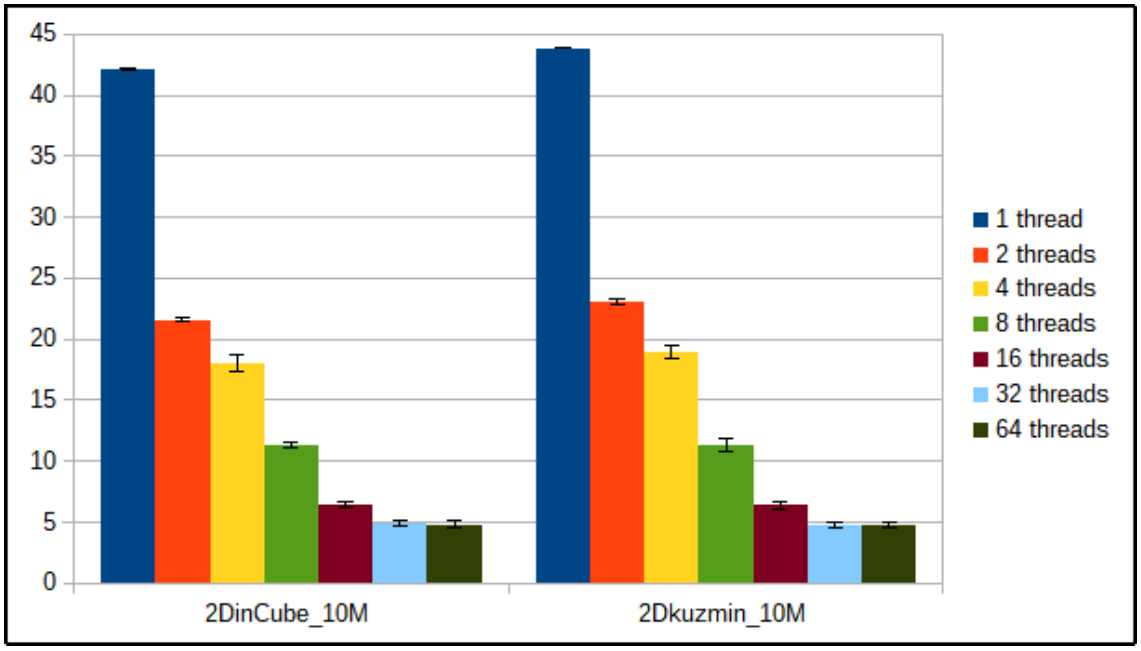


Figure A.369: Execution times (in minutes) of Range Query 2D ran on AMD 32-64 using SBLCWS (**second version**)

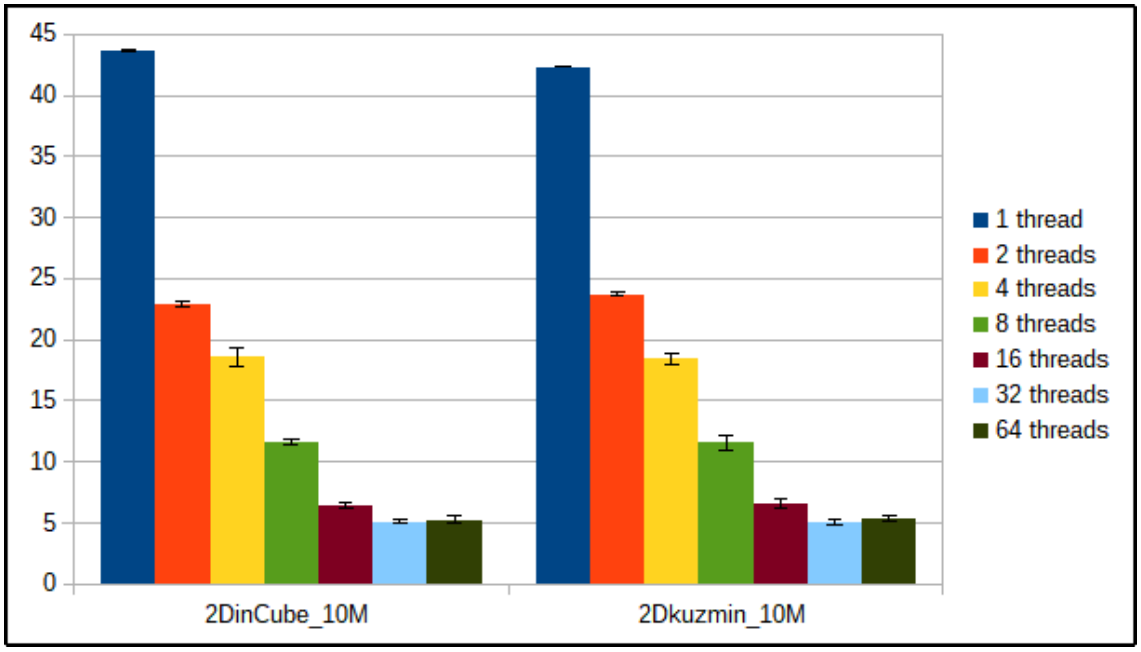


Figure A.370: Execution times (in minutes) of Range Query 2D ran on AMD 32-64 using LCWS with WShare (**tit-for-tat**).

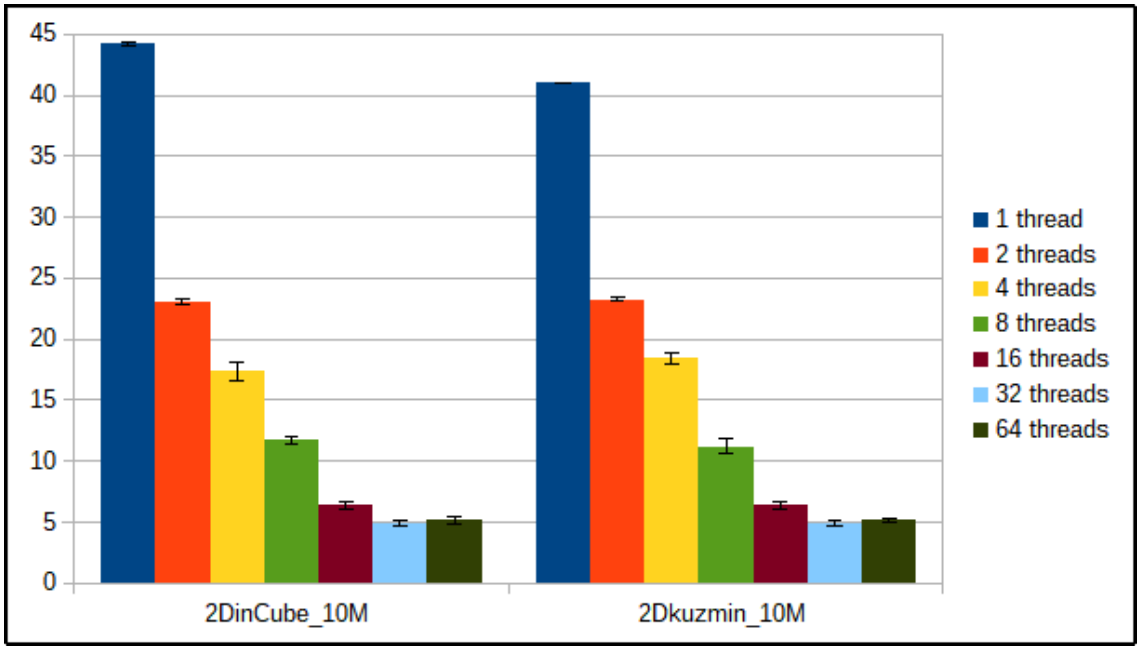


Figure A.371: Execution times (in minutes) of Range Query 2D ran on AMD 32-64 using LCWS with WShare (**continuous tit-for-tat**).

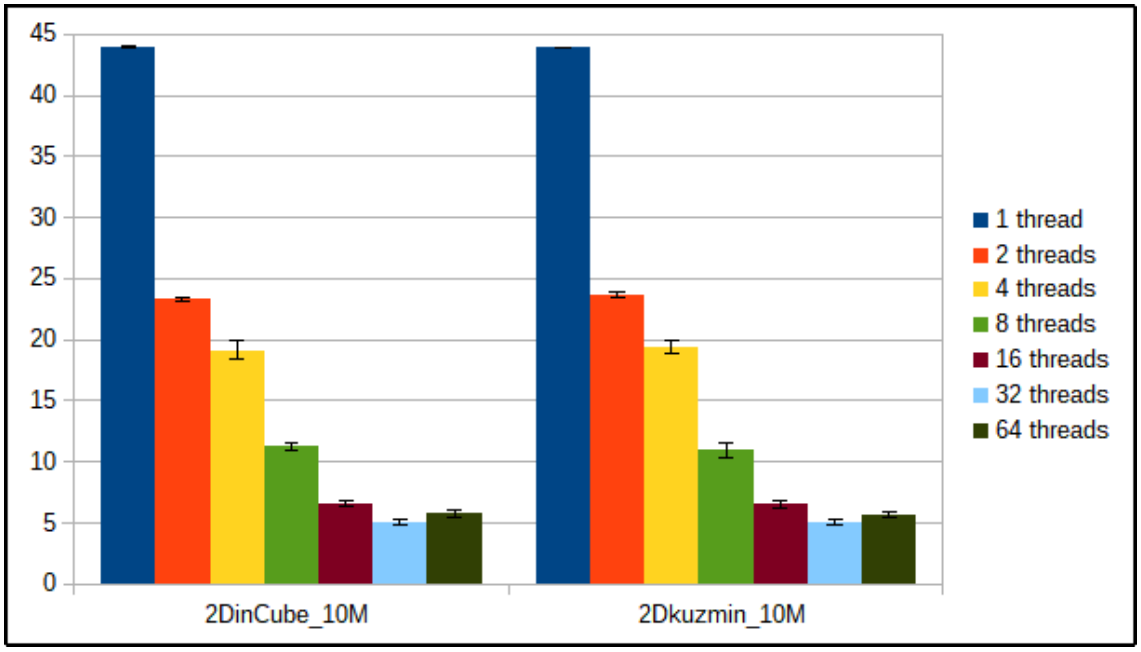


Figure A.372: Execution times (in minutes) of Range Query 2D ran on AMD 32-64 using SBLCWS with WShare (**first version**).

