Dynamic Loop Self-scheduling with Distributed Data for MPI Applications

Master Thesis

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
High Performance Parallel And Distributed Computing
https://hpc.dmi.unibas.ch

Advisor: Prof. Dr. Florina M. Ciorba
Supervisor: Jonas Henrique Müller Korndörfer

Nderim Shatri
nderim.shatri@stud.unibas.ch
HS16-062-234

31.08.2023

Acknowledgment

I am very grateful to have the opportunity to choose this path and to conclude my Master's studies as part of the HPC group at the University of Basel. I want to thank Prof. Dr. Florina M. Ciorba for her advice, guidance throughout the project, and the input she delivered. Always supportive and open to discussion, Prof. Dr. Florina M. Ciorba encouraged me throughout the Thesis. Also, I am grateful to have been supervised by Dr. Jonas Henrique Muller Korndorfer, who was continuously available for help, advice, and feedback. He was open to proposing a solution to issues, roadblocks, and bugs, even in the late hours. Additionally, I want to thank Mr. Gian-Andrea Wetten, who introduced the prior work of this Thesis. Furthermore, I want to thank my parents, Nazif and Safete Shatri, who were permanently supportive throughout the Thesis. Always being ready to help indirectly, they indeed encouraged me during this time. Last, I would like to thank my friends and colleagues for their support and open ears, even though I had to cancel some events and nights out.

Abstract

The development of supercomputers reaching exascale performance allowed computationally-intensive applications to execute immense workloads. Most significant scientific applications are executed in large HPC systems containing heterogeneous processors. They are ultimately leading to uneven performance progress among the parallel processing elements. The immense workloads are often affected by non-uniform memory accesses, algorithms, and idling processors, which lead to an imbalanced execution affecting a scientific application's performance. Various self-scheduling techniques have been proposed and developed to mitigate load imbalance and minimize scheduling overhead, as finding optimal schedules for parallel execution is NP-hard.

Applications sharing the property of being implemented with distributed data tend to handle the immense data required by dividing the data among processes. In most cases, communication among the processes is required, whereas scientists can benefit from the popular programming paradigm Message Passing Interface(MPI). A further approach to tackle load imbalance is work-stealing. In work-stealing, idling processes can be thieves who try to steal the work and data from slower processes, referred to as victims. This involves communication, which reduces load imbalance significantly with minimal scheduling overhead.

A library that can be integrated into MPI applications implemented with distributed data is the load balance for MPI(LB4MPI) library. In this work, we extended the work-stealing algorithm in LB4MPI to employ dynamic self-scheduling techniques for MPI applications. The work to be stolen relies on the scheduling technique used at a steal request. Further, we adapted and extended the random victim selection by two additional victim selection strategies, which bear the placement of the process into account.

The performance has been evaluated in four different applications, consisting of Pisolver, Mandelbrot, SPH-EXA(Sedov), and miniAMR. In Pisolver, we achieved a significant performance increase of up to 25.96%, depending on the configurable load imbalance factor. In Mandelbrot, LB4MPI shows a communication overhead leading to a performance decrease but can highlight the importance of having an immense workload to benefit the application's performance. The real scientific application SPH-EXA(Sedov) shows insignificant induced communication overhead. The miniAMR helped to detect the limitations of LB4MPI, which consists of extending the library by a work-borrowing method.

Table of Contents

1 Introduction

The demand for high-performing computer systems induced the development of supercomputers reaching exascale performance, allowing scientific parallel applications to conduct data-intensive computations. Most large scientific applications scale up to many processes where different factors may impact the implementation. Load imbalance, scheduling overheads, communication costs, and hardware inferences can negatively impact the performance of those applications. Computationally-intensive applications having idling processors lead to uneven execution, which can appear from facets of application, algorithm, and computer systems[1, 2]. Efficient dynamic scheduling and load balancing can mitigate load imbalance. Numerous scheduling techniques have been proposed over time[1, 3, 4], as finding optimal schedules is NP-hard[5]. To minimize load imbalance and scheduling overhead, dynamic self-scheduling addresses specific application- and system-induced performance deviations[6].
MPI[7] is a widely used parallel program paradigm to write parallel code by communication across processors. With MPI, one can facilitate the communication of applications within a distributed area but has to overcome the challenge of aligning a schedule.
Another approach to mitigate the negative factors in applications implemented with distributed data is the well-studied work-stealing algorithm by Blumofe et al[8]. Work stealing allows more performant processes to take work from a less performant process by dynamically communicating and receiving a fraction of the work and data. Hence, work stealing is a dynamic load balance technique that follows a policy where whenever a processor runs out of work, it tries to "steal" work from another processor. The induced communication cost should be lower than the remaining process execution time to be a beneficial load balance technique. By incorporating a simple yet effective victim selection strategy, the induced communication cost can be lower than the actual victim process would have taken. A random victim selection has been proven as a computationally cheap calculation[8, 9].
Another aspect of work-stealing is the amount of data and work that will be stolen by a process. A process might impact performance by stealing more work to induce a higher execution time than originally needed by the victim process. Hence, an efficient work-stealing algorithm depends on the induced communication cost, the amount of work and data to be stolen, and the computation to select a victim. Furthermore, executing distributed applications in high-scaled environments can cause degradations of balancing the loads in

work-stealing[10]. Thus, the performance can increase when processes can steal within their executing node.

LB4MPI[2, 11], Load Balancing for MPI, has been introduced to provide various dynamic and adaptive scheduling techniques as a library to integrate into scientific applications. Originally designed for applications with replicated data, a distributed approach to employ work-stealing has been added[11]. The work-stealing algorithm follows a coordinator-worker approach, using a random victim selection strategy. Each process is statically decomposed to perform the application's computation, whereas, at a steal, the work and data to be stolen rely on a manually defined "steal ratio".

In this work, we extended the LB4MPI coordinator-worker approach work-stealing with a new way to calculate the work that will be stolen by a worker. By benefitting from the available scheduling techniques in LB4MPI, we introduced dependence on the self-scheduling techniques in the calculated amount of work and data stolen. Furthermore, we provided two additional victim selection strategies where a process might only steal from the same node other processes execute. To benchmark the performance, we decided to integrate the modified LB4MPI in four applications: Pisolver, SPH-Exa (Sedov)[12, 13], Mandelbrot, and miniAMR[14].

In Pisolver, LB4MPI improved the performance by a factor ranging from 11.95% to 25.96% depending on the input parameters. In Mandelbrot and SPH-Exa (Sedov), the integration led to a higher induced communication cost which degraded the performance. However, those applications highlight the importance of choosing a suitable application that needs to be optimized in performance.

In the following sections, we provided the background on which the LB4MPI has been developed. Furthermore, we will discuss the related work which tackles previous work in dynamic self-scheduling techniques and work-stealing. Then, we will discuss the concept of the merged work-stealing algorithm with dynamic self-scheduling techniques. Afterward, we evaluated the performance in four different applications with distinct measurements. In the last two sections, we summarized the project and gave inputs for future work.

2 Background

New methodologies to enhance the performance of different applications have been developed and published over time. In the following, we present different variations of scheduling techniques that are commonly used. Each has further adaptions and improvements for either specific or generic problem sets used in different applications. Furthermore, a brief overview of work-stealing and LB4MPI is given.

2.1 Scheduling

The data may be centralized, replicated, or distributed in a distributed system. A centralized data approach follows the policy that data is located in a single processor, and other processors need to access the data by communicating with that location. A replicated data approach states that during execution, each processor working on a problem has its local replica of the data required. In a distributed data approach, the data is distributed among nodes. With non-uniform memory access (NUMA) Multiprocessors, the need for an efficient schedule is raised. The NUMA architecture states that each processor has local memory. In addition, a CPU can access the memory remotely, which is significantly slower than accessing a processor's local memory. Having data that lies in various processors requires data movement among the processors. Hence, one must encounter several issues to achieve an efficient schedule: memory access interference, random processor latencies, and others[3].
Scientific HPC applications solve a particular problem, e.g., Molecular Dynamics, which requires scheduling for enhanced execution performance. The principal source of optimizing execution time in parallel applications is loops. Loops may be regularly loaded or irregularly by having, for instance, a sparse matrix multiplication. For this reason, many propositions have emerged. We present static scheduling and dynamic scheduling methods in the following.

2.2 Static Scheduling

Static Scheduling algorithms share the property that the loop is divided into equal-sized chunks of data where iterations are executed in parallel per processing element. Such scheduling methods can be Block scheduling, cyclic scheduling, and a hybrid form of block-cyclic scheduling[15]. As the name suggests, block scheduling divides the loop into blocks or chunks of $[N/P]$ iterations, where N is the number of iterations and P is the number of processors. As we have multiple processors, each block is assigned to a processor. Cyclic scheduling assigns loop iterations to processors in a cyclic order. Hence, Processor p will execute the iterations p, $p+P$, $p+2P$, $\dots$, where P is the number of processors executing the loop. The hybrid form, block-cyclic scheduling, assigns blocks of a fixed size to processors, and if the block size shrinks to one, the execution will be cyclic scheduling. Static scheduling is well-suited for regular workloads. However, static algorithms do not consider the variation of execution time per iteration. This happens when computations do not require the same time for a chunk. Load imbalance may occur due to data load, different processor speeds, and communication time.

2.3 Dynamic Scheduling

Static scheduling methods and dynamic scheduling methods differ in the fact that static scheduling is employed before the execution starts. On the contrary, dynamic scheduling methods calculate their data chunks to allocate during execution time. Therefore, there is a higher risk that communication overhead is critical to reducing execution time.

2.3.1 Self-scheduling (SS)

SS is partitioning the loops into subtasks containing one or more iterations [3]. Processors allocate and execute one subtask at a time until no subtasks are left to process. Assigning one task at a time provides an optimal load balance. Contrary, the overhead is significant, as only one subtask is assigned. Fixed-size chunking is the procedure in which a subtask has a fixed number of iterations. The chunk sizes are calculated dynamically to reduce scheduling overhead as an improvement of SS. Problems occur with fine and coarse granularity of the work and data. Fine granularity might lead to overhead, and contrary coarse granularity might cause load imbalance.

2.3.2 Fixed Size Chunking(FSC)

FSC[16] improved the scheduling overhead of SS by scheduling chunks of a larger size. Chunks are added to a queue from which idling processing elements can take their chunks of iterations.

2.3.3 Guided Self-scheduling (GSS)

GSS [3]assigns several iterations to each processor. A parallel loop is supposed to be executed on p processors. With the assumption that each processor p starts executing some iterations of L at a different time, the goal is that processors finish at approximately the same time by assigning blocks of iterations at idle processors. Additionally, an incoming processor $p*$ should leave enough iterations to keep the remaining $p-1$ processors busy by sending several iterations x_i to them. Thus, each idle processor p_i operates: $x_i = [R_i/p]$; $R_i{+}1 = R_i{-}x_i$, and the range of iterations assigned to the $i{-}th$ processor is given by $[N{-}R_i{+}1, \ldots, N{-}R_i{+}x_i]$, where $R_1 = N$ and N is the total number of iterations. This leads to decreasing chunk sizes over execution time. The process leads to low overhead and load balancing in most cases.

2.3.4 Trapezoidal Self-scheduling (TSS)

TSS[17] is based on a linearly decreasing chunk function which aims to remove the overhead disadvantage of GSS. The chunks are calculated linearly by specifying the first chunk size f and the last chunk l, where it is suggested to set $f = N/2l$ while N the is the number of chores and depends on the number of total iterations$I$.

$$\begin{aligned}
\delta &= \frac{f-l}{N-1} \\
N &= \frac{2I}{f+l} \\
C(1) &= f \\
C(i) &= f-(i-1)\delta
\end{aligned} \tag{2.1}$$

2.3.5 Factoring (FAC)

Factoring has been proposed by [4] to mitigate the case in which GSS suffers from load imbalances. This case might occur when allocating too many iterations in early chunks to a processor that cannot finish the task by the time the other processors finish. Although FAC is similar to GSS, the main difference is that iterations are scheduled in probabilistic computed batches of P equal-size chunks. The total number of iterations per batch is a fixed ratio of those remaining. The chunk sizes are then decreased with finishing batches.

$$\begin{aligned}
b_j &= \frac{P}{2\sqrt{R_j}}\frac{\sigma}{\mu} \\
x_j &= 2 + b_j^2 + b_j\sqrt{b_j^2+4},\ j>0
\end{aligned} \tag{2.2}$$

As in 2.2 displayed, the chunk size x_j depends on the total processors P, iterations remaining R_j, and the coefficient of variance $[\sigma/\mu]$.
Depending on the variance of iterations, factoring degenerates into FSC when there is no variance, and with significant variance, it degenerates to SS. Factoring is considered the generalized version of FSC and GSS. However, probabilistic methods require knowledge about the processes' data and work distribution which is not always available.

2.3.6 Weighted Factoring (WF)

A modification of the factoring has been done by [18] and is an adaption to FAC. In GSS and FAC, large chunks are assigned in the beginning to smooth the variance among processors to achieve approximately the same finishing time. Contrary to FAC, WF has the rule of thumb to assign half of the remaining work in early batches. This heuristic has been evaluated experimentally. However, the main difference lies in including weights for different processor speeds. A fast processor eventually gets a larger batch to work on. The weights are estimated by benchmarking the system a priori and stay constant during execution.

2.3.7 Adaptive Self-scheduling

All listed dynamic scheduling techniques above, SS, FSC, TSS, GSS, FAC, and WF are non-adaptive. However, adaptive self-scheduling techniques attempt to reduce overhead by adapting to the system's performance. Among the adaptive self-scheduling techniques, there exists a Bold strategy[19] where the first chunk is calculated on a bolder approach. Chunk sizes are calculated based on the mean and standard deviation of the execution time, as well as estimates of the scheduling overhead. Apart from the Bold strategy[19], many proposed adaptive self-scheduling techniques were based on non-adaptive techniques. For instance, the adaptive weighted factoring (AWF)[6, 20] has been introduced to consider the cumulative performance during an application's previous computations apart from the weights in WF[18].