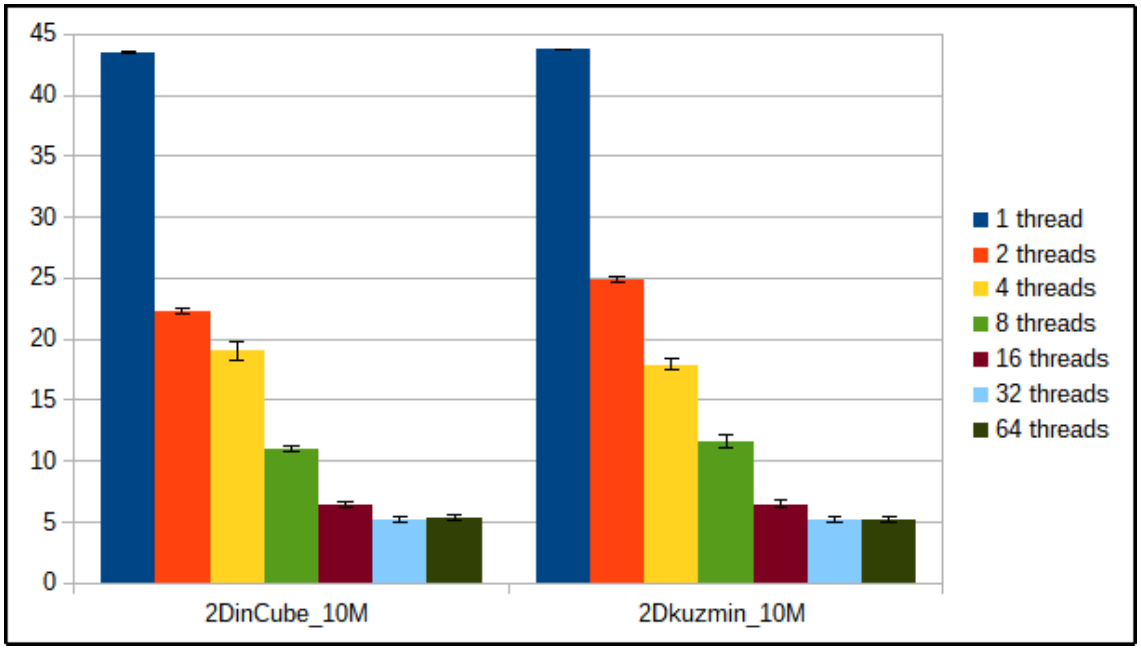


Figure A.373: Execution times (in minutes) of Range Query 2D ran on AMD 32-64 using SBLCWS with WShare (**second version**).

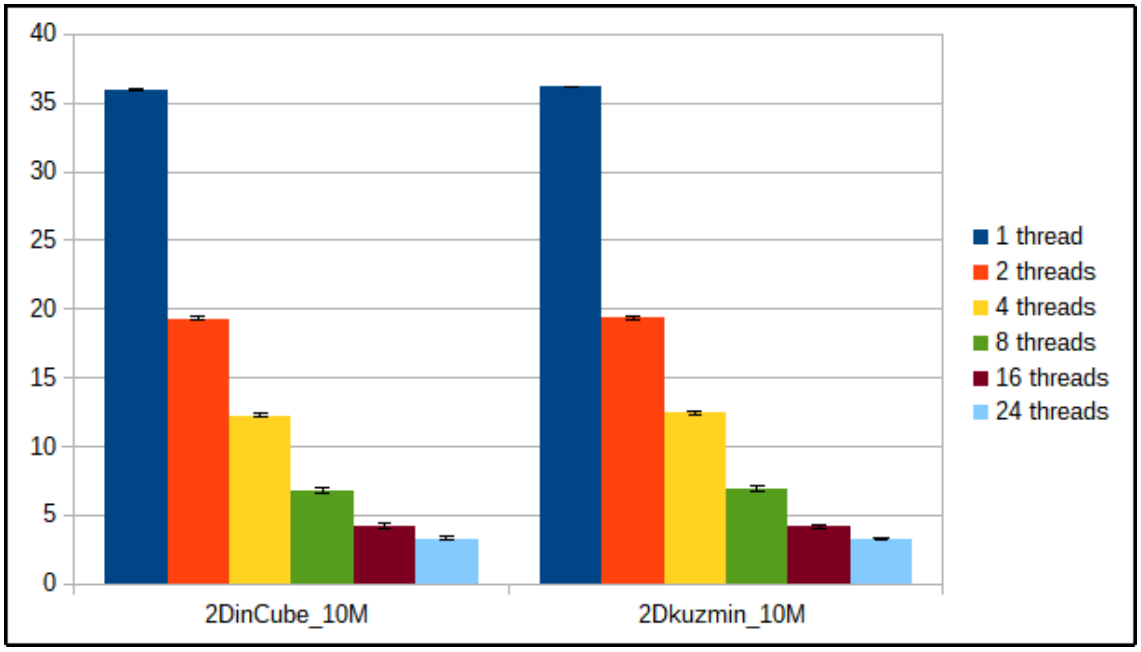


Figure A.374: Execution times (in minutes) of Range Query 2D ran on Intel 16-16 using WSteal

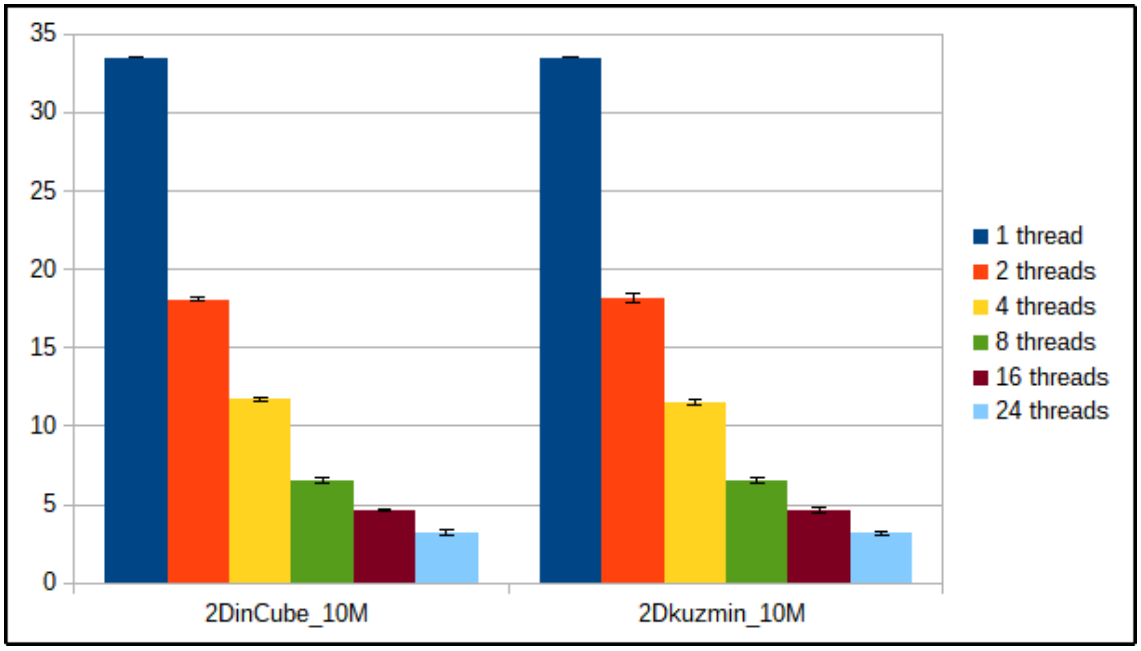


Figure A.375: Execution times (in minutes) of Range Query 2D ran on Intel 16-16 using LCWS

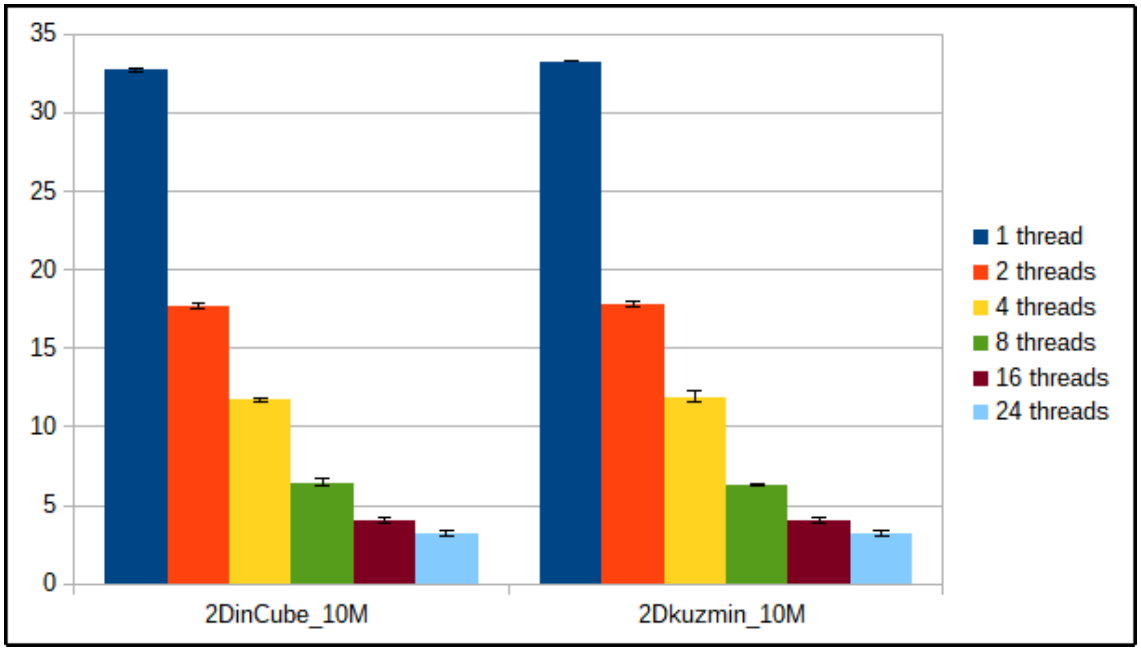


Figure A.376: Execution times (in minutes) of Range Query 2D ran on Intel 16-16 using SBLCWS (**first version**)

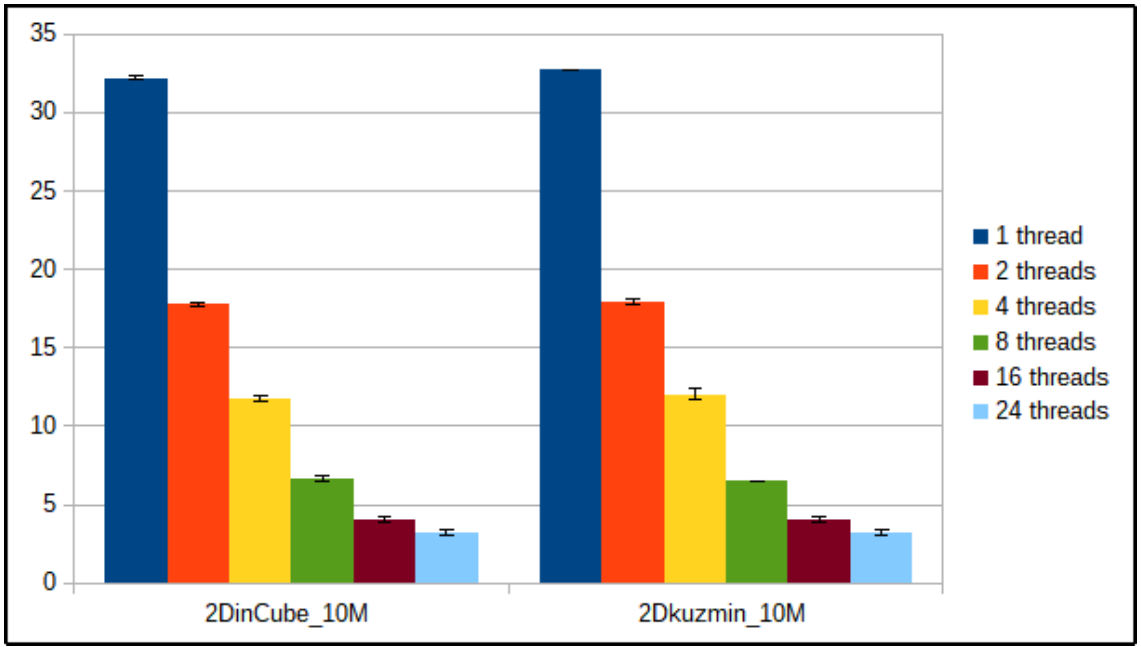


Figure A.377: Execution times (in minutes) of Range Query 2D ran on Intel 16-16 using SBLCWS (**second version**)

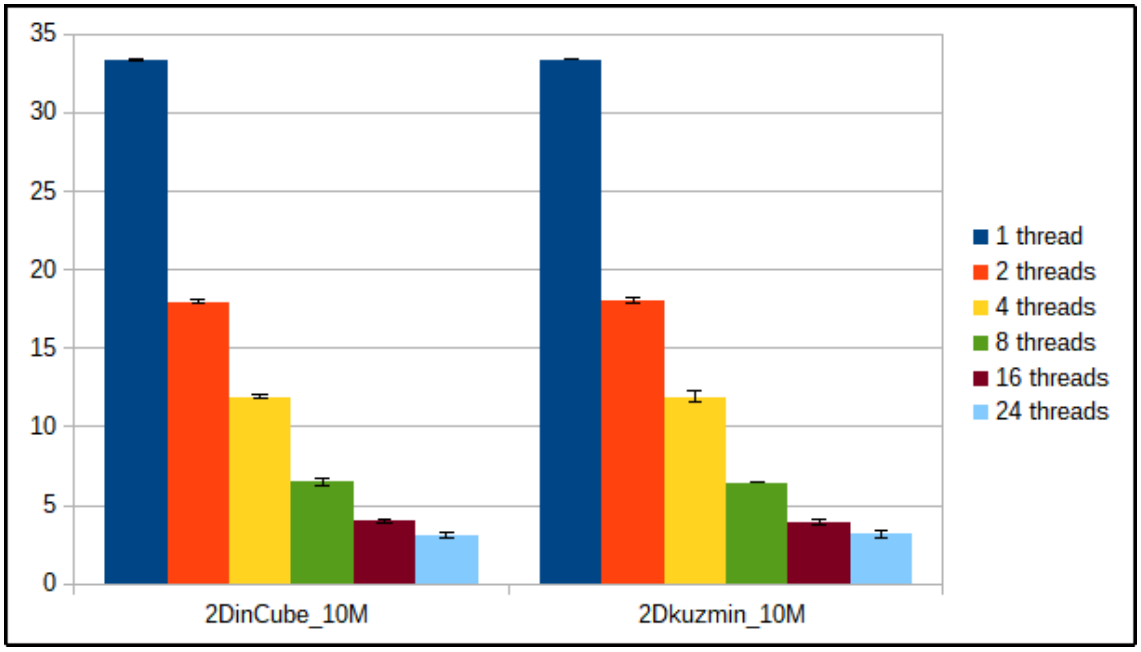


Figure A.378: Execution times (in minutes) of Range Query 2D ran on Intel 16-16 using LCWS with WShare (**tit-for-tat**).

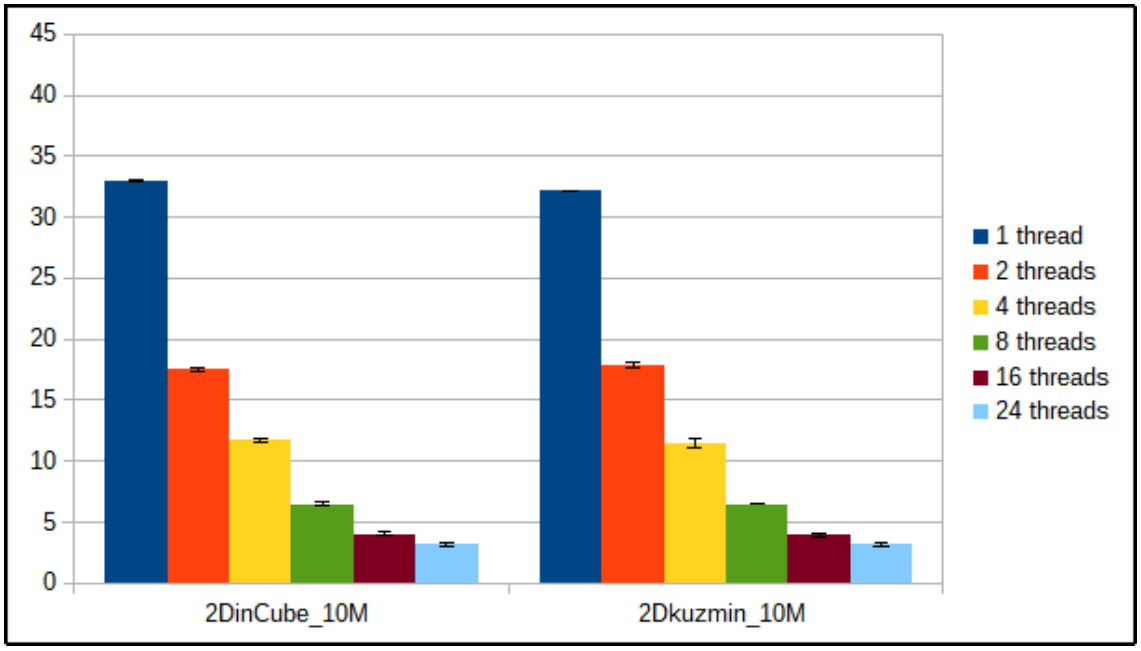


Figure A.379: Execution times (in minutes) of Range Query 2D ran on Intel 16-16 using LCWS with WShare (**continuous tit-for-tat**).