2.4 Work-stealing

Work stealing is the idea that underutilized processors attempt to "steal" work from working processors. This concept has already been introduced by Blumofe et al.[8], where underworked adjacent processors can steal from the process tree. Later works have been introduced, resulting in the popular approach: randomized work-stealing algorithm (RWS). Each processor maintains a ready deque data structure of threads. The ready deque has two ends, a top, and a bottom. This data structure allows threads to be inserted and removed on either end. Generally, a processor gets work by removing the thread at the bottom of its ready deque until it becomes empty. Then, the work-stealing can begin. A processor becomes a thief and attempts to steal work from a victim processor. The victim processor is chosen uniformly at random. If the victim processor has some work, the thief obtains the ready deque from the victim and begins to work on the top thread. The thief starts a new attempt if the victim processor has no work available.
As communications through processors are required, taking care of the communication overhead is crucial. Such a possibility is when multiple thieves try to steal work from victim processors. Another aspect that might be considerable is the amount of work that a thief might steal. However, over time variations and extensions of the work-stealing methods have been proposed [9, 11, 21].

2.4.1 Victim Selection

In work stealing, we define the processor who "steals" the work as the thief. The victim is the processor chosen by the thief. In most earlier works, the victim has been chosen randomly. However, taking the locality of a processor within a large HPC cluster might be beneficial for performance, as systems might be highly distributed in their memory approach. How to choose a victim processor is called victim selection. Another relevant aspect is the tasks that will be stolen, meaning the workload of the victim.

In a distributed memory environment, the processors employ a master-worker execution model[9, 11]. A global processor is then responsible for orchestrating task distribution among its workers. Considering the topologies and orchestrating work to processors with a low communication rate is beneficial for load balancing.
Instead of choosing a victim uniform randomly [8], weight and priority assignments can be done [22]. Each worker gets a priority to calculate a task, and larger priorities are assigned to larger tasks. Based on the priority, a thief can steal a large dividable task instead of a smaller task. Before a steal request starts, the workers are listed and chosen based on the priorities [22]. Analogous is the approach of assigning weights to the worker. With this, the thief sends only a steal request and can obtain larger tasks to work on.
The amount to steal from a victim is also crucial. Avoiding complex calculations from the work amount, pragmatic ways are chosen. In classical work-stealing [8], only one task is stolen at a time. However, there have been adaptions to take half of the tasks stolen from idle workers [21]. Another approach is to calculate the total load of each processor and select the victim with the highest workload.

2.4.2 LB4MPI

Load Balancing for(4) MPI is an MPI-based load balancing library that contains non-adaptive and adaptive dynamic loop self-scheduling techniques. With implementations in C and FORTRAN90, it is programmed to support scientific applications on HPC[2]. As LB4MPI supports fourteen scheduling techniques, the tool can be used for various scientific applications to encounter load balancing. Recent extensions initially supported load balancing through distributed work and data by a work-stealing algorithm[11]. The work-stealing algorithm employs a coordinator-worker approach. By defining a coordinator who is responsible for selecting a victim, a worker may send requests to steal work to the coordinator. By relaying the steal requests to a possible victim, the victim sends the work directly to the thief. At a steal, the amount of work to be stolen is relying on a static steal-ratio to steal an amount of work.
In this work, we extended the LB4MPI by allowing a process to steal an amount based on a self-scheduling technique. Furthermore, we extended and experimented with two new victim selection strategies that consider the locality of a node.

3 Related Work

Various works have been proposed as applications implemented with distributed data are arising due to memory limitations. Work-stealing has been a popular approach to engaging in distributed HPC systems and balancing workloads.
One strategy has been proposed by Acar et al.[23] and introduced Locality-guided work stealing. They studied the data locality of the work-stealing scheduling algorithm on hardware-controlled shared-memory systems. By defining lower and upper bounds of cache misses when using work-stealing, they introduced an adaption to the work-stealing algorithm. The adaptions organize an affinity that a process can have for work obtained. Hence, at a steal, the obtained work is prioritized to threads with an affinity for the work. Each process maintains a directed acyclic graph (dag) consisting of threads. Instead of working directly on threads, the algorithm operates on individual nodes in the computation dag. Execution is divided into time steps, such that a process either works on a node – assigned node - or tries to steal work at each time iteration. Based on the locality of nodes, the working-steal algorithm may perform relatively poorly. This can be when specific program applications that access the same data are not close in the computational graph. These types of applications have been called iterative data-parallel applications. A heuristic modification has been introduced as locality-guided work stealing to get a good locality for iterative data-parallel applications. Each thread can be given an affinity for a process. When a process obtains work, it prioritizes threads with an affinity for it. In addition to the deque, each process maintains a mailbox constructed as a FIFO queue consisting of pointers to the process-affine thread. There are then two differences between the locality-guided work-stealing and work-stealing algorithms. First, a process will push the thread onto the deque and the tail of the mailbox. In work-stealing, only the thread will be pushed onto the deque. Second, a process will try to obtain work from its mailbox before attempting a steal. This requires synchronization between the mailbox and deque, as threads might be there twice. So, the thread is executed once. With this structure, data locality is considered, and they achieved a performance increase of up to 50% under multiprocessors to traditional work-stealing algorithms.
Hierarchical Work Stealing [9] is designed to tackle heavy communication-intensive applications on a distributed platform. They balance the load between the thief and the victim

processor on the cluster level. This means a large amount of work can be stolen in a victim cluster. Therefore, the HWS needs information about the platform, divided into some processor groups connected with fast links. The restriction of processors that can steal from another group is required to limit the risk of congestion between groups. A chosen processor may send steal requests within a group and is considered the group's leader. Therefore, tasks are distributed by the group's leader, which usually contains much work. The user gives a limit by distinguishing tasks with a large amount of work. They can be global tasks that can be stolen between groups. Local tasks belong to a single group. The leader executes only global tasks, and they balance the load between groups and manage the load inside their groups. Furthermore, leaders decide if the task will be executed on a local stack or in a global task. The workers within a group execute only local tasks and perform the work-stealing algorithm within their group. With this approach, they increased a performance improvement of 20 percent of the execution time over standard algorithms used in a merge sort.
Chen et al.[24]. consider work-stealing within the modern multi-socket CPU architectures and overcome the issues of traditional work-stealing methods, which struggle with the accessibility of data from the fast local memory. Therefore, they implemented a locality-aware work-stealing (LAWS) algorithm. The locality-awareness is aimed at reducing the memory accesses in multi-socket CPU architectures. By having a load-balanced task allocator that equally splits and stores the data of a program for all memory nodes. The task allocator is also responsible for allocating tasks to sockets where the data is stored. After the allocation, the tree-shaped tasks are put into an execution graph. Applying the Divide and Conquer principle, the tasks are organized into cache-friendly sub-trees. This reduction of memory accesses led to a performance increase of up to 54.2% in AMD-based platforms or 48.6% on Intel-based platforms compared to classical work-stealing schedulers.
An evaluation on the eight thousand compute nodes scale has been done by [10]. They investigated the communication latencies on work stealing with new metrics to highlight issues in work-stealing algorithms executing on 8'192 MPI processes. They used deterministic work stealing by choosing a victim in a round-robin fashion. Using the deterministic work-stealing approach and comparing it with a modified version of the random work-stealing method showed that communication latencies matter explicitly when choosing a victim from a physical distance of the node. The modified version of random work stealing implies that one thief steals half of the victim's work because the thief has the remaining work when the processor becomes a victim.
Drebes et al. [25] considered work pushing to optimize data locality for benchmarks with asymmetric dependences. Work pushing is the transfer of tasks to workers executing on NUMA nodes containing the tasks' input data, which optimizes the read accesses. Combining topology awareness within the work-stealing algorithm could improve the performance of benchmarks running on NUMA nodes.
The N-body problem has been stated in physics as the prediction of individual motions of a group of masses interacting with each other by forces during time. N-body problems consist computationally of irregular loops and irregular data, which induce, among other things, load imbalance. Hence, Banicescu and Flynn[1, 26, 27] introduced Fractiling. Fractiling is

the combination of Factoring and Tiling. Factoring is the process wherein the size for each consecutive batch scheduled is half of the previous batch. Tiling is the process of assigning processing elements to an area. Each subtile has then an iteration space, and work is assigned in the form of decreasing batches. Work is allocated to tiles with fractal shuffled row-major numbering to spread the work evenly. When a processing element finishes its tiles, it can borrow loads from other slower processing elements and finish their tiles.
A dynamic load balancing and data migration library for performance improvement of scientific applications on parallel and distributed computing systems has been developed by [2] and is called the LB_Migrate library. LB_Migrate uses the coordinator-work execution model. The master processor is responsible for assigning chunks of tasks to all the worker processors. Furthermore, the master processor needs to track the execution performance of the worker processors based on their task completion time. Then, the master processor can distribute data from one worker to another whenever there is a computational load imbalance. However, all processors execute work and ask the coordinator processor for more work when nearly finished. The master processor decides, based on the information obtained, if he gives his iterations or requests a slower worker to migrate tasks. The LB_migrate has various DLS techniques and can use only one technique at a time. With the recent extension of Wetten [11] of the LB4MPI library, it is possible to allow scheduling with distributed data. His approach combines Work-Stealing based on a coordinator-worker method to allow scheduling with distributed data. The victim selection strategy employed uses a random victim selection. However, when a thief processor runs out of work, he can only obtain a fixed work ratio. The fixed steal ratio is a static value decided by the user.
In this work, we provided a further extension to the LB4MPI library. The LB4MPI library allows balancing MPI applications implemented with a distributed data approach by employing a work-stealing algorithm. The work-stealing algorithm is based on a coordinator-worker approach where various dynamic scheduling techniques[2, 11] can be chosen. In this approach, the coordinator handles incoming steal requests by choosing a victim. Furthermore, the steal amount of the data can be defined by the chunk calculation of the scheduling technique. Last but not least, we introduced two new victim selection strategies. The first is a locality-aware victim selection strategy where steal requests may only be in processes executing within the same node by inducing intra-node communication. The second strategy is a naive locality-aware approach where the coordinator chooses a victim based on the coordinator's guess that does not include induced communication.

4 Methods

4.1 Implementation of Work stealing in LB4MPI

This section presents the implementation of work stealing in LB4MPI. We will first show how the framework applies the coordinator-worker approach by introducing our concept of work-stealing. Then, we will dive into a more detailed insight into the roles of the coordinator and workers. At the end of this chapter, we will give an overview of the workflow within LB4MPI.

4.1.1 Scheduling techniques in LB4MPI

Applications can be implemented with replicated, centralized, or distributed data. Replicated data refers to parallel processes having a copy of the original data in their memory. Centralized data is the concept where one process keeps the data, whereas the data is accessed by other processes executing. Applications implemented with a distributed approach follow the principle that data is partitioned for each process which allows to use of less memory per process. Initially, the LB4MPI library provided various scheduling techniques for applications using replicated data. With the recent extension, those scheduling techniques have been adapted for applications implemented with distributed data. By giving a parameter to the LB4MPI library, various techniques are deployed. Each process performs a computation to calculate its chunks based on one of the techniques in the following table 4.1. The IDs of the scheduling techniques also correspond to the environment variable value for the scheduling technique in LB4MPI.

LB4MPI_SCHEDULING		
Scheduling techniqe	Abbreviation	id
STATIC	static	0
Self-scheduling	SS	1
Fixed size chunking	FSC	2
Modified FSC	mFSC	3
Guided Self-scheduling	GSS	4
Trapezoidal Self scheduling	TSS	5
Factoring	FAC	7
Weighted Factoring	wFAC	8
Adaptive weighted Factoring	AWF	9
Batch AWF	AWF-B	10
Chunk AWF	AWF-C	11
Batch AWF (chunk times)	AWF-D	12
Chunk AWF (chunk times)	AWF-E	13
Adaptive Factoring	AF	14
Simulation Assisted	SimAS	15

Table 4.1: Scheduling techniques in LB4MPI

4.1.2 Concept of Work stealing in LB4MPI

The idea of LB4MPI is to use the library as an API for applications that work with distributed data and work in a distributed environment. How the data is organized depends on the application. Hence, LB4MPI cannot do data-partitioning of an application's data. The work-stealing in LB4MPI follows a coordinator-worker approach. The coordinator, usually the rank with the ID 0, will handle steal requests from workers and forward those steal requests to other workers. The worker might become either a thief or a victim.

The illustration in Figure 4.1 shows a successful and unsuccessful steal request in LB4MPI. Initially, the data was equally distributed among the processes. The work itself is divided into chunks that need to be calculated. Hence, the chunk sizes vary. Each process works on its work and fulfills the calculation of an application. At the point where one process finishes its work, it will try to ask for more work. The worker with rank ID 3 becomes a thief and asks the coordinator for work by sending a STL-request. The coordinator relays the STL-request to the worker with the rank ID 1. As the worker with the rank ID 1 is already working on its last chunk of data, the worker with the rank ID 1 rejects the steal request by sending a REJ-tag. After receiving the REJ-tag, the coordinator searches for another possible victim. Hence, he relays the STL-tag to another worker. The worker will look at the data and recognize that the work is unfinished. Accepting the STL-request makes the worker a victim with the rank ID 2. The victim will send its next unstarted chunk of work to the thief with the rank ID 3. The thief will obtain the work directly from the victim and work on the data chunk.

With this basic concept, we can crystallize three roles with different responsibilities. The coordinator is responsible for handling the steal requests and relaying the steal request to a worker by applying a victim selection strategy. Therefore, the coordinator performs a type of victim selection strategy. The worker might be a victim who either rejects the steal request or calculates the next chunk to be sent to the thief. Hence, the victim is responsible for calculating the next chunk sent to the thief. The thief sends the steal request after finishing his work and must ensure receiving the complete data at a successful steal that goes with the chunk.