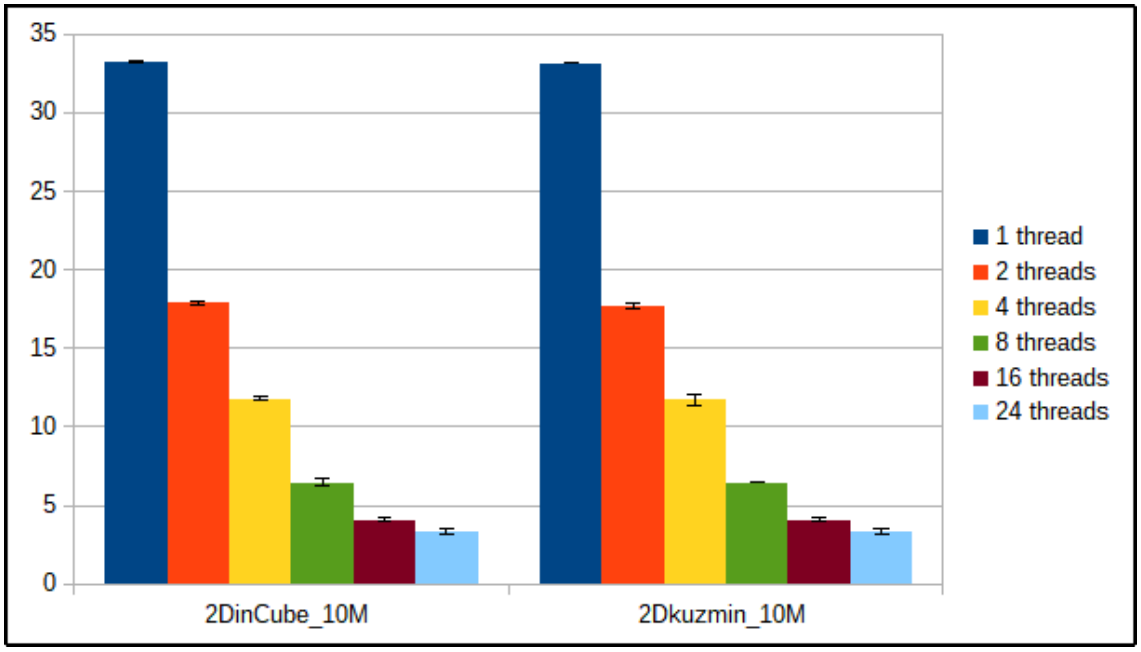


Figure A.380: Execution times (in minutes) of Range Query 2D ran on Intel 16-16 using SBLCWS with WShare (**first version**).

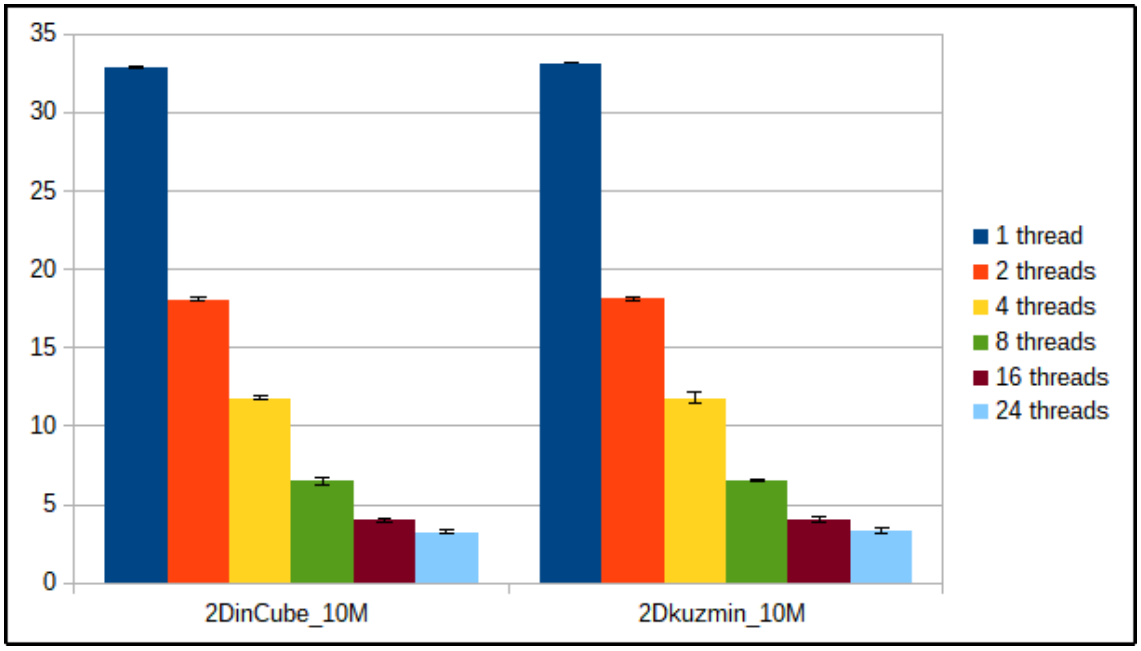


Figure A.381: Execution times (in minutes) of Range Query 2D ran on Intel 16-16 using SBLCWS with WShare (**second version**).

I Annex

I.1 C++ Implementations

I.1.1 C++ implementation of the LCWS algorithm.

```
const int RACE = 1;
const int EMPTY = 2;
const int ABORT = 3;
const int FULL_EMPTY = 4;

template <typename Job>
struct Wrapper {
Job *job = nullptr;
int error = -1;
};

struct Deque {
      using qidx = int;
      using tag_t = int;

      // use std::atomic<age_t> for atomic access.
      // Note: Explicit alignment specifier required
      // to ensure that Clang inlines atomic loads.
      struct alignas(int64_t) age_t {
        tag_t tag;
        qidx top;
      };

      // align to avoid false sharing
      struct alignas(64) padded_job {
        Job *job;
```

```
};

static constexpr int q_size = 1000;
qidx b_official;
qidx bot_private;
std::atomic < age_t > age;
std::array < padded_job, q_size > deq;

Deque(): b_official(0), bot_private(0), c1(0), c2(0), c3(0), c4(0), age(age_t {
  0,
  0
}) {}

void push_bottom(Job * job) {
  deq[bot_private++].job = job;

  if (bot_private == q_size) {
    throw std::runtime_error("internal error: scheduler queue overflow");
  }
}

Wrapper<Job> pop_top() {
  auto old_age = age.load(std::memory_order_relaxed);
  Wrapper<Job> wrap;

  if (b_official > old_age.top) {
    wrap.job = deq[old_age.top].job;
    auto new_age = old_age;
    new_age.top = new_age.top + 1;

    if (age.compare_exchange_strong(old_age, new_age, std::memory_order_relaxed, s
      return wrap;

    wrap.job = nullptr;
    wrap.error = ABORT;
  }
  else if(b_official < bot_private)
      wrap.error = EMPTY;
    else
      wrap.error = FULL_EMPTY;
```

```
  return wrap;
}

void update_bottom() {
    if(b_official < bot_private)
      b_official++;
}

Job * pop() {
  if (bot_private == b_official)
    return nullptr;

  auto item = deq[--bot_private].job;
  return item;
}

Job * pop_bottom() {
  Job * result = nullptr;

  if(b_official != 0) {
    auto b = --b_official;

    std::atomic_thread_fence(std::memory_order_seq_cst);
    auto job = deq[b].job;
    auto old_age = age.load(std::memory_order_relaxed);

      if (b > old_age.top) {
        bot_private = b;
        result = job;

      } else {

          bot_private = 0;
          auto age_new = age_t {
              old_age.tag + 1, 0
          };
          b_official = 0;

          if ((b == old_age.top) && age.compare_exchange_strong(old_age, age_new, s
              result = job;
          } else {
```

```
                age.store(age_new, std::memory_order_relaxed);
                result = nullptr;
            }

            std::atomic_thread_fence(std::memory_order_seq_cst);
          }
        }

    return result;
  }
};

template < typename Job >
struct scheduler {
  public:
  unsigned int num_threads;

  bool finished() {
    return finished_flag.load(std::memory_order_relaxed);
  }

  static thread_local unsigned int thread_id;

  scheduler(): num_threads(init_num_workers()),
  num_deques(num_threads),
  deques(num_deques),
  targeted(num_deques, false),
  attempts(num_deques),
  spawned_threads(),
  finished_flag(false) {

    // Stopping condition
    auto finished = [this]() {
      return finished_flag.load(std::memory_order_relaxed);
    };

    thread_id = 0; // thread-local write

    for (unsigned long i = 1; i < num_threads; i++) {
      spawned_threads.emplace_back([ & , i, finished]() {
        thread_id = i;
```

```
      start(finished);
    });
  }

  start(finished);
}

~scheduler() {
  finished_flag.store(true, std::memory_order_relaxed);
  for (unsigned int i = 1; i < num_threads; i++) {
    spawned_threads[i - 1].join();
  }
}

// Push onto local stack.
void spawn(Job * job) {
  int id = worker_id();
  deques[id].push_bottom(job);
}

// All scheduler threads quit after this is called.
void finish() {
  finished_flag.store(true, std::memory_order_relaxed);
}

// Pop from local stack.
Job * try_pop() {
  auto id = worker_id();
  auto job = deques[id].pop();
  if (!job)
    return deques[id].pop_bottom();

  if (targeted[id]) {
    targeted[id] = false;
    deques[id].update_bottom();
  }

  return job;
}

unsigned int init_num_workers() {
```

```
    if (const auto env_p = std::getenv("PARLAY_NUM_THREADS")) {
      return std::stoi(env_p);
    } else {
      return std::thread::hardware_concurrency();
    }
  }

  unsigned int worker_id() {
    return thread_id;
  }

  private:
    // Align to avoid false sharing.
    struct alignas(128) attempt {
      size_t val;
    };

  int num_deques;
  std::vector < Deque < Job >> deques;
  std::vector < attempt > attempts;
  std::vector < bool > targeted;
  std::vector < std::thread > spawned_threads;


  template < typename F >
  void start(F finished) {
    size_t id = worker_id();
    while (true) {

      Job * job = get_job(finished, id);
      if (!job)
        break;

      ( * job)();
    }
  }

  Job * try_steal(size_t id) {
    size_t target = (hash(id) + hash(attempts[id].val)) % num_deques;
    attempts[id].val++;
```

```
  auto wrap = deques[target].pop_top();

  if (wrap.error == EMPTY )
      targeted[target] = true;

  return wrap.job;
}


template < typename F >
Job * get_job(F finished, size_t id) {

  if (finished())
    return nullptr;
  Job * job = try_pop();
  if (job)
    return job;

  if (targeted[id])
    targeted[id] = false;

  while (true) {
    for (int i = 0; i <= num_deques * 100; i++) {
      if (finished())
        return nullptr;

      job = try_steal(id);

      if (job)
        return job;
    }
    std::this_thread::sleep_for(std::chrono::nanoseconds(num_deques * 100));
  }
}

size_t hash(uint64_t x) {
  x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9ULL;
  x = (x ^ (x >> 27)) * 0x94d049bb133111ebULL;
  x = x ^ (x >> 31);
  return static_cast < size_t > (x);
}
```

```
    };

    template < typename T >
    thread_local unsigned int scheduler < T > ::thread_id = 0;
```

I.1.2 C++ implementation of the SBLCWS (first version) algorithm.

```
const int RACE = 1;
const int EMPTY = 2;
const int ABORT = 3;
const int FULL_EMPTY = 4;

template <typename Job>
struct Wrapper {
Job *job = nullptr;
int error = -1;
};

struct Deque {
  using qidx = int;
  using tag_t = int;

  // use std::atomic<age_t> for atomic access.
  // Note: Explicit alignment specifier required
  // to ensure that Clang inlines atomic loads.
  struct alignas(int64_t) age_t {
    tag_t tag;
    qidx top;
  };

  // align to avoid false sharing
  struct alignas(64) padded_job {
    Job *job;
  };

  static constexpr int q_size = 1000;
  qidx b_official;
  qidx bot_private;
  std::atomic < age_t > age;
  std::array < padded_job, q_size > deq;
```

```
bool targeted;

Deque(): b_official(0), bot_private(0), c1(0), c2(0), c3(0), c4(0), age(age_t {
  0,
  0
}) {}

void push_bottom(Job * job) {
  deq[bot_private++].job = job;

  if (bot_private == q_size) {
    throw std::runtime_error("internal error: scheduler queue overflow");
  }
  targeted = false;
}

Wrapper<Job> pop_top() {
  auto old_age = age.load(std::memory_order_relaxed);
  Wrapper<Job> wrap;

  if (b_official > old_age.top) {
    wrap.job = deq[old_age.top].job;
    auto new_age = old_age;
    new_age.top = new_age.top + 1;

    if (age.compare_exchange_strong(old_age, new_age, std::memory_order_relaxed, std::m
      targeted = false;
      return wrap;
    }

    wrap.job = nullptr;
    wrap.error = ABORT;
  }
  else if(b_official < bot_private)
      wrap.error = EMPTY;
    else
      wrap.error = FULL_EMPTY;

  return wrap;
}
```

```
  void update_bottom() {
      if(b_official + 1 < bot_private)
        b_official++;
  }

  Job * pop() {
    if (bot_private == b_official)
      return nullptr;

    auto item = deq[--bot_private].job;
    return item;
  }

  Job * pop_bottom() {
    Job * result = nullptr;

    if(b_official != 0) {
      auto b = --b_official;

      std::atomic_thread_fence(std::memory_order_seq_cst);
      auto job = deq[b].job;
      auto old_age = age.load(std::memory_order_relaxed);

        if (b > old_age.top) {
          bot_private = b;
          result = job;

        } else {

            bot_private = 0;
            auto age_new = age_t {
                old_age.tag + 1, 0
            };
            b_official = 0;

            if ((b == old_age.top) && age.compare_exchange_strong(old_age, age_new, std:
                result = job;
            } else {
                age.store(age_new, std::memory_order_relaxed);
                result = nullptr;
            }
```

```
            std::atomic_thread_fence(std::memory_order_seq_cst);
          }
        }

    return result;
  }

  bool check() {
    return b_official + 1< bot_private;
  }
};

template < typename Job >
struct scheduler {
  public:
  unsigned int num_threads;

  bool finished() {
    return finished_flag.load(std::memory_order_relaxed);
  }

  static void signal_handler(int sig) {
    static_deques->at(thread_id).update_bottom();
  }

  static thread_local unsigned int thread_id;

  scheduler(): num_threads(init_num_workers()),
  num_deques(num_threads),
  deques(num_deques),
  attempts(num_deques),
  spawned_threads(),
  thread_handles(),
  finished_flag(false) {

    static_deques = &deques;
    std::signal(SIGUSR1, scheduler<Job>::signal_handler);

    // Stopping condition
    auto finished = [this]() {
```

```
      return finished_flag.load(std::memory_order_relaxed);
    };

    thread_id = 0; // thread-local write

    thread_handles[0] = pthread_self();
    for (unsigned long i = 1; i < num_threads; i++) {
      deques[i].id = i;
      spawned_threads.emplace_back([ & , i, finished]() {
        thread_id = i;
        start(finished);
      });
    thread_handles.emplace_back(spawned_threads[i - 1].native_handle());
    }

    start(finished);
  }

  ~scheduler() {
    finished_flag.store(true, std::memory_order_relaxed);
    for (unsigned int i = 1; i < num_threads; i++) {
      spawned_threads[i - 1].join();
    }
  }

  // Push onto local stack.
  void spawn(Job * job) {
    int id = worker_id();
    deques[id].push_bottom(job);
  }

  // All scheduler threads quit after this is called.
  void finish() {
    finished_flag.store(true, std::memory_order_relaxed);
  }

  // Pop from local stack.
  Job * try_pop() {
    auto id = worker_id();
    auto job = deques[id].pop();
    if (!job)
```

```
    return deques[id].pop_bottom();

  return job;
}

unsigned int init_num_workers() {
  if (const auto env_p = std::getenv("PARLAY_NUM_THREADS")) {
    return std::stoi(env_p);
  } else {
    return std::thread::hardware_concurrency();
  }
}

unsigned int worker_id() {
  return thread_id;
}

private:
  // Align to avoid false sharing.
  struct alignas(128) attempt {
    size_t val;
  };

int num_deques;
std::vector < Deque < Job >> deques;
std::vector < attempt > attempts;
std::vector < std::thread > spawned_threads;
std::vector <pthread_t> thread_handles;
static std::vector<Deque<Job>>* static_deques;


template < typename F >
void start(F finished) {
  size_t id = worker_id();
  while (true) {

    Job * job = get_job(finished, id);
    if (!job)
      break;

    ( * job)();
```

```
    }
  }

  Job * try_steal(size_t id) {
    size_t target = (hash(id) + hash(attempts[id].val)) % num_deques;
    attempts[id].val++;

    auto wrap = deques[target].pop_top();

    if (wrap.error == EMPTY) {
        deques[target].targeted = true;
        pthread_kill(thread_handles[target], SIGUSR1);
    }

    return wrap.job;
  }


  template < typename F >
  Job * get_job(F finished, size_t id) {

    if (finished())
      return nullptr;
    Job * job = try_pop();
    if (job)
      return job;

    if (targeted[id])
      targeted[id] = false;

    while (true) {
      for (int i = 0; i <= num_deques * 100; i++) {
        if (finished())
          return nullptr;

        job = try_steal(id);

        if (job)
          return job;
      }
      std::this_thread::sleep_for(std::chrono::nanoseconds(num_deques * 100));
```

```
    }
  }

  size_t hash(uint64_t x) {
    x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9ULL;
    x = (x ^ (x >> 27)) * 0x94d049bb133111ebULL;
    x = x ^ (x >> 31);
    return static_cast < size_t > (x);
  }
};

template < typename T >
thread_local unsigned int scheduler < T > ::thread_id = 0;

template < typename T >
std::vector<Deque<T>>* scheduler<T>::static_deques = nullptr;
```