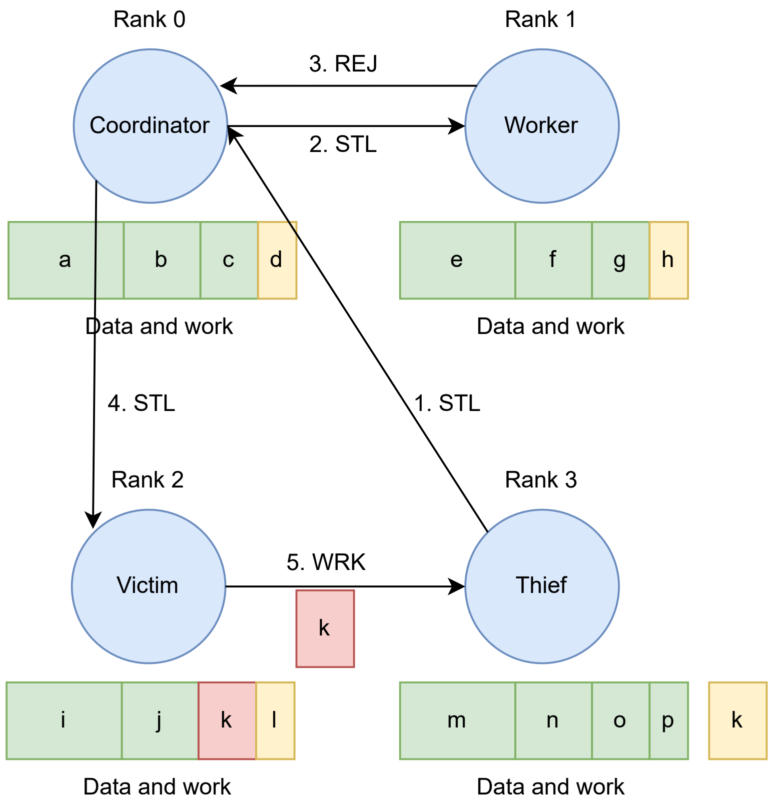


Figure 4.1: **Work stealing in LB4MPI - rejecting steal request and successful steal request:** In green, we can see the data chunks already calculated by the employed scheduling techniques. The work currently being worked on in yellow and red shows the work that will be sent.

4.1.3 Coordinator

The coordinator maintains a queue where incoming requests are tagged and inserted into the queue. The tags might be of type STL_Tag which indicates a steal request, a REJ_Tag which means a reject message obtained by a worker, or a TRM_Tag which marks the end of execution of a worker. Using a coordinator-worker approach, the coordinator cannot steal to prevent deadlocks. As the coordinator is responsible for relaying the steal request to the possible victim, it must follow a policy by choosing a victim. In LB4MPI, we have three different approaches to selecting a victim. The first is the random victim selection, the second is a locality-aware victim selection, and the last is a naive locality-aware victim selection.

4.1.3.1 Random victim selection

To select a victim randomly in LB4MPI, we maintain a list of which possible victims might be chosen from. The list in listing 4.1 is created and distributed among the workers so that each worker obtains a list of the available victims.

```c
info->availableVictim = (int *) malloc(sizeof(int) * info->lastRank);
for (worker = info->firstRank; worker <= info->lastRank; worker++) {
    info->availableVictim[worker] = (worker != info->myRank);
}
```

Listing 4.1: Inizialization of list with available victims

At some point during the execution, the coordinator will receive a steal request by an $MPI\_Recv()$ as listed in 4.2. The tag might either be of the type STL_TAG obtained by a thief or REJ_TAG received by a rejecting worker.

```c
MPI_Recv(chunkInfo, 2, MPI_LONG, mStatus.MPI_SOURCE, STL_TAG, info->comm, &
    tStatus);

// OR

MPI_Recv(chunkInfo, 2, MPI_LONG, mStatus.MPI_SOURCE, REJ_TAG, info->comm, &
    tStatus);
```

Listing 4.2: Coordinator receives either STL_Tag or REJ_Tag

However, the resulting step in both tags is the same: the coordinator randomly chooses a victim. In 4.3, the $availableVictim$ list has been trimmed to choose victims who might be all other victims except for the coordinator or the thief. We can see in line 21 that the method get_random_victim is called to choose the victim. The method get_random_victim requires the sum of potential victims in $get\_sum\_potential\_victims$. Based on this sum, the weights are added to the variable vis and looked if it fits the randomly chosen $victimInd$. Ultimately, the victim is chosen, and the coordinator sends a steal request to the victim.

```c
 1  int get_sum_potential_victims(int *weights, int size) {
 2      int sum = 0;
 3      for (int i = 0; i < size-1; i++)
 4          sum += weights[i];
 5      return sum;
 6  }
 7
 8  int get_random_victim(int *weights, int size) {
 9      int sum = get_sum_potential_victims(weights, size);
10      int victimInd = rand() % sum + 1;
11      int victim = 0;
12      int vis = 0;
13      while (1) {
14          vis += weights[victim];
15          if (vis == victimInd)
16              break;
17          victim++;
18      }
```

```
        return victim;
    }
    victim = get_random_victim(info->availableVictim, info->lastRank);
    MPI_Send(chunkInfo, 1, MPI_LONG, victim, STL_TAG, info->comm);
```

Listing 4.3: Random victim selection in LB4MPI

4.1.3.2 Locality-Aware Work Stealing

Although random victim selection has been shown in [8] as a reasonable way of selecting a victim, we extended our library with a further approach. Locality-aware work stealing follows the idea that processes executing on nodes in a wide-ranged HPC system are limited to stealing only within the node. Intra-node communication is faster than inter-node communication due to a node's hardware. As multiple processes can be executed within a node, we try to reduce the work-stealing's communication latencies by allowing processes to steal only within a node containing multiple processes. However, in distributed MPI applications, we have the drawback that information between processes is not set by default. This means that a coordinator cannot know where the workers are executing. So, the coordinator needs to be informed and needs to store the locality of a process to perform the victim selection. This approach involves communication so that a process can inform the coordinator which node it executes. Assuming that each node of an HPC system has a unique name, we benefitted from the MPI function of *MPI_Get_processor_name*. This will return the node name where a rank is executing. The code issuing the communication of the node names is listed in 4.4. The workers know the node they are executing by getting the processor name. Then, they need to inform the coordinator where they are executing by sending their node name to the coordinator. The coordinator maintains a list of the node names where the indices represent the rank ID of the worker.

```
    MPI_Get_processor_name(info->node_name, &name_len);
    if (info->comm == MPI_COMM_NULL){
            MPI_Send(info->node_name, NODENAMESIZE, MPI_CHAR, 0, 0, MPI_COMM_WORLD
                );
    } else {
        for (int i = 1; i < info->commSize; i++) {
                char nodename[50];
                MPI_Recv(nodename, NODENAMESIZE, MPI_CHAR, i, 0, MPI_COMM_WORLD, &
                    tStatus);
                strcpy(info->all_node_names[i], nodename);
        }
    }
```

Listing 4.4: Coordinator receives the node name and worker send the node name

So, the list is now stored on the coordinator's side and contains all information of processes executing in the nodes. By mapping the thief's node name, the coordinator knows then where the processes are executing. Based on the list, the coordinator randomly chooses a victim operating on the same node as the thief. The function to map is represented in the listing 4.5

```c
long get_same_nodes(infoDLS *info, long* same_node_ranks, MPI_Status mStatus)
    {
    long same_node_count = 0;
    // Loop to map the node names to the thief where his id is stored in
        mStatus.MPI_SOURCE
    for (int i = 0; i < info->commSize; ++i) {
        if (strcmp(info->all_node_names[mStatus.MPI_SOURCE], info->
            all_node_names[i]) == 0 && i != mStatus.MPI_SOURCE) {
            same_node_ranks[same_node_count++] = i;
        }
    }
    return same_node_count
}
// same_node_ranks contains now the ids of the available processes in the node
     of thief
victim = (long) same_node_ranks[rand() % same_node_count];
```

Listing 4.5: Coordinator maps the list containing the node names to same ranks as the thief and returns the list

We use only one coordinator responsible for handling steal requests, so the cross-node communication latencies cannot be avoided entirely. However, through this approach, we aimed to reduce the possible latencies in sending data and work from the victim to the thief. Due to hardware restrictions, inter-node communication is less performant than intra-node communication. Hence, we assumed that sending data within a node might reduce the communication time performing a process-to-process data transfer.

4.1.3.3 Naive locality-aware Work stealing

As locality-aware work stealing needs additional communication and, therefore, additional time to communicate, we included an instead naive way to perform locality-aware work stealing. So, to reduce this communication latency and take care not to involve heavy computations to select a victim, we adapted the random victim selection in the following way: Under the assumption that an HPC system employs a space-filling-curve (SFC) [28], such that it follows a linear order that processes are executing. The communication size is the sole known condition by every process. Hence, the coordinator has no prior knowledge about the environment in which the program is executing. A coordinator will estimate which node a worker is executing on at an incoming steal request. By getting the rank ID of the thief, the coordinator will choose among half of the processes to select a victim randomly. We let the coordinator divide the communication group size by two, as this will work in a small execution containing two processes per node. Hence, dividing the communication group size by two will lead to a margin containing half of the processes.
An example of a possible execution can be seen in 4.2. We can see an application executing on four nodes with six processes each. This yields 24 processes that an application uses. So, the sole number that is known by all processes, including the coordinator, is the group size

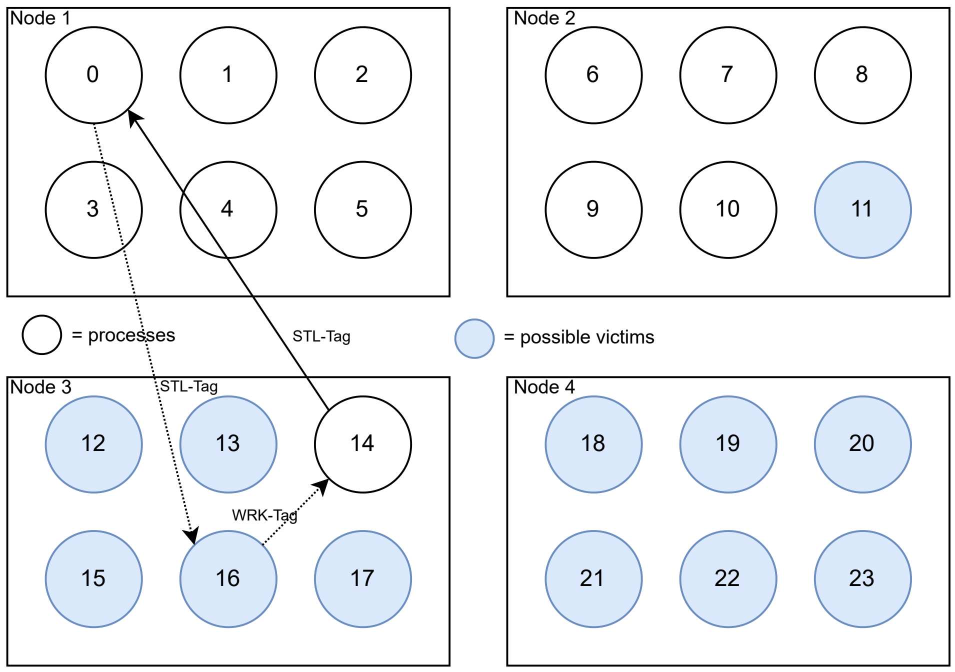


Figure 4.2: **Example of Naive locality-aware work stealing:** The dotted arrows represent the possible successful steal request from the coordinator to the victim.

of 24 in this case. Consider a steal request from the process with the rank ID 12. It will communicate to the coordinator with the rank ID 0 that it requests work. The coordinator must estimate based on the rank ID 12 on which node it executes. As the coordinator knows the group size of the communicator, he will do the following: The coordinator will divide the group size by 2, yielding $size/2$ and, in this example, 14. If the $size/2$ exceeds the rank ID 14, it will fill its possible victim list with values up until $size$. As this might not get all the possible victims, the coordinator will fill the remaining list with values lower than the rank ID 14 until the list contains $size/2$ elements. Otherwise, the coordinator will fill the list with values lower than the rank ID received. As this also does not get all the possible victims, the coordinator will fill the remaining list with values higher than the rank ID 14 until the list contains $size/2$ elements. In this case, the coordinator creates a list of possible victims containing these values: $list = [15, 16, 17, 18, 19, 20, 21, 22, 23, 13, 12, 11]$. Among this list, the coordinator chooses a victim randomly to select by chance a victim that is running on the same node.

The estimation relies solely on the rank ID and the MPI_Comm_size, which determines the group size of a communicator. As we are always taking half of the group size of a communicator, a coordinator might choose a victim that operates on the same node as the thief. Furthermore, this yields a wide margin by using half of the group size. However, the margin can be set finer by using a higher number to divide the MPI_Comm_size In a distributed system without communication, gathering all the executing nodes for one process is inconceivable with MPI. Hence, we extended to have a hybrid version between the random victim selection strategy and the locality-aware work stealing that does not involve communication and a loaded computation.

4.1.4 Worker

Each worker might become a thief who requests data from the coordinator. On the other hand, each worker might also become a victim who either rejects the steal request or accepts it and sends its next chunk to the thief. How much data a worker should send and receive is based on the chunk calculation based on the scheduling techniques employed in LB4MPI. Thus, the amount of data to steal or to send depends on the outcome of the scheduling technique used during execution. How the data is sent and received is an essential task that the thief and the victim must overcome.

4.1.5 Data type handling in LB4MPI

HPC applications vary in operating data types to perform calculations. LB4MPI is not directly operating on the data and is not intended to pursue data modifications. However, LB4MPI takes the pointers to the data to communicate them to the processors. Furthermore, it sets the indices of the computation that an application is performing. Nevertheless, it should be able to be integrated with various data structures, such as one-dimensional arrays, cubic arrays, and complex objects.