I.1.3 C++ implementation of the SBLCWS (second version) algorithm.

```
const int RACE = 1;
const int EMPTY = 2;
const int ABORT = 3;
const int FULL_EMPTY = 4;

template <typename Job>
struct Wrapper {
Job *job = nullptr;
int error = -1;
};

struct Deque {
      using qidx = int;
      using tag_t = int;

      // use std::atomic<age_t> for atomic access.
      // Note: Explicit alignment specifier required
      // to ensure that Clang inlines atomic loads.
      struct alignas(int64_t) age_t {
        tag_t tag;
        qidx top;
      };
```

```
// align to avoid false sharing
struct alignas(64) padded_job {
  Job *job;
};

static constexpr int q_size = 1000;
volatile qidx b_official;
volatile qidx bot_private;
std::atomic < age_t > age;
std::array < padded_job, q_size > deq;

bool targeted;

Deque(): b_official(0), bot_private(0), c1(0), c2(0), c3(0), c4(0), age(age_t {
  0,
  0
}) {}

void push_bottom(Job * job) {
  deq[bot_private++].job = job;

  if (bot_private == q_size) {
    throw std::runtime_error("internal error: scheduler queue overflow");
  }
  targeted = false;
}

Wrapper<Job> pop_top() {
  auto old_age = age.load(std::memory_order_relaxed);
  Wrapper<Job> wrap;

  if (b_official > old_age.top) {
    wrap.job = deq[old_age.top].job;
    auto new_age = old_age;
    new_age.top = new_age.top + 1;

    if (age.compare_exchange_strong(old_age, new_age, std::memory_order_relaxed, s
      targeted = false;
      return wrap;
    }
```

```
    wrap.job = nullptr;
    wrap.error = ABORT;
  }
  else if(b_official < bot_private)
      wrap.error = EMPTY;
    else
      wrap.error = FULL_EMPTY;

  return wrap;
}

void update_bottom() {
    if(b_official < bot_private)
      b_official++;
}

Job * pop() {
  --bot_private;

  if (bot_private == b_official)
    return nullptr;

  auto item = deq[bot_private].job;
  return item;
}

Job * pop_bottom() {
  Job * result = nullptr;

  if(b_official != 0) {
    auto b = --b_official;

    std::atomic_thread_fence(std::memory_order_seq_cst);
    auto job = deq[b].job;
    auto old_age = age.load(std::memory_order_relaxed);

      if (b > old_age.top) {
        bot_private = b;
        result = job;
```

```
            } else {

                bot_private = 0;
                auto age_new = age_t {
                    old_age.tag + 1, 0
                };
                b_official = 0;

                if ((b == old_age.top) && age.compare_exchange_strong(old_age, age_new, s
                    result = job;
                } else {
                    age.store(age_new, std::memory_order_relaxed);
                    result = nullptr;
                }

                std::atomic_thread_fence(std::memory_order_seq_cst);
            }
          }
          else bot_private = 0;

      return result;
    }

    bool check() {
      return b_official < bot_private;
    }
  };

  template < typename Job >
  struct scheduler {
    public:
    unsigned int num_threads;

    bool finished() {
      return finished_flag.load(std::memory_order_relaxed);
    }

    static void signal_handler(int sig) {
      static_deques->at(thread_id).update_bottom();
    }
```

```
static thread_local unsigned int thread_id;

scheduler(): num_threads(init_num_workers()),
num_deques(num_threads),
deques(num_deques),
attempts(num_deques),
spawned_threads(),
thread_handles(),
finished_flag(false) {

  static_deques = &deques;
  std::signal(SIGUSR1, scheduler<Job>::signal_handler);

  // Stopping condition
  auto finished = [this]() {
    return finished_flag.load(std::memory_order_relaxed);
  };

  thread_id = 0; // thread-local write

  thread_handles[0] = pthread_self();
  for (unsigned long i = 1; i < num_threads; i++) {
    deques[i].id = i;
    spawned_threads.emplace_back([ & , i, finished]() {
      thread_id = i;
      start(finished);
    });
  thread_handles.emplace_back(spawned_threads[i - 1].native_handle());
  }

  start(finished);
}

~scheduler() {
  finished_flag.store(true, std::memory_order_relaxed);
  for (unsigned int i = 1; i < num_threads; i++) {
    spawned_threads[i - 1].join();
  }
}

// Push onto local stack.
```

```
void spawn(Job * job) {
  int id = worker_id();
  deques[id].push_bottom(job);
}

// All scheduler threads quit after this is called.
void finish() {
  finished_flag.store(true, std::memory_order_relaxed);
}

// Pop from local stack.
Job * try_pop() {
  auto id = worker_id();
  auto job = deques[id].pop();
  if (!job)
    return deques[id].pop_bottom();

  return job;
}

unsigned int init_num_workers() {
  if (const auto env_p = std::getenv("PARLAY_NUM_THREADS")) {
    return std::stoi(env_p);
  } else {
    return std::thread::hardware_concurrency();
  }
}

unsigned int worker_id() {
  return thread_id;
}

private:
  // Align to avoid false sharing.
  struct alignas(128) attempt {
    size_t val;
  };

int num_deques;
std::vector < Deque < Job >> deques;
std::vector < attempt > attempts;
```

```
std::vector < std::thread > spawned_threads;
std::vector <pthread_t> thread_handles;
static std::vector<Deque<Job>>* static_deques;


template < typename F >
void start(F finished) {
  size_t id = worker_id();
  while (true) {

    Job * job = get_job(finished, id);
    if (!job)
      break;

    ( * job)();
  }
}

Job * try_steal(size_t id) {
  size_t target = (hash(id) + hash(attempts[id].val)) % num_deques;
  attempts[id].val++;

  auto wrap = deques[target].pop_top();

  if (wrap.error == EMPTY) {
      deques[target].targeted = true;
      pthread_kill(thread_handles[target], SIGUSR1);
  }

  return wrap.job;
}


template < typename F >
Job * get_job(F finished, size_t id) {

  if (finished())
    return nullptr;
  Job * job = try_pop();
  if (job)
    return job;
```

```
        if (targeted[id])
          targeted[id] = false;

        while (true) {
          for (int i = 0; i <= num_deques * 100; i++) {
            if (finished())
              return nullptr;

            job = try_steal(id);

            if (job)
              return job;
          }
          std::this_thread::sleep_for(std::chrono::nanoseconds(num_deques * 100));
        }
      }

      size_t hash(uint64_t x) {
        x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9ULL;
        x = (x ^ (x >> 27)) * 0x94d049bb133111ebULL;
        x = x ^ (x >> 31);
        return static_cast < size_t > (x);
      }
    };

    template < typename T >
    thread_local unsigned int scheduler < T > ::thread_id = 0;

    template < typename T >
    std::vector<Deque<T>>* scheduler<T>::static_deques = nullptr;
```

I.1.4 C++ implementation of the LCWS algorithm with work giving.

```
const int RACE = 1;
const int EMPTY = 2;
const int ABORT = 3;
const int FULL_EMPTY = 4;

const int FALSE = 0;
const int TRUE = 1;
```

```
//Tit-for-tat heuristic
/*
const int HEURISTIC_SPREAD[3] = {FALSE,FALSE};
const int HEURISTIC_STEAL[4] = {TRUE,TRUE,TRUE};
*/


//Continuous Tit-for-tat heuristic
const int HEURISTIC_SPREAD[3] = {TRUE,FALSE};
const int HEURISTIC_STEAL[4] = {TRUE,TRUE,TRUE};


const int WORKING = -1;
const int IDLE = -2;

template <typename Job>
struct Wrapper {
Job *job = nullptr;
int error = -1;
};

struct Deque {
      using qidx = int;
      using tag_t = int;

      // use std::atomic<age_t> for atomic access.
      // Note: Explicit alignment specifier required
      // to ensure that Clang inlines atomic loads.
      struct alignas(int64_t) age_t {
        tag_t tag;
        qidx top;
      };

      // align to avoid false sharing
      struct alignas(64) padded_job {
        Job *job;
      };

      static constexpr int q_size = 1000;
      qidx b_official;
```

```
qidx bot_private;
std::atomic < age_t > age;
std::array < padded_job, q_size > deq;

Deque(): b_official(0), bot_private(0), c1(0), c2(0), c3(0), c4(0), age(age_t {
  0,
  0
}) {}

void push_bottom(Job * job) {
  deq[bot_private++].job = job;

  if (bot_private == q_size) {
    throw std::runtime_error("internal error: scheduler queue overflow");
  }
}

Wrapper<Job> pop_top() {
  auto old_age = age.load(std::memory_order_relaxed);
  Wrapper<Job> wrap;

  if (b_official > old_age.top) {
    wrap.job = deq[old_age.top].job;
    auto new_age = old_age;
    new_age.top = new_age.top + 1;

    if (age.compare_exchange_strong(old_age, new_age, std::memory_order_relaxed, s
      return wrap;
    }

    wrap.job = nullptr;
    wrap.error = ABORT;
  }
  else if(b_official < bot_private)
      wrap.error = EMPTY;
    else
      wrap.error = FULL_EMPTY;

  return wrap;
}
```

```
void update_bottom() {
    if(b_official < bot_private)
      b_official++;
}

Job * pop() {
  if (bot_private == b_official)
    return nullptr;

  auto item = deq[--bot_private].job;
  return item;
}

Job * pop_bottom() {
  Job * result = nullptr;

  if(b_official != 0) {
    auto b = --b_official;

    std::atomic_thread_fence(std::memory_order_seq_cst);
    auto job = deq[b].job;
    auto old_age = age.load(std::memory_order_relaxed);

      if (b > old_age.top) {
        bot_private = b;
        result = job;

      } else {

          bot_private = 0;
          auto age_new = age_t {
              old_age.tag + 1, 0
          };
          b_official = 0;

          if ((b == old_age.top) && age.compare_exchange_strong(old_age, age_new, s
              result = job;
          } else {
              age.store(age_new, std::memory_order_relaxed);
              result = nullptr;
          }
```

```
            std::atomic_thread_fence(std::memory_order_seq_cst);
          }
        }

      return result;
    }

};

template < typename Job >
struct scheduler {
  public:
  unsigned int num_threads;

  bool finished() {
    return finished_flag.load(std::memory_order_relaxed);
  }

  static thread_local unsigned int thread_id;

  scheduler(): num_threads(init_num_workers()),
  num_deques(num_threads),
  deques(num_deques),
  attempts(num_deques),
  spawned_threads(),
  rand_devices(num_deques),
  worker_state(num_deques, IDLE),
  worker_heuristic(num_deques),
  finished_flag(false) {
    // Stopping condition
    auto finished = [this]() {
      return finished_flag.load(std::memory_order_relaxed);
    };

    for(int i = 0; i < num_deques; i++) {
      generator.push_back(std::mt19937(rand_devices[i]()));
    }

    thread_id = 0; // thread-local write
```

```
  for (unsigned long i = 1; i < num_threads; i++) {
    deques[i].id = i;
    spawned_threads.emplace_back([ & , i, finished]() {
      thread_id = i;
      start(finished);
    });
  }

  start(finished);
}

~scheduler() {
  finished_flag.store(true, std::memory_order_relaxed);
  for (unsigned int i = 1; i < num_threads; i++) {
    spawned_threads[i - 1].join();
  }
}

// Push onto local stack.
void spawn(Job * job) {
  int id = worker_id();
  deques[id].push_bottom(job);
}

// All scheduler threads quit after this is called.
void finish() {
  finished_flag.store(true, std::memory_order_relaxed);
}

// Pop from local stack.
Job * try_pop() {
  auto id = worker_id();
  auto job = deques[id].pop();
  if (!job)
    return deques[id].pop_bottom();

  if(worker_heuristic[id] == TRUE)
  spread();

  return job;
}
```

```
unsigned int init_num_workers() {
  if (const auto env_p = std::getenv("PARLAY_NUM_THREADS")) {
    return std::stoi(env_p);
  } else {
    return std::thread::hardware_concurrency();
  }
}

unsigned int worker_id() {
  return thread_id;
}



private:
  // Align to avoid false sharing.
  struct alignas(128) attempt {
    size_t val;
  };

int num_deques;
std::vector < Deque < Job >> deques;
std::vector < attempt > attempts;
std::vector < std::thread > spawned_threads;

std::vector<int> worker_state;
std::vector<int> worker_heuristic;

std::vector<std::random_device> rand_devices;
std::vector<std::mt19937> generator;
std::uniform_int_distribution<int>  distr{0, 100};



template < typename F >
void start(F finished) {
  size_t id = worker_id();
  while (true) {

    Job * job = get_job(finished, id);
    if (!job)
      break;
```

```
      ( * job)();
    }
  }

void update(int id, int new_state) {
    worker_heuristic[id] = new_state;
}

void spread() {
    auto id = worker_id();

    choice = distr(generator[id]) % num_deques;

    if(worker_state[choice] == IDLE) {
        deques[id].update_bottom();
        worker_state[choice] = id;
        update(id, HEURISTIC_SPREAD[0]);
    }
    else
        update(id, HEURISTIC_SPREAD[1]);

}

  Job * try_steal(size_t id) {
    size_t target = (hash(id) + hash(attempts[id].val)) % num_deques;
    attempts[id].val++;

    auto wrap = deques[target].pop_top();
    auto job = wrap.job;
    auto error = wrap.error;

    choice = distr(generator[id]) % num_deques;

    if(error == ABORT)
        update(choice, HEURISTIC_STEAL[0]);
    else if(error == EMPTY)
        update(choice, HEURISTIC_STEAL[1]);
    else
        update(choice, HEURISTIC_STEAL[2]);
```

```
    if(job)
      worker_state[id] = WORKING;
    else {
      if(worker_state[id] != IDLE) {
        job = deques[worker_state[id]].pop_top().job;
        if(job)
          worker_state[id] = WORKING;
        else
          worker_state[id] = IDLE;
      }
    }

    return job;
  }


  template < typename F >
  Job * get_job(F finished, size_t id) {

    if (finished())
      return nullptr;
    Job * job = try_pop();
    if (job)
      return job;

    while (true) {
      for (int i = 0; i <= num_deques * 100; i++) {
        if (finished())
          return nullptr;

        job = try_steal(id);

        if (job)
          return job;
      }
      std::this_thread::sleep_for(std::chrono::nanoseconds(num_deques * 100));
    }
  }

  size_t hash(uint64_t x) {
```

```
        x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9ULL;
        x = (x ^ (x >> 27)) * 0x94d049bb133111ebULL;
        x = x ^ (x >> 31);
        return static_cast < size_t > (x);
      }
    };

    template < typename T >
    thread_local unsigned int scheduler < T > ::thread_id = 0;
```