4.1.5.1 One-dimensional data type

Assuming an application uses a one-dimensional array with the primitive data type long where each index of the array represents a different workload. LB4MPI employs a scheduling technique responsible for creating chunks of the array and returning the calculated indices of the array to the application. At a steal request, this is straightforward. The victim will first calculate the chunk to send. Then, it will communicate the chunk to the thief using a call with MPI_Send. As listed in 4.6, we can see the communication made by the victim. Alongside the WRK_TAG, the variable to_steal and the variable $info \rightarrow data.array$ are sent. The variable $info \rightarrow data.array$ represents a pointer pointing to the data, and to_steal contains the chunk size calculated by the employed scheduling technique.

```
    MPI_Send(info->data.array, to_steal, info->data.oldtype, chunkInfo[0],
                                    WRK_TAG, info->crew);
```

Listing 4.6: Victim sends chunk to the thief

The thief will wait for the blocking to send and receive the chunk calculated in the following way (4.7). Obtaining the WRK_TAG, the thief might continue on this newly received data.

```
    MPI_Recv(info->data.array, number_amount, info->data.oldtype, mStatus.
        MPI_SOURCE, WRK_TAG,
                                    info->crew, &tStatus);
```

Listing 4.7: Thief receives chunk of data from the victim

4.1.5.2 Cubic data type

Such a cubic array, sending a more complex data type, is more complicated to achieve with MPI. Let us consider a cubic data array as in 4.3.

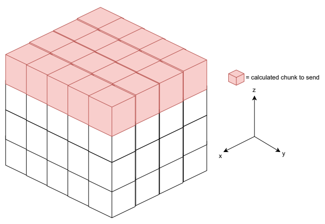


Figure 4.3: Chunk to send to the thief in a cube

To send this chunk with MPI, we need the following: First, we need to know how large the data is in the dimensions x, y, and z, which are obtained by the application and parsed to LB4MPI. Secondly, MPI provides a method named $MPI\_Type\_create\_subarray()$. Essentially, this method will create a new derived $MPI\_Datatype$ that knows how the cube is structured by providing the dimensions. Then, we must commit this newly created datatype and prepare it to send. By performing the computation on the x-dimension, we will always send a cube slice as listed in 4.8.

```
void set_cubic_DataType(infoDLS *info, int chunk_size) {
    int sizes[3] = {chunk_size, info->cubic_data.dim_y, info->cubic_data.dim_z};
    int subsizes[3] = {chunk_size, info->cubic_data.dim_y, info->cubic_data.dim_z};
    int starts[3] = {0, 0, 0};
    MPI_Type_create_subarray(3, sizes, subsizes, starts, MPI_ORDER_C, info->cubic_data.oldtype, &info->cubic_data.newtype);
    MPI_Type_commit(&(info->cubic_data.newtype));
}

set_cubic_DataType(info, to_steal);
```

Listing 4.8: Preparing a cubic slice by creating a new data type

We parse the cubic array's first index to send and receive the pointer pointing to the array. The newly created $MPI\_Datatype$ is then responsible for setting the correct indexes.

```
// Victim side
set_cubic_DataType(info, to_steal);
MPI_Send(&(info->cubic_data.cubic_array[0][0][0]), 1, info->cubic_data.newtype, chunkInfo[0], WRK_TAG, info->crew);
// Thief side
set_cubic_DataType(info, to_steal);
MPI_Recv(&(info->cubic_data.cubic_array[0][0][0]), 1, info->cubic_data.newtype, mStatus.MPI_SOURCE, WRK_TAG,
                                  info->crew, &tStatus);
```

Listing 4.9: Three-dimensional send and receive

With this approach, we managed to send and receive a slice of a cubic data structure. However, this depends on the application because the cubic data needs to be contiguous in space. Having a sparse cube, unpredicted behavior might happen. Hence, we have the concept of serialization and deserialization.

4.1.6 Deserialization and Serialization

As LB4MPI is written in the programming language C and, unlike $C++$, C does not provide objects. Therefore, serialization on the application side is needed before using LB4MPI. By creating function pointers that can be parsed to LB4MPI, we can indirectly influence the behavior in the application and perform work stealing on the serialized task. For instance, let us look at a $C++$ object called Node which the application used to perform computations. First, we need to create a serialization function that takes the Node and returns a string with the help of the *ostringstream* and *archive* from boost(4.10.

```cpp
std::string serializeNode(const Node &t) {
  // creates a string stream ss
  std::ostringstream ss;
  // boost to serialize string stream ss
  boost::archive::text_oarchive oa{ss};
  oa << t;
  return ss.str();
}
```

Listing 4.10: Serialization on the application

Second, we create a function pointer on the application side that can be parsed to LB4MPI.

```cpp
void dMap(void* in, int start, int end, MPI_Datatype* out_type,void** out, int
    * len) {
  Node *t = static_cast<Node *>(in);
  std::string ts= serializeNode(*t);
  *out_type = MPI_BYTE;
  std::string *out_str = new std::string(std::move(ts));
  *out = static_cast<void*>(const_cast<char*>(out_str->data()));
  *len = out_str->size();
}
```

Listing 4.11: Function pointer for serialization on the application

Third, we create the counterparts, which include the deserialization and the function pointer to call later on the deserialization, as illustrated in listing 4.12.

```cpp
Node deserializeNode(const std::string &in) {
  // input will be streamed in ss
  std::istringstream ss(in);
  // input will be deserialized through boost
  boost::archive::text_iarchive ia{ss};
  Task obj;
  ia >> obj;
  return obj;
}
```

```

void nMap(void* in, int start, int end, MPI_Datatype* out_type, void** out,
    int* len) {
  const std::string tstring(static_cast<char*>(in),*len);
  Node a = deserializeNode(tstring);
}
```

Listing 4.12: Deserialization on the application

In listing 4.11 and 4.12, we encounter *dMap* and *nMap*. These function pointers can be parsed to *DLS_Data_Setup*, which sets the data for LB4MPI.

```
void (*ser_pointer)(void*, int, int, MPI_Datatype*, void**, int*) = &dMap;
void (*deser_pointer)(void*, int, int, MPI_Datatype*, void**, int*) = &nMap;
DLS_DataSetup ( &iInfo, reinterpret_cast<void*>(&nodeList[0]), NULL, MPI_BYTE,
     ser_pointer, deser_pointer);
```

Listing 4.13: Calling LB4MPI function DLS_DataSetup

Later, the victim calls the function pointer to serialize the data at a successful steal request. This serialization call will then serialize the chunk calculated to steal. On the other hand, the thief will receive the serialized data but needs to deserialize it such that the thief works then on the stolen chunk.

4.2 Integration of LB4MPI

LB4MPI is designed to benefit distributed applications by allowing work stealing for various data types. In this section, we will go through the workflow of LB4MPI, which finally leads to successful integration in the application.

4.2.1 Workflow of the integration

In 4.4, we can see the integration workflow an application needs to use LB4MPI successfully. Starting on the application side, the factor data will come into play instantly. Depending on the data, one must ensure that the data might need serialization for the library written in C. However, the first call to LB4MPI is *DLS_Parameters_Setup* which requires various parameters that the library needs. Then, a decision must be made if the data has a three-dimensional form or belongs to a one-dimensional array of a primitive data type, leading to choosing either *DLS_CubicDataSetup* or *DLS_DataSetup*. As LB4MPI aims to be wrapped around heavy-loaded code blocks in the application, the *DLS_StartLoop* will initiate the scheduling right before the while loop. After that, the *DLS_StartChunk* is responsible for the work-stealing algorithm and chunking of the data. At a successful steal, *GetChunkSizeWS* is called within the library. Both latter calls are needed to send the newly indexed chunk back to the application where the computation occurs. *DLS_EndChunk* signalizes the LB4MPI that the Chunk has been finished. As the integration happens during a continuous while loop, a check is needed if there is further data to work on. Hence, *DLS_Terminated*? will check if processes work on the data. Last but not least, *DLS_Endloop* is called, and LB4MPI will finish its scheduling and check if results need to be printed based on the user's input.

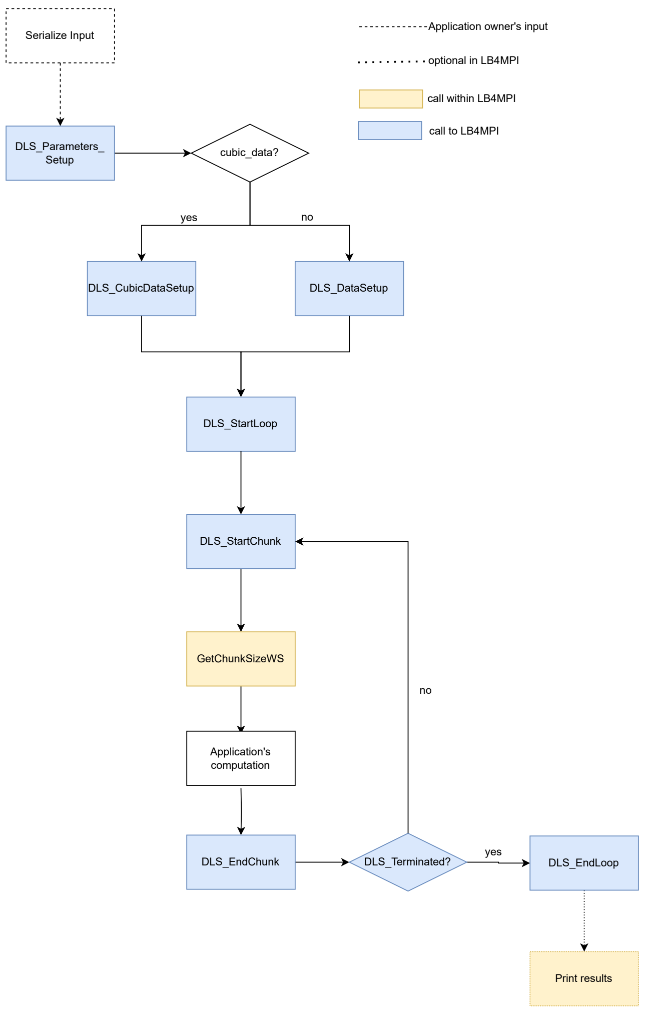


Figure 4.4: Implementation workflow of LB4MPI

4.2.2 Configuration

An important aspect of our work is to make LB4MPI highly configurable. We added different environment variables by allowing various scheduling techniques and victim selection strategies. In table 4.2, we can see the possible environment variables that can be parsed to LB4MPI. For instance, in the variable LB4MPI_ITERATIONS_INFO, we provided the possibility to print data gathered during the execution, including the number of iterations completed and chunk sizes scheduled per process and per total execution. As various print statements can affect the performance of LB4MPI, the environment variable LB4MPI_ITERATIONS_INFO is mutable.

Configuration of the environment variables		
Environment variable	Value	Description
LB4MPI_STEAL_RATIO	0-100	Fraction of the workload that is going to be stolen
LB4MPI_SCHEDULING	0-14	Sets the DLS (Static, SS, FSC, mFSC, GSS..)
LB4MPI_WS_SCHEDULING	0-14	Steal amount calculated on DLS
LB4MPI_LAWS	0, 1	Activates locality-aware work stealing
LB4MPI_nLAWS	0, 1	Activates naive locality-aware work stealing
LB4MPI_ITERATIONS_INFO	0, 1	Activates the possible .csv output files
LB4MPI_DIR_ID	string	Sets the name of the output directory

Table 4.2: Environment variables to set up different possibilities in LB4MPI

4.2.3 Parameters

We provide a library with various options that require different input from the application. In listing 4.14, we can see input parameters *DLS_Parameters_Setup* requires. The customized struct *infoDLS* will be set according to the parsed parameters where they can set the adaptive scheduling techniques, providing that work stealing should be enabled and the wished output name. By setting all the parameters, we gather the general conditions to use LB4MPI.

```c
void DLS_Parameters_Setup(MPI_Comm icomm, infoDLS *info, int numProcs, int
    requestWhen, int breakAfter, int minChunk, double h_overhead, double sigma
    , int nKNL, double Xeon_speed, double KNL_speed, int workStealing, char *
    info_name)
```

Listing 4.14: DLS_Parameters_Setup

4.2.4 Data setup

LB4MPI is capable of dealing with one-dimensional and three-dimensional data of primitives. The one-dimensional case includes serialized objects when they have been transformed to a char array. Hence, the *DLS_DataSetup* will set up the struct *DataElement* with the function pointers shown in listing 4.11. The *DataElement* is part of the struct of *infoDLS*, which provides the library with all information.

```c
void DLS_DataSetup(infoDLS *info, void *data, void *results, MPI_Datatype
    oldtype, fMapPtr dMap, fMapPtr nMap) {
DataElement dt;
dt.array = data;
dt.oldtype = oldtype;
dt.DLS_IterMap = dMap;
dt.DLS_DeSer = nMap;
dt.results = results;
info->data = dt;
info->n_dim = 0;
```

Listing 4.15: DLS_Parameters_Setup

Similarly to the one-dimensional in 4.15 case, the cubic data array needs a setup 4.16. However, additional information, such as the dimensions, is needed to initialize the new data type.

```
void DLS_DataSetup_Cubic(infoDLS *info, double ***data, MPI_Datatype oldtype,
    int dim_x, int dim_y, int dim_z) {
    CubicDataElement cdt;
    cdt.cubic_array = data;
    cdt.dim_x = dim_x;
    cdt.dim_y = dim_y;
    cdt.dim_z = dim_z;
    cdt.oldtype = oldtype;
    info->cubic_data = cdt;
```

Listing 4.16: DLS_Parameters_Setup

4.2.5 Start loop

After initializing the required data, the *DLS_StartLoop* is called. Here, the first and last iteration indices are given to let the library know how many iterations need to be scheduled. It is also possible to give the requested scheduling technique if it has originally not been passed over the environment variable. Furthermore, it sets the various environment variables or provides default values if some variables are not defined. Then, the *DLS_StartLoop* continues to set up the initial chunk sizes for every process involved. As the name suggests, this function is called at the beginning of the loop.