I.1.5 C++ implementation of the SBLCWS algorithm with WShare (first version).

```
const int RACE = 1;
const int EMPTY = 2;
const int ABORT = 3;
const int FULL_EMPTY = 4;

const int FALSE = 0;
const int TRUE = 1;


const int HEURISTIC_SPREAD[3] = {FALSE,FALSE};
const int HEURISTIC_STEAL[4] = {TRUE,TRUE,TRUE};


const int WORKING = -1;
const int IDLE = -2;

template <typename Job>
struct Wrapper {
Job *job = nullptr;
int error = -1;
};

struct Deque {
      using qidx = int;
      using tag_t = int;

      // use std::atomic<age_t> for atomic access.
      // Note: Explicit alignment specifier required
```

```
// to ensure that Clang inlines atomic loads.
struct alignas(int64_t) age_t {
  tag_t tag;
  qidx top;
};

// align to avoid false sharing
struct alignas(64) padded_job {
  Job *job;
};

static constexpr int q_size = 1000;
qidx b_official;
qidx bot_private;
std::atomic < age_t > age;
std::array < padded_job, q_size > deq;

Deque(): b_official(0), bot_private(0), c1(0), c2(0), c3(0), c4(0), age(age_t {
  0,
  0
}) {}

void push_bottom(Job * job) {
  deq[bot_private++].job = job;

  if (bot_private == q_size) {
    throw std::runtime_error("internal error: scheduler queue overflow");
  }
}

Wrapper<Job> pop_top() {
  auto old_age = age.load(std::memory_order_relaxed);
  Wrapper<Job> wrap;

  if (b_official > old_age.top) {
    wrap.job = deq[old_age.top].job;
    auto new_age = old_age;
    new_age.top = new_age.top + 1;

    if (age.compare_exchange_strong(old_age, new_age, std::memory_order_relaxed, s
      return wrap;
```

```
    }

    wrap.job = nullptr;
    wrap.error = ABORT;
  }
  else if(b_official < bot_private)
      wrap.error = EMPTY;
    else
      wrap.error = FULL_EMPTY;

  return wrap;
}

void update_bottom() {
    if(b_official + 1 < bot_private)
      b_official++;
}

Job * pop() {
  if (bot_private == b_official)
    return nullptr;

  auto item = deq[--bot_private].job;
  return item;
}

Job * pop_bottom() {
  Job * result = nullptr;

  if(b_official != 0) {
    auto b = --b_official;

    std::atomic_thread_fence(std::memory_order_seq_cst);
    auto job = deq[b].job;
    auto old_age = age.load(std::memory_order_relaxed);

      if (b > old_age.top) {
        bot_private = b;
        result = job;

      } else {
```

```
                bot_private = 0;
                auto age_new = age_t {
                    old_age.tag + 1, 0
                };
                b_official = 0;

                if ((b == old_age.top) && age.compare_exchange_strong(old_age, age_new, s
                    result = job;
                } else {
                    age.store(age_new, std::memory_order_relaxed);
                    result = nullptr;
                }

                std::atomic_thread_fence(std::memory_order_seq_cst);
            }
          }

      return result;
    }

      bool check() {
          return b_official + 1 < bot_private;
      }
};

template < typename Job >
struct scheduler {
  public:
  unsigned int num_threads;

  static void signal_handler(int sig) {
    scheduler_p->spread();
  }

  bool finished() {
    return finished_flag.load(std::memory_order_relaxed);
  }

  static thread_local unsigned int thread_id;
```

```
scheduler(): num_threads(init_num_workers()),
num_deques(num_threads),
deques(num_deques),
attempts(num_deques),
spawned_threads(),
rand_devices(num_deques),
worker_state(num_deques, IDLE),
worker_heuristic(num_deques),
thread_handles(num_deques),
finished_flag(false) {

  static_scheduler = this;
  std::signal(SIGUSR1, scheduler<Job>::signal_handler);

  // Stopping condition
  auto finished = [this]() {
    return finished_flag.load(std::memory_order_relaxed);
  };

  for(int i = 0; i < num_deques; i++) {
    generator.push_back(std::mt19937(rand_devices[i]()));
  }

  thread_id = 0; // thread-local write

  thread_handles.emplace_back(pthread_self());
  for (unsigned long i = 1; i < num_threads; i++) {
    spawned_threads.emplace_back([ & , i, finished]() {
      thread_id = i;
      start(finished);
    });
    thread_handles.emplace_back(spawned_threads[i - 1].native_handle());
  }

  start(finished);
}

~scheduler() {
  finished_flag.store(true, std::memory_order_relaxed);
  for (unsigned int i = 1; i < num_threads; i++) {
    spawned_threads[i - 1].join();
```

```
  }
}

// Push onto local stack.
void spawn(Job * job) {
  int id = worker_id();
  deques[id].push_bottom(job);
}

// All scheduler threads quit after this is called.
void finish() {
  finished_flag.store(true, std::memory_order_relaxed);
}

// Pop from local stack.
Job * try_pop() {
  auto id = worker_id();
  auto job = deques[id].pop();
  if (!job)
    return deques[id].pop_bottom();

  if(worker_heuristic[id] == TRUE)
      spread();

  return job;
}

unsigned int init_num_workers() {
  if (const auto env_p = std::getenv("PARLAY_NUM_THREADS")) {
    return std::stoi(env_p);
  } else {
    return std::thread::hardware_concurrency();
  }
}

unsigned int worker_id() {
  return thread_id;
}


private:
```

```
    // Align to avoid false sharing.
    struct alignas(128) attempt {
      size_t val;
    };

  int num_deques;
  std::vector < Deque < Job >> deques;
  std::vector < attempt > attempts;
  std::vector < std::thread > spawned_threads;

  std::vector<int> worker_state;
  std::vector<int> worker_heuristic;

  std::vector<std::random_device> rand_devices;
  std::vector<std::mt19937> generator;
  std::uniform_int_distribution<int>  distr{0, 100};

  static scheduler<Job>* static_scheduler;


  template < typename F >
  void start(F finished) {
    size_t id = worker_id();
    while (true) {

      Job * job = get_job(finished, id);
      if (!job)
        break;

      ( * job)();
    }
  }

void update(int id, int new_state) {
    worker_heuristic[id] = new_state;
}

void spread() {
    auto id = worker_id();

    choice = distr(generator[id]) % num_deques;
```

```
        if(worker_state[choice] == IDLE) {
            deques[id].update_bottom();
            worker_state[choice] = id;
            update(id, HEURISTIC_SPREAD[0]);
        }
        else
            update(id, HEURISTIC_SPREAD[1]);

    }

      Job * try_steal(size_t id) {
        size_t target = (hash(id) + hash(attempts[id].val)) % num_deques;
        attempts[id].val++;

        auto wrap = deques[target].pop_top();
        auto error = wrap.error;

        choice = distr(generator[id]) % num_deques;

        if(worker_heuristic[choice] == FALSE && deques[choice].check())
        {
            worker_heuristic[choice] = TRUE;
            pthread_kill(thread_handles[choice], SIGUSR1);
        }


        if(job)
          worker_state[id] = WORKING;
        else {
          if(worker_state[id] != IDLE) {
            job = deques[worker_state[id]].pop_top().job;
            if(job)
              worker_state[id] = WORKING;
            else
              worker_state[id] = IDLE;
          }
        }

        return job;
      }
```

```
  template < typename F >
  Job * get_job(F finished, size_t id) {

    if (finished())
      return nullptr;
    Job * job = try_pop();
    if (job)
      return job;

    while (true) {
      for (int i = 0; i <= num_deques * 100; i++) {
        if (finished())
          return nullptr;

        job = try_steal(id);

        if (job)
          return job;
      }
      std::this_thread::sleep_for(std::chrono::nanoseconds(num_deques * 100));
    }
  }

  size_t hash(uint64_t x) {
    x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9ULL;
    x = (x ^ (x >> 27)) * 0x94d049bb133111ebULL;
    x = x ^ (x >> 31);
    return static_cast < size_t > (x);
  }
};

template < typename T >
thread_local unsigned int scheduler < T > ::thread_id = 0;

template < typename T >
scheduler<T>* scheduler<T>::static_scheduler = nullptr;
```