4.2.6 Chunk calculation

Within the loop, the *DLS_ChunkCalculationWS* should be called. The work-stealing algorithm operates from this method. This will ultimately set the start and end indices of the chunks to calculate and return to the application. As listed in 4.17, the processes probe with $MPI\_IProbe()$ for messages within their message queue. The coordinator looks for the three tags: STL-, REJ-, and TRM-tag. If there is a STL-tag or a REJ-tag, the coordinator chooses a victim by the victim selection strategy. Otherwise, the coordinator receives a TRM-tag to be informed that a process has finished all the iterations. The worker checks for STL- and WRK-tags. If the worker receives a STL-tag from the coordinator, the worker checks if it has remaining work. Then, it either sends a REJ-tag back to the coordinator or accepts the STL-tag and sends its work to the requesting worker. Otherwise, a worker can receive a WRK-tag which shows that a steal request has been successful and work will be obtained.

```
MPI_Iprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, info->crew, &MsgInQueue, &mStatus);
while (MsgInQueue) {
    if worker {
        case (STL_TAG):
            if remaining_work {
                send_work();
            } else {
                    // Reject Steal request
                    send_rej();
                }
            case (WRK_TAG):
                receive_work();
            case (TRM_TAG):
                // finished all iterations, no work left
                return;
        } else { // Coordinator
            case REJ_TAG:
                // relay STL_Tag to new victim
                select_new_victim_by_strategy()
            case STL_TAG:
                // relay STL_Tag to victim
                select_victim_by_strategy()
            case TRM_TAG:
                return;
        }
    }
```

Listing 4.17: DLS_ChunkCalculationWS - Pseudo Code

4.2.7 End loop

When all iterations are scheduled and the computation of the parallelized loop is finished, the execution of LB4MPI ends. At this point, information about the performance of the total execution is obtained. So, the application is then decoupled from the use of LB4MPI.

5 Results

The following will present the results obtained by integrating LB4MPI into the four applications, Pisolver, Mandelbrot, MiniAMR[14], and SPH-Exa (Sedov)[12]. All the experiments have been executed on the miniHPC[29] of the University of Basel. The mini HPC system contains 22 computing nodes of Intel Xeon E5-2640 nodes on which the experiments were executed.

5.1 Design of the Experiments

The experiments have been designed according to the table 5.1. We executed each application with 10 scheduling techniques consisting of seven non-adaptive dynamic scheduling techniques and three static ones with a fixed steal ratio. The ten scheduling techniques have used the three introduced victim selection strategies, random work-stealing naive- and locality-aware work-stealing. Each experiment was executed 5 times and led to a total of $1'050$ experiments performed. Due to the possibility of configuring the workload imbalance, Pisolver has been chosen. Mandelbrot has been selected to perform the experiments as this is a workload-imbalanced application. Furthermore, SPH-EXA (Sedov)[12] is a real-used scientific application for smoothed particle hydrodynamics. Finally, miniAMR[14] has been chosen, as it is a significant MPI application executable in a large distributed environment. Having various applications leads to building and adapting the LB4MPI to a generic environment for multiple datatypes.

Table 5.1: Design of the experiments

Factors		**Values**	**Properties**
Applications	Process-level parallelism	Mandelbrot (distributed)	N = 1,048,576,144 — T = 1500 — Total loops = 3 — Modified loops = 3
		Pisolver	N = 2,000,000 — T = 150 —Total loops = 2 — Modified loops = 1
		SPH-EXA Sedov	N = 216,000,000 — T = 100 — Total loops = 16 — Modified loops = 1
		miniAMR	N = 110,080 — T = 200 — Total loops = 2 — Modified loops = 1
Process-level Scheduling	LB4MPI-RWS	rws_static	Randomized work stealing and chunk calculation is based upon a steal ratio
		rws_ss, rws_FSC, rws_mFSC, rws_tss, rws_gss, rws_fac, rws_wf	Randomized work stealing and chunk calculation is based on dynamic and non-adaptive self-scheduling technique
	LB4MPI-LAWS	laws_static	Locality aware work stealing and chunk calculation is based upon a steal ratio
		laws_ss, laws_FSC, laws_mFSC, laws_tss, laws_gss, laws_fac, laws_wf	Locality aware work stealing and chunk calculation is based on dynamic and non-adaptive self-scheduling technique
	LB4MPI-nLAWS	nlaws_static	Naive locality aware work stealing and chunk calculation is based upon a steal ratio
		nlaws_ss, nlaws_FSC, nlaws_mFSC, nlaws_tss, nlaws_gss, nlaws_fac, nlaws_wf	Naive locality aware work stealing and chunk calculation is based on dynamic and non-adaptive self-scheduling technique
	Steal Ratio	25%,35%,50%	Percentage to steal from the remaining iterations on the victim thread for rws_static

Computing nodes		miniHPC-KNL	Intel(R) Xeon Phi(TM) CPU 7210 (1 socket, 64 cores)
			P=64 cores without hyperthreading, Pinning: OMP_PLACES=cores OMP_PROC_BIND=close

		miniHPC-Xeon	Intel Broadwell E5-2640 v4 (2 sockets, 10 cores each) P=20 cores without hyperthreading, Pinning: OMP_PLACES=cores OMP_PROC_BIND=close
		miniHPC-Cascade Lake	Intel Xeon Gold 6258R (2 sockets, 28 cores each) P=56 cores without hyperthreading, Pinning: OMP_PLACES=cores OMP_PROC_BIND=close
Metrics		T_{Par}	Parallel execution time of the loops
		LIB	Work load imbalance
		N_{Steal}	Number of successful stealing operations

5.2 Pisolver

Pisolver is a C/C++ implementation of parallel workloads in an MPI environment for message exchange with blocking or non-blocking pairwise and collective operations on a Cartesian grid topology. It intends to test the performance of distributed memory parallel applications. Initially, it followed a distribution with replicated data, which was changed to a version that uses distributed data. It is a suitable test benchmark, as we can parse the initial workload with a workload imbalance parameter in percentage. The algorithm generates random numbers up to the given workload while considering the maximum possible given workload imbalance. Based on the configured imbalanced workload Pisolver achieves an imbalanced workload during execution. The input data is generated during execution, resulting in a one-dimensional array consisting of the datatype long. During the execution, it will take the workload and perform the computations where LB4MPI can dynamically schedule the distributed data with the work-stealing algorithm. The experiments have been executed with an input imbalanced workload of 0%, 20%, 30%, and 40% while the long array contained up to $680'000$ iterations as a workload. Additionally, the environment was set to have ten nodes containing each 20 processes.

5.2.1 Comparison of different workload imbalances

By comparing the configured workload imbalance in Pisolver, we can see that with an increasing percentage, the application's parallel execution time increased. Thus, we expected that a higher workload imbalance would mean a longer execution time for the application. As shown in figure 5.1, an increasing workload in Pisolver is followed by a longer parallel execution time per rank. Furthermore, we can see the employed scheduling techniques during the experiments. The best-performing scheduling technique is the static division with a fixed steal-ratio of 50%. On the other hand, a static execution without LB4MPI leads to the worst performance. In the following subsections, we dive into the details of the executions with the different workload imbalances.

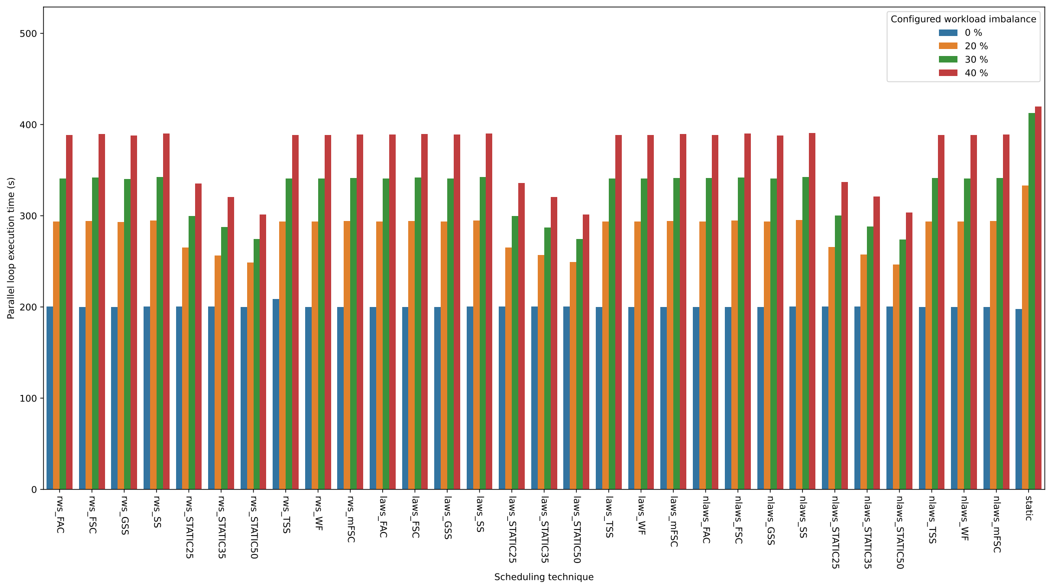


Figure 5.1: **Comparison of the parallel execution time about the scheduling techniques used and categorized by the different input workload imbalance:** In the x-axis, we can see all the scheduling techniques employed during execution. The y-axis corresponds to the parallel execution time in seconds. Furthermore, the plot is categorized by the configured workload imbalance of 0%, 20%, 30%, and 40%.

5.2.2 Performance evaluation

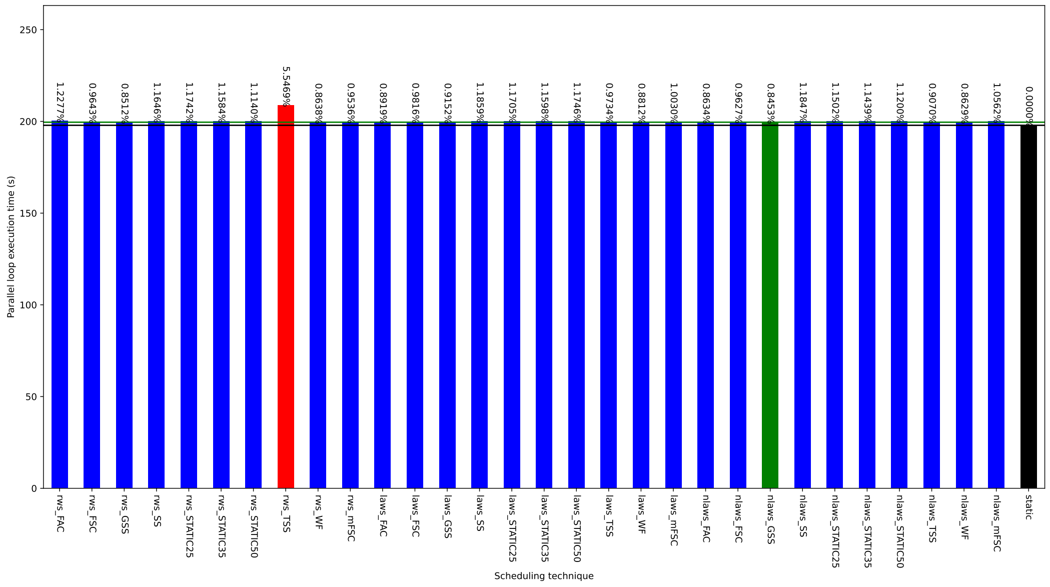
Following the last section, we analyzed how Pisolver performed with different workload imbalances. First, the performance is compared with 0% and 20% workload imbalance and then with 30% and 40% load imbalance.

In figure 5.2a, the experiments executed with no LB4MPI are similar to those achieved with LB4MPI. All the scheduling techniques seem to perform equivalently in their execution with a maximal difference in the performance of 5.55%. The parallel execution time is in the range of 200s for every scheduling technique.

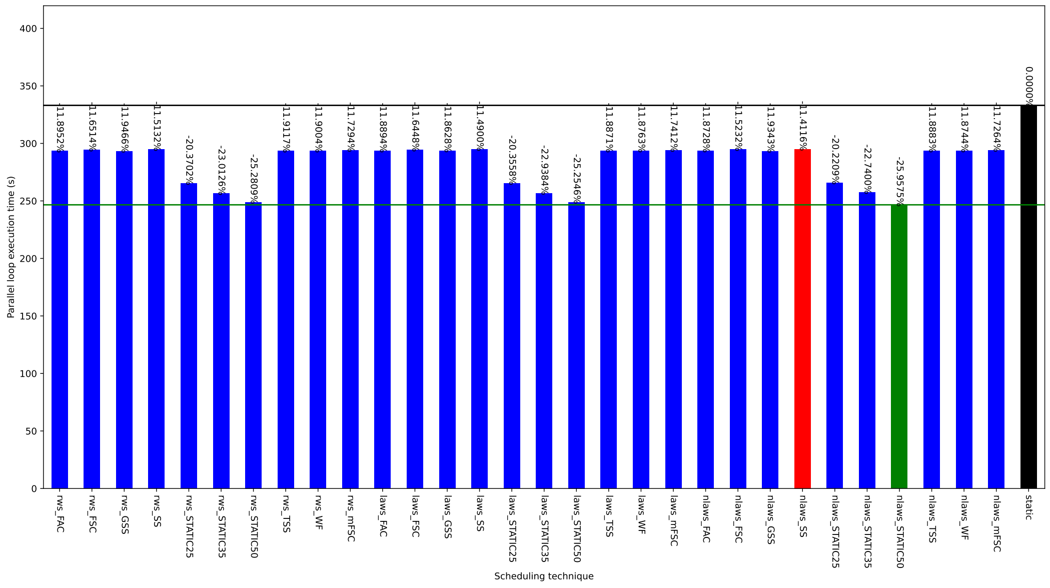
Comparing the results in 5.2b, we can notice that the employed scheduling techniques performed better than the application without LB4MPI. The performance increase of the LB4MPI lies between 11.95% to 25.96%. The differences in the victim selection strategy seem to make a significant difference in performance. However, the employed locality-aware and naive locality-aware victim selection strategies do not induce communication overhead to the experiment. Among the scheduling techniques employed, the $rws\_STATIC50$ is performing the best.

The experiments performed with a workload imbalance of 30%, respectively of 40%, are shown in figure 5.3. In both, the integrated LB4MPI library leads to increased performance. However, by comparing the performance ranges relative to the execution without LB4MPI, we can see an increased performance range of 17.05% to 33.61%. for a workload imbalance of 30%, respectively a range of 7.02% to 28.22% with a workload imbalance of 40%. Again, the $rws\_STATIC50$ outperformed the employed scheduling techniques. Furthermore, the com-

munication overhead of the victim selection strategies seems not to lead to communication overhead.
In both figures 5.2 and 5.3, the static scheduling with an input steal ratio of 25%, 35%, and 50% outperform the dynamic scheduling technique in the distributed version of Pisolver. The input steal ratio parameter of 50% performs better than the steal ratios with 25% and 35%.

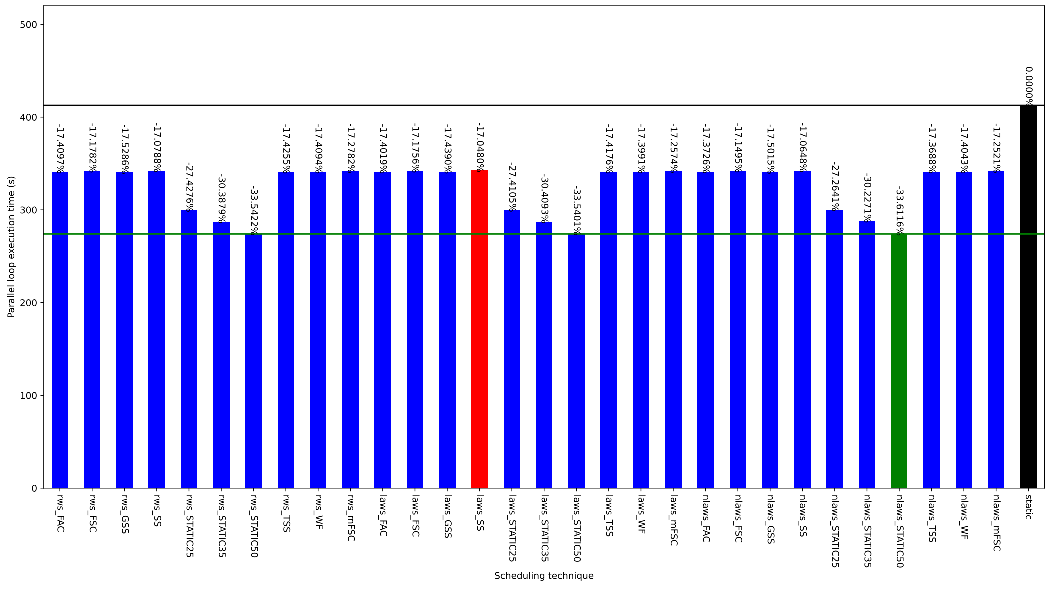


(a) **Pisolver with** 0% **workload imbalance**

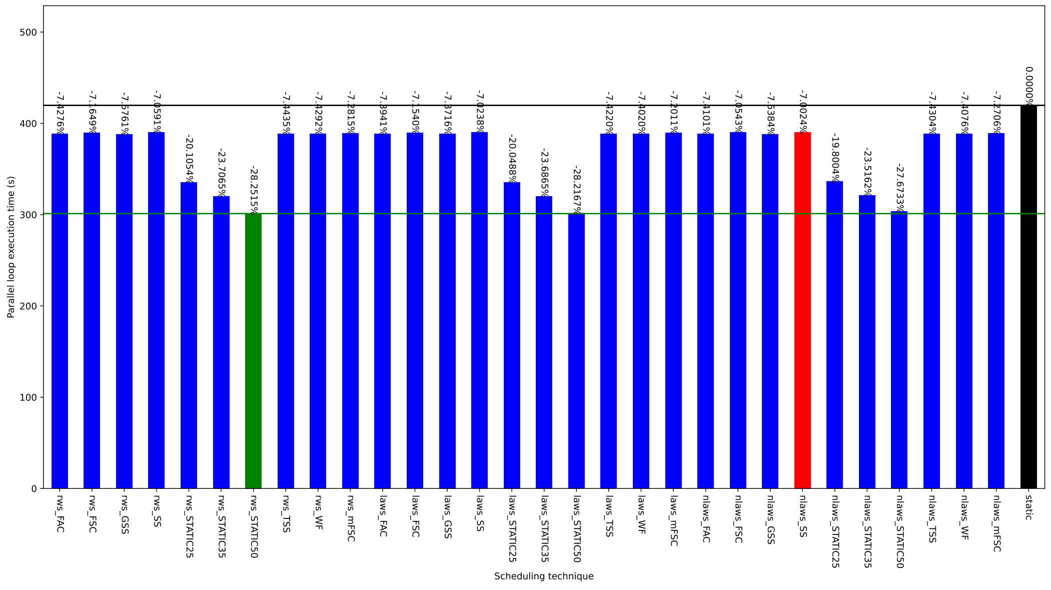


(b) **Pisolver with** 20% **workload imbalance**

Figure 5.2: **Pisolver executed with** 0% **and** 20% **workload imbalance:** On the x-axes, we show the employed scheduling techniques, whereas on the y-axes, the parallel loop execution time is shown. Furthermore, the black color represents the results of the static execution without LB4MPI. The red bar illustrates the worst-performing dynamic scheduling technique. The green bar represents the best-performing scheduling technique. The black labels above the bars are the percentage differences from the static version without LB4MPI.



(a) **Pisolver with** 30% **workload imbalance**

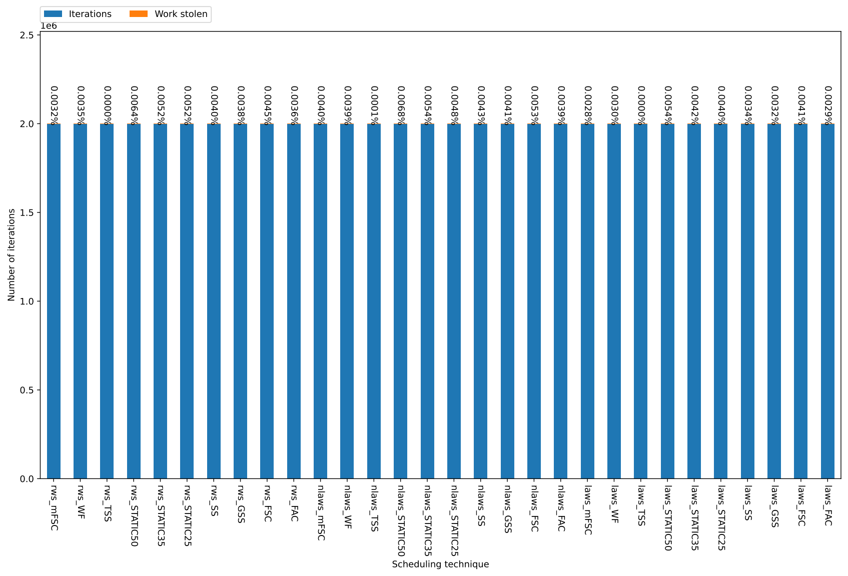


(b) **Pisolver with** 40% **workload imbalance**

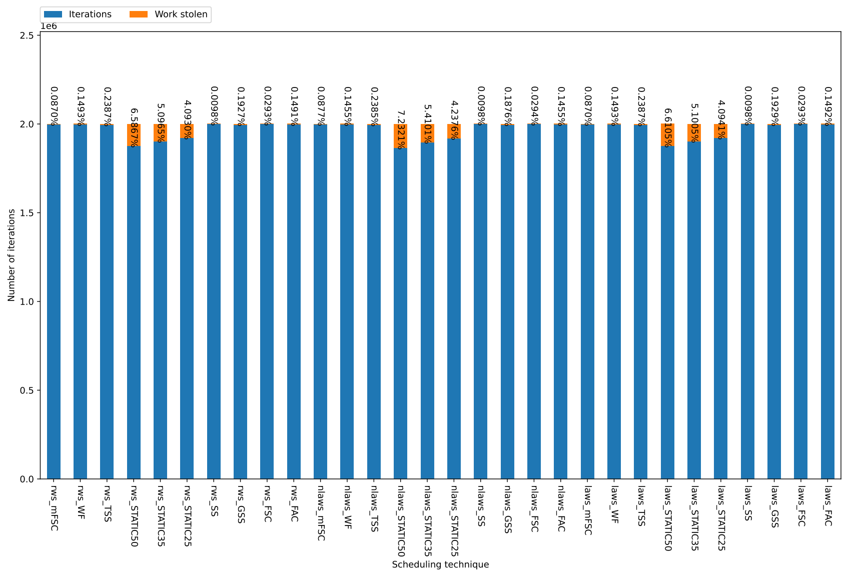
Figure 5.3: **Pisolver executed with** 30% **and** 40% **workload imbalance:** On the x-axes, we show the employed scheduling techniques, whereas on the y-axes, the parallel loop execution time is shown. Furthermore, the black color represents the results of the static execution without LB4MPI. The red bar illustrates the worst-performing dynamic scheduling technique. The green bar represents the best-performing scheduling technique. The black labels above the bars are the percentage differences from the static version without LB4MPI.

5.2.3 Iterations stolen

We measured the iterations executed by the processes and gathered the total amount of stolen iterations. Investigating the number of iterations stolen in figure 5.4 shows the total work stolen about the total number of iterations scheduled. Hence, only a fraction of the total work has been stolen at a small percentage rate. In figure 5.4a with an input workload imbalance of 0%, not many iterations are stolen compared to the employed scheduling techniques. However, the work stolen is increased as soon as the input workload imbalance to 20%. Moreover, static scheduling techniques with a fixed steal ratio steal the most iterations. Their amount of work stolen ranges from approximately 4.1% to 7.2%. Considering the total iterations of two million, the total stolen work corresponds to $82'000$ to $144'000$ iterations, where a higher steal ratio leads to more stolen work. Similarly, a further increase of the workload imbalance of 30% and 40% is shown in figure 5.5. Again, static scheduling with a fixed steal ratio steals the most iterations. Moreover, an increasing workload imbalance leads to more stolen work. Interestingly, dynamic self-scheduling techniques do not steal much work compared to the static case with a fixed steal ratio. Hence, this indicates that the work available to steal is less, as processes already perform self-scheduling techniques to tackle load imbalance.

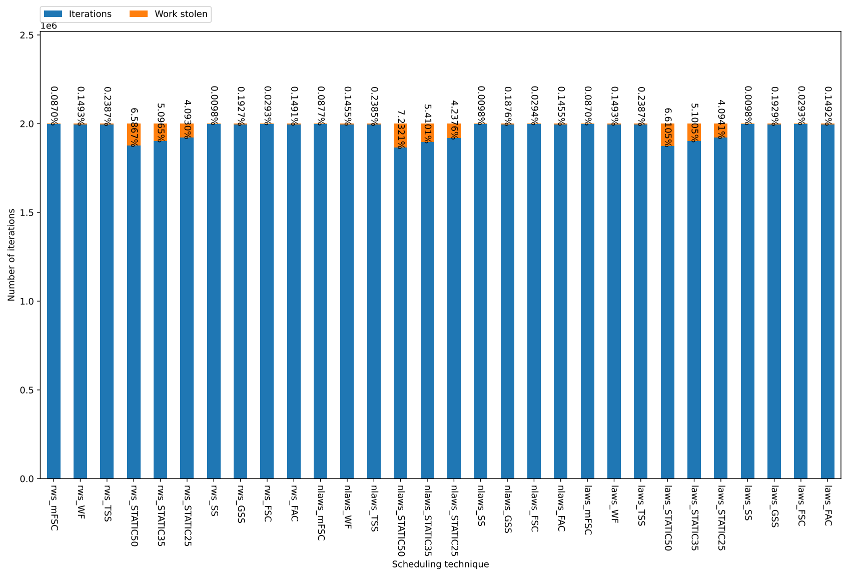


(a) **Pisolver with** 0% **workload imblance**

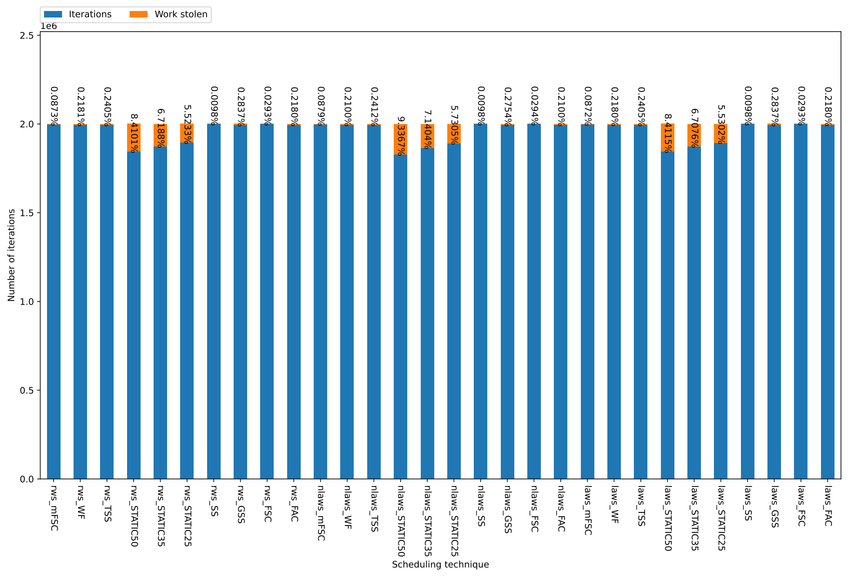


(b) **Pisolver with** 20% **workload imbalance**

Figure 5.4: **Pisolver executed with** 0% **and** 20% **workload imbalance - Relation of total iterations scheduled and total work stolen:** The y-axes display the total number of iterations performed on average. The x-axes show the employed scheduling techniques. On top of the blue bars, we can see the fraction of work that has been stolen per scheduling technique with the corresponding percentage.