I.1.6 C++ implementation of the SBLCWS algorithm with WShare (second version).

```
const int RACE = 1;
const int EMPTY = 2;
const int ABORT = 3;
const int FULL_EMPTY = 4;

const int FALSE = 0;
const int TRUE = 1;


const int HEURISTIC_SPREAD[3] = {FALSE,FALSE};
const int HEURISTIC_STEAL[4] = {TRUE,TRUE,TRUE};


const int WORKING = -1;
const int IDLE = -2;

template <typename Job>
struct Wrapper {
Job *job = nullptr;
int error = -1;
};

struct Deque {
      using qidx = int;
      using tag_t = int;

      // use std::atomic<age_t> for atomic access.
      // Note: Explicit alignment specifier required
      // to ensure that Clang inlines atomic loads.
      struct alignas(int64_t) age_t {
        tag_t tag;
        qidx top;
      };

      // align to avoid false sharing
      struct alignas(64) padded_job {
        Job *job;
      };
```

```
static constexpr int q_size = 1000;
volatile qidx b_official;
volatile qidx bot_private;
std::atomic < age_t > age;
std::array < padded_job, q_size > deq;

Deque(): b_official(0), bot_private(0), c1(0), c2(0), c3(0), c4(0), age(age_t {
  0,
  0
}) {}

void push_bottom(Job * job) {
  deq[bot_private++].job = job;

  if (bot_private == q_size) {
    throw std::runtime_error("internal error: scheduler queue overflow");
  }
}

Wrapper<Job> pop_top() {
  auto old_age = age.load(std::memory_order_relaxed);
  Wrapper<Job> wrap;

  if (b_official > old_age.top) {
    wrap.job = deq[old_age.top].job;
    auto new_age = old_age;
    new_age.top = new_age.top + 1;

    if (age.compare_exchange_strong(old_age, new_age, std::memory_order_relaxed, s
      return wrap;
    }

    wrap.job = nullptr;
    wrap.error = ABORT;
  }
  else if(b_official < bot_private)
      wrap.error = EMPTY;
    else
      wrap.error = FULL_EMPTY;
```

```
  return wrap;
}

void update_bottom() {
    if(b_official < bot_private)
      b_official++;
}

Job * pop() {
  --bot_private;

  if (bot_private == b_official)
    return nullptr;

  auto item = deq[bot_private].job;
  return item;
}

Job * pop_bottom() {
  Job * result = nullptr;

  if(b_official != 0) {
    auto b = --b_official;

    std::atomic_thread_fence(std::memory_order_seq_cst);
    auto job = deq[b].job;
    auto old_age = age.load(std::memory_order_relaxed);

      if (b > old_age.top) {
        bot_private = b;
        result = job;

      } else {

          bot_private = 0;
          auto age_new = age_t {
              old_age.tag + 1, 0
          };
          b_official = 0;

          if ((b == old_age.top) && age.compare_exchange_strong(old_age, age_new, s
```

```
                result = job;
            } else {
                age.store(age_new, std::memory_order_relaxed);
                result = nullptr;
            }

            std::atomic_thread_fence(std::memory_order_seq_cst);
        }
      }
      else bot_private = 0;

    return result;
  }

    bool check() {
        return b_official < bot_private;
    }
};

template < typename Job >
struct scheduler {
  public:
  unsigned int num_threads;

  static void signal_handler(int sig) {
    scheduler_p->spread();
  }

  bool finished() {
    return finished_flag.load(std::memory_order_relaxed);
  }

  static thread_local unsigned int thread_id;

  scheduler(): num_threads(init_num_workers()),
  num_deques(num_threads),
  deques(num_deques),
  attempts(num_deques),
  spawned_threads(),
  rand_devices(num_deques),
  worker_state(num_deques, IDLE),
```

```
    worker_heuristic(num_deques),
    thread_handles(num_deques),
    finished_flag(false) {

      static_scheduler = this;
      std::signal(SIGUSR1, scheduler<Job>::signal_handler);

      // Stopping condition
      auto finished = [this]() {
        return finished_flag.load(std::memory_order_relaxed);
      };

      for(int i = 0; i < num_deques; i++) {
        generator.push_back(std::mt19937(rand_devices[i]()));
      }

      thread_id = 0; // thread-local write

      thread_handles.emplace_back(pthread_self());
      for (unsigned long i = 1; i < num_threads; i++) {
        spawned_threads.emplace_back([ & , i, finished]() {
          thread_id = i;
          start(finished);
        });
        thread_handles.emplace_back(spawned_threads[i - 1].native_handle());
      }

      start(finished);
    }

    ~scheduler() {
      finished_flag.store(true, std::memory_order_relaxed);
      for (unsigned int i = 1; i < num_threads; i++) {
        spawned_threads[i - 1].join();
      }
    }

    // Push onto local stack.
    void spawn(Job * job) {
      int id = worker_id();
      deques[id].push_bottom(job);
```

```
}

// All scheduler threads quit after this is called.
void finish() {
  finished_flag.store(true, std::memory_order_relaxed);
}

// Pop from local stack.
Job * try_pop() {
  auto id = worker_id();
  auto job = deques[id].pop();
  if (!job)
    return deques[id].pop_bottom();

  if(worker_heuristic[id] == TRUE)
      spread();

  return job;
}

unsigned int init_num_workers() {
  if (const auto env_p = std::getenv("PARLAY_NUM_THREADS")) {
    return std::stoi(env_p);
  } else {
    return std::thread::hardware_concurrency();
  }
}

unsigned int worker_id() {
  return thread_id;
}



private:
  // Align to avoid false sharing.
  struct alignas(128) attempt {
    size_t val;
  };

int num_deques;
std::vector < Deque < Job >> deques;
```

```
  std::vector < attempt > attempts;
  std::vector < std::thread > spawned_threads;

  std::vector<int> worker_state;
  std::vector<int> worker_heuristic;

  std::vector<std::random_device> rand_devices;
  std::vector<std::mt19937> generator;
  std::uniform_int_distribution<int>  distr{0, 100};

  static scheduler<Job>* static_scheduler;


  template < typename F >
  void start(F finished) {
    size_t id = worker_id();
    while (true) {

      Job * job = get_job(finished, id);
      if (!job)
        break;

      ( * job)();
    }
  }

void update(int id, int new_state) {
    worker_heuristic[id] = new_state;
}

void spread() {
    auto id = worker_id();

    choice = distr(generator[id]) % num_deques;

    if(worker_state[choice] == IDLE) {
        deques[id].update_bottom();
        worker_state[choice] = id;
        update(id, HEURISTIC_SPREAD[0]);
    }
    else
```

```
        update(id, HEURISTIC_SPREAD[1]);

}

  Job * try_steal(size_t id) {
    size_t target = (hash(id) + hash(attempts[id].val)) % num_deques;
    attempts[id].val++;

    auto wrap = deques[target].pop_top();
    auto error = wrap.error;

    choice = distr(generator[id]) % num_deques;

    if(worker_heuristic[choice] == FALSE && deques[choice].check())
    {
        worker_heuristic[choice] = TRUE;
        pthread_kill(thread_handles[choice], SIGUSR1);
    }


    if(job)
      worker_state[id] = WORKING;
    else {
      if(worker_state[id] != IDLE) {
        job = deques[worker_state[id]].pop_top().job;
        if(job)
          worker_state[id] = WORKING;
        else
          worker_state[id] = IDLE;
      }
    }

    return job;
  }


  template < typename F >
  Job * get_job(F finished, size_t id) {

    if (finished())
      return nullptr;
```

```
      Job * job = try_pop();
      if (job)
        return job;

      while (true) {
        for (int i = 0; i <= num_deques * 100; i++) {
          if (finished())
            return nullptr;

          job = try_steal(id);

          if (job)
            return job;
        }
        std::this_thread::sleep_for(std::chrono::nanoseconds(num_deques * 100));
      }
    }

    size_t hash(uint64_t x) {
      x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9ULL;
      x = (x ^ (x >> 27)) * 0x94d049bb133111ebULL;
      x = x ^ (x >> 31);
      return static_cast < size_t > (x);
    }
  };

  template < typename T >
  thread_local unsigned int scheduler < T > ::thread_id = 0;

  template < typename T >
  scheduler<T>* scheduler<T>::static_scheduler = nullptr;
```