(a) **Pisolver with** 30% **workload imbalance**



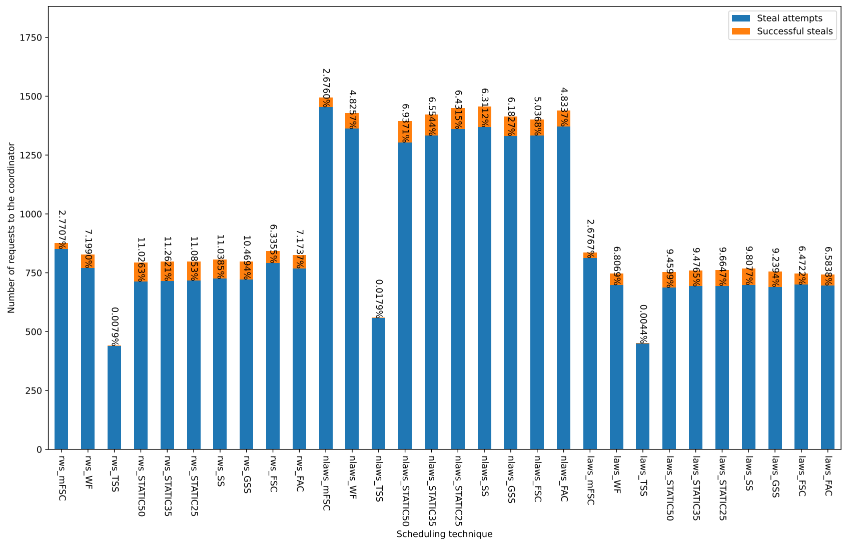
(b) **Pisolver with** 40% **workload imbalance**

Figure 5.5: **Pisolver executed with** 30% **and** 40% **workload imbalance - Relation of total iterations scheduled and total work stolen:** The y-axes display the total number of iterations performed on average. The x-axes show the employed scheduling techniques. On top of the blue bars, we can see the fraction of work that has been stolen per scheduling technique with the corresponding percentage.

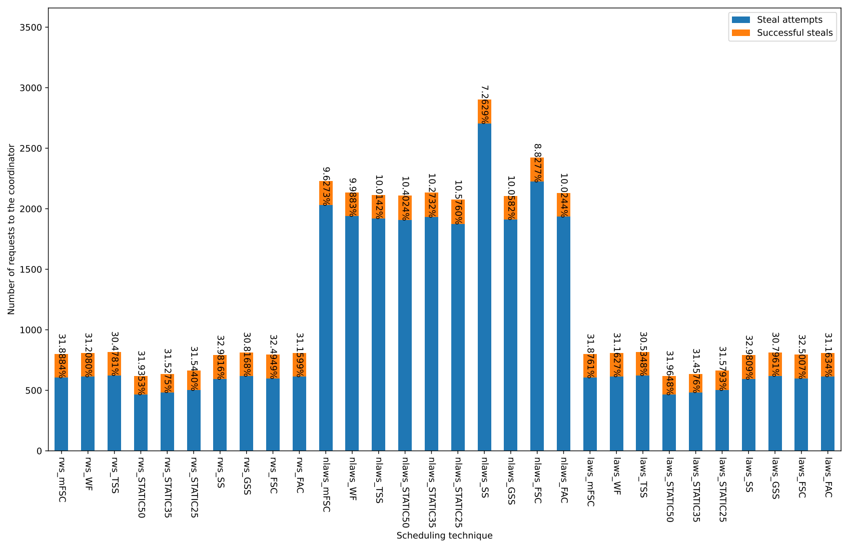
5.2.4 Steal attempts

Another aspect that we are interested in is measuring the steal attempts and comparing them to the number of successful steals during the experiments. In figure 5.6, the total amount of steal attempts in relation to the successful steals during the experiments is illustrated.

By comparing the executed version of Pisolver with 20% induced work imbalance with no induced work imbalance, we can see that the number of steals is significantly higher in naive locality-aware work-stealing. Generally, we can see that the naive locality-aware victim selection strategy leads to more attempts to steal the work from other ranks. However, the locality-aware victim selection strategy requests work similarly to the random victim selection strategy. Comparing the employed scheduling techniques, the dynamic scheduling techniques are performing more steal attempts to steal the work. Furthermore, the success rates are similar to ca. 30% in the scheduling techniques using a locality-aware or random victim selection strategy. Contrary, the naive locality-aware strategy has a lower successful steal attempts with ca. 10%. As the performance evaluation showed that the differences in the used victim selection strategies were insignificant, we can conclude that the amount of work to be stolen matters more. Hence, even if the naive locality-aware victim selection strategies yielded higher total requests, the induced communication overhead did not affect the performance considerably.



(a) **Pisolver with** 0% **workload imbalance**



(b) **Pisolver with** 20% **workload imbalance**

Figure 5.6: **Pisolver executed with** 0% **and** 20% **workload imbalance - Number of steal attempts versus successful steals per scheduling technique:** The fraction of the successful steal requests is given within the orange bars. The x-axis lists the scheduling techniques. The y-axis shows the total number of steal attempts and successful steals labeled by the fraction of successful steal requests.

5.2.5 Coefficient of variance after LB4MPI

During the execution, each process performs its own data chunk. Having in Pisolver an induced workload imbalance, we were interested in the outcome of the c.o.v obtained by all process execution times. In figure 5.7, we can see that for Pisolver executed with a workload imbalance of 20%, the c.o.v is in the range of 0.00002% to 0.00003%. This showed that the processes were finishing their execution with differences in microseconds.

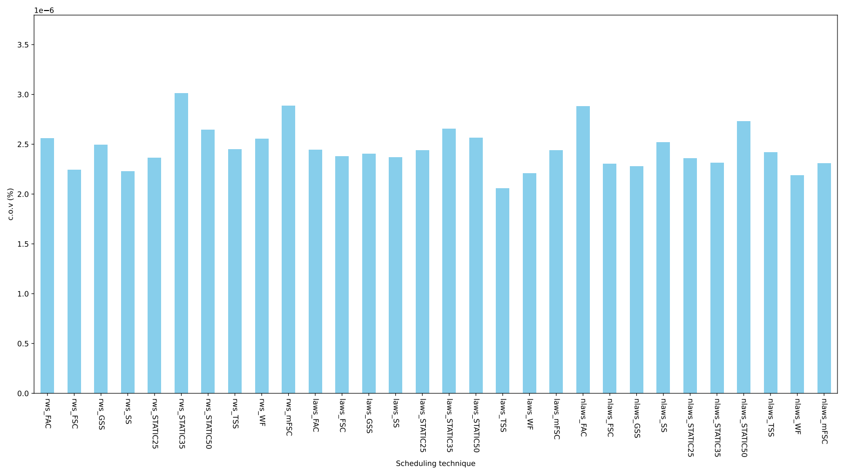


Figure 5.7: **Pisolver with 20% load imbalance - Calculated c.o.v of the parallel loop execution times obtained:** On the x-axis, the scheduling techniques are listed. On the y-axis, one can see the c.o.v. of the parallel loop execution time

5.2.6 Send and receive time per chunk

This section presents the variations of sending and receiving chunks. We measured the send time within the responsible victim's *MPI_Send*, which sends the work to the coordinator. On the other hand, we measured the time to receive a chunk by the thief within its corresponding *MPI_Recv*(). In both cases, we took measurements before and after the MPI operation.

As the iterations are sent in chunks varying in sizes from the victim processor to the thief processor, we illustrate the time to send and receive a chunk for a rank in figure 5.8. The chunk send times have fewer outliers than the chunk receive times. This might be due to synchronization overhead, as the sending side can put the work and data in the buffer and continue to send. Besides, the receiving side waits until the work and data are delivered. Hence, we have a slightly more extended time on the receiving side. Another factor that influences the send and receive times is the workload that will be sent. So, larger chunks of data need automatically more time to be received. A good example is the send time of *rws_GSS*, which has larger chunks to send at the beginning of the chunk calculation. When the victim prepares the work, it depends on how much its remaining work will be sent, leading to a higher send time in the outliers. On the contrary, the thief waits to gather the data and execute, which means it has a more considerable receive time.

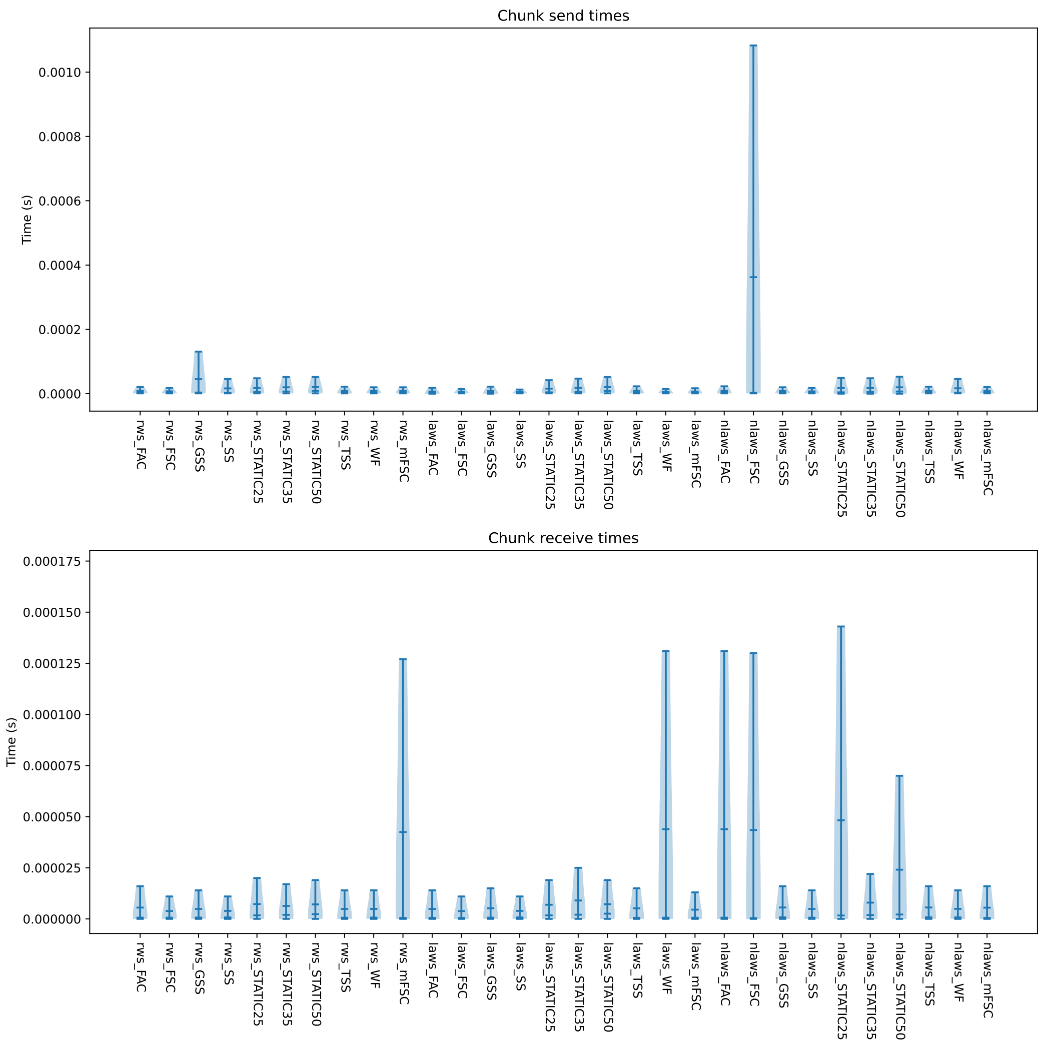


Figure 5.8: **Pisolver executed with 20% workload imbalance - Send and receive times of the work per iteration:** Here, a representation of the end and receive times of the work and data to be delivered from the victim to the thief is shown. On the y-axis, the needed time is shown. The scheduling techniques are listed on the x-axis. In blue, one can see the distribution of the minimum, average, and maximum time needed. The light blue area represents the occurrences of time needed.

5.3 Mandelbrot

Mandelbrot is an initially replicated sample *C* application that calculates the Mandelbrot set[30]. In [11], the experiments have been performed to engage with thread-level scheduling. So, we adapted the application to execute the experiments with process-level scheduling during this work. To perform the experiments, we used $1'500$ time steps with a $1'024 \times 1'024$ pixel size on eight nodes containing each of 16 processes. In the following experiments, we wanted to understand how the LB4MPI performs with an application with no real data input but a set of maximal iterations executed. Furthermore, Mandelbrot is an imbalanced application where we integrated the LB4MPI.

5.3.1 Performance evaluation

Compared to the distributed version of mandelbrot without LB4MPI, the work-stealing algorithm performs poorer by a factor of two, as shown in figure 5.9. Hence, communication costs have been introduced to an application where balancing is not necessarily required. However, we can still see and compare the performance of the scheduling technique incorporated with the victim selection strategy. Noticeably, the victim selection strategy naive locality-aware scheduling achieves the best performance among the other strategies. Furthermore, dynamic scheduling techniques perform better than the static steal ratio scheduling technique. So, weighted factoring steal amount calculation in combination with a naive locality-aware victim selection strategy performs the best out of the other techniques.

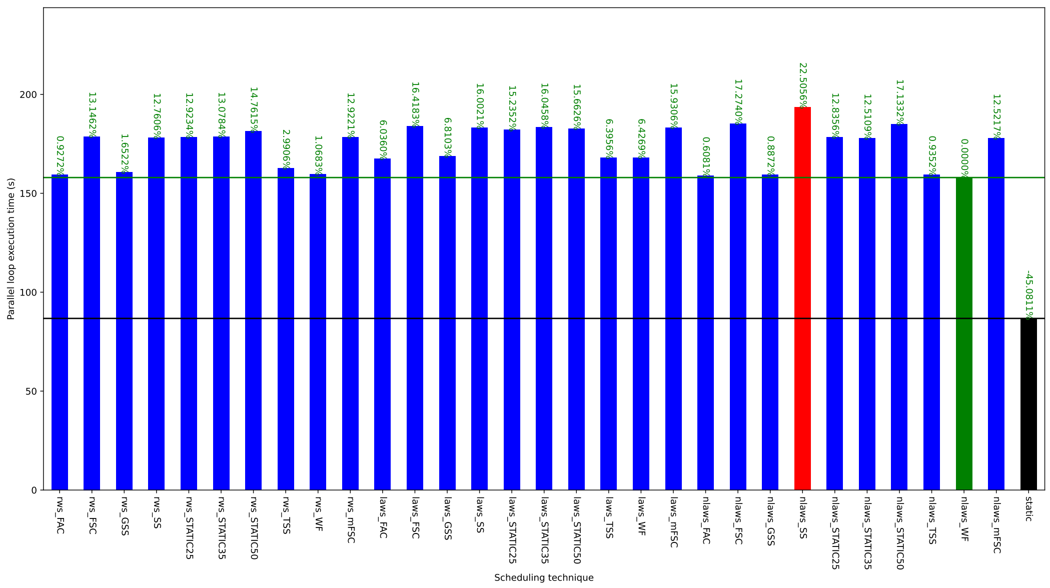


Figure 5.9: **Mandelbrot - Parallel loop execution time in comparison with the no LB4MPI version** On the x-axes, we show the employed scheduling techniques, whereas on the y-axes, the parallel loop execution time is shown. Furthermore, the black color represents the results of the static execution without LB4MPI. The red bar illustrates the worst-performing dynamic scheduling technique. The green bar represents the best-performing scheduling technique. The green labels above the bars are the percentage differences from the best-performing scheduling technique.

5.3.2 Iterations stolen

Looking at the relation between stolen work and iterations completed during the experiment, we can see that generally, a maximal amount of 0.48% have been stolen. Having around $8'000$ iterations scheduled per rank leads to a lower amount of work to be stolen by a rank; for instance, 0.48% of $8'000$ iterations means that 38.4 iterations have been stolen on average. Thus, having a lower amount of iterations that will be stolen would induce a higher communication cost, as can be seen in 5.10.

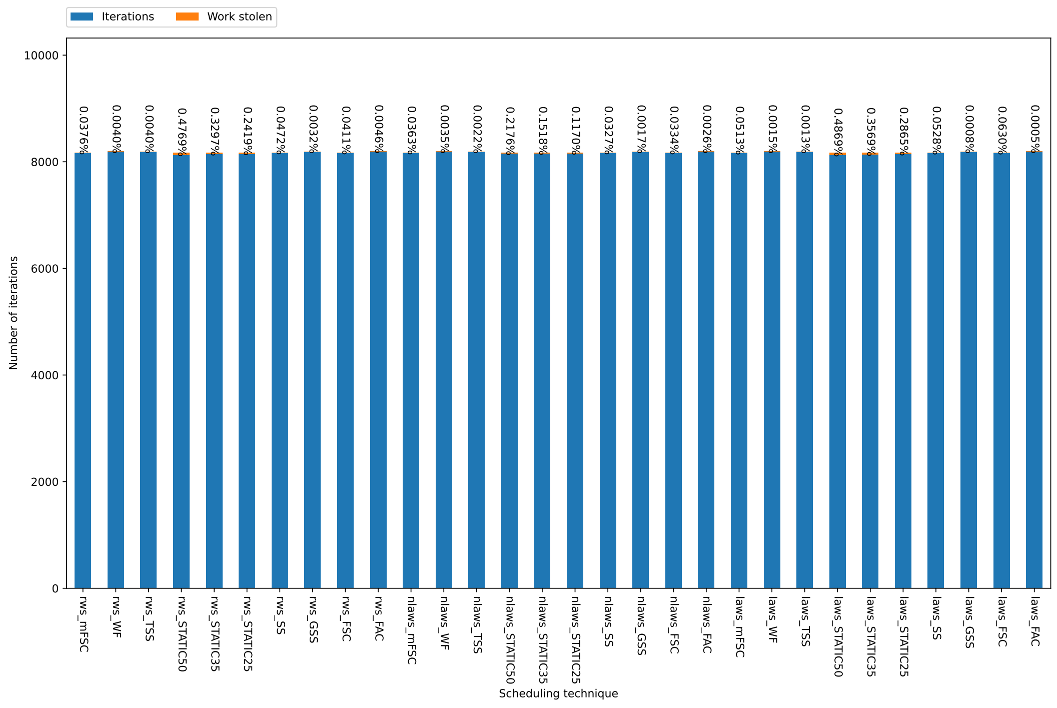


Figure 5.10: **Mandelbrot - Relation of total iterations scheduled and total work stolen:** The y-axes display the total number of iterations performed on average. The x-axes show the employed scheduling techniques. On top of the blue bars, we can see the fraction of work that has been stolen per scheduling technique with the corresponding percentage.

5.3.3 Steal attempts

The higher induced communication cost can also be seen in the ratio between successful steals and steal attempts in 5.11. There are scheduling techniques that have had successful steals. However, about the total number of requests to the coordinator, this would be a relatively poor quote, as only a few steal requests have been successful. Thus, the work-stealing algorithm induces in this application more communication overhead, as the application can benefit from the load balance it achieves. Interestingly, scheduling techniques responsible for stealing a fixed amount of work, such as FSC, $STATIC25$, $STATIC35$, and $STATIC50$, induced a higher communication overhead merged with all victim selection strategies.

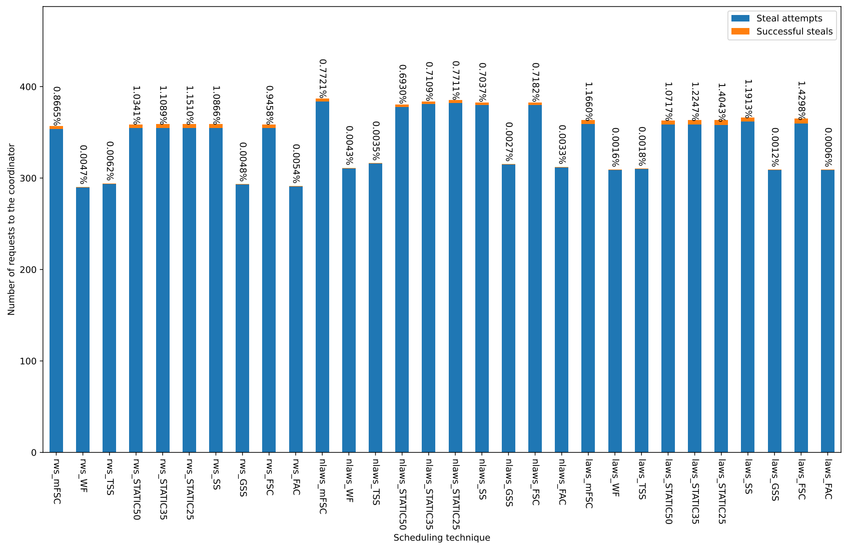


Figure 5.11: **Mandelbrot - Number of steal attempts versus successful steals per scheduling technique:** The fraction of the successful steal requests is given within the orange bars. The x-axis lists the scheduling techniques. The y-axis shows the total number of steal attempts and successful steals labeled by the fraction of successful steal requests.

5.3.4 Coefficient of variance after LB4MPI

Each process during the computation performs its own data chunk. Hence, when a process finishes its chunks, it asks for work from the coordinator and receives it from the selected victim. So, by the end of an experiment, we should have a lower amount of c.o.v in parallel execution times. Looking at figure 5.12, we achieve a low c.o.v in percentage with all scheduling techniques. Interestingly, the random and locality-aware victim selection strategies have scheduling techniques with a higher c.o.v. Among the scheduling techniques, FAC, GSS, TSS, and WF have higher c.o.v rates ranging up to 0.08% whereas the lowest c.o.v appears to be at 0.04%.

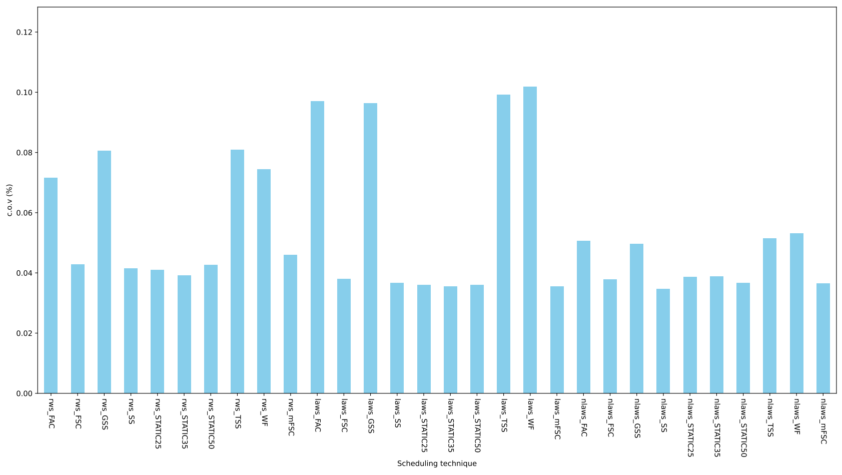


Figure 5.12: **Mandelbrot - Calculated c.o.v of the parallel loop execution times obtained:** On the x-axis, the scheduling techniques are listed. On the y-axis, one can see the c.o.v. of the parallel loop execution time

5.4 SPH-EXA - Sedov

SPH-EXA is an application of the smoothed particle hydrodynamic (SPH) technique, calculated by the Lagrangian method [13]. This application is designed to counter harmful elements such as imbalanced multi-scale physics, unique time-stepping, halos exchange, and prolonged ranged forces in physical calculations. It is part of the $SPHYNX$ project, designed to perform SPH on astrophysics applications[12]. We chose to integrate the LB4MPI extensions into the Sedov blast calculation of the SPH-EXA due to the encountered challenges in serializing the $C++-$objects used to pass into LB4MPI. The experiments were performed on eight nodes with 16 processes, whereas the input size was on 400^3 particles within a time-step of 50.

5.4.1 Performance evaluation

Resembling the application with LB4MPI and without LB4MPI leads to decreasing performance in SPH-EXA Sedov, as illustrated in figure 5.13. However, the difference between the LB4MPI and the library without lies in the range between 10 to 20 seconds. Among the chosen victim selection strategies, the naive locality-aware and random victim selection strategies perform better than the locality-aware work stealing. In this application, dynamic and static scheduling techniques are performing similarly.

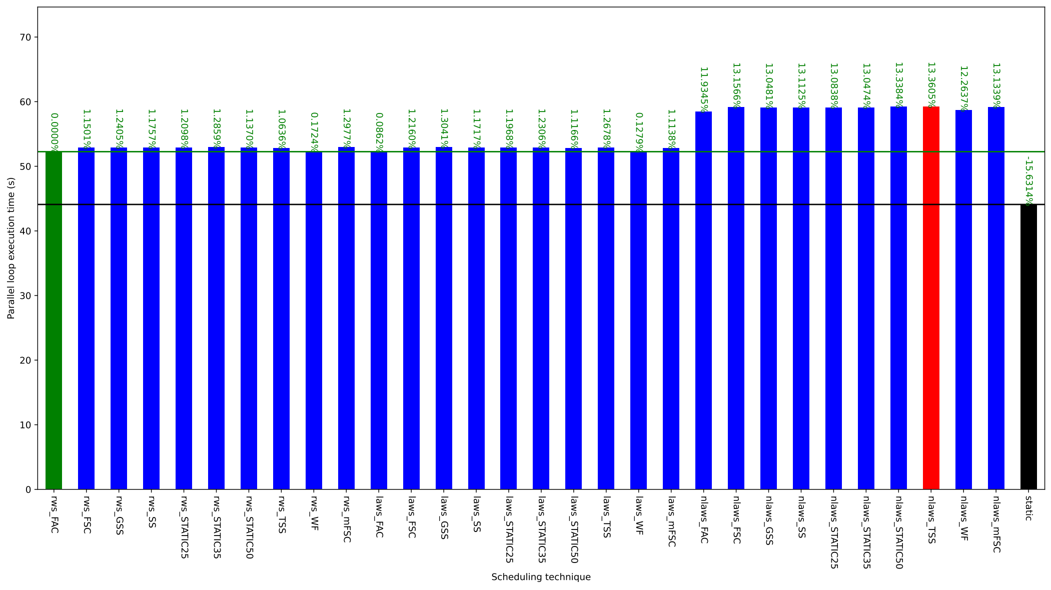


Figure 5.13: **SPH-Exa Sedov - Parallel loop execution time in comparison with the no LB4MPI version** On the x-axes, we show the employed scheduling techniques, whereas on the y-axes, the parallel loop execution time is shown. Furthermore, the black color represents the results of the static execution without LB4MPI. The red bar illustrates the worst-performing dynamic scheduling technique. The green bar represents the best-performing scheduling technique. The green labels above the bars are the percentage differences from the best-performing scheduling technique.

5.4.2 Iterations stolen

Analyzing the total iterations performed by a rank and the relation to the stolen work in figure 5.14 allows us to see the constant appearing ratio of stolen work. The values range between 0.54% to 1.04%, translating into 43.2 to 83.2 stolen iterations. Having this low number of work stolen lies like the well-balanced SPH-EXA Sedov application. In this case, the naive locality-aware are stealing double the amount of the other employed strategies. The scheduling techniques show no larger difference in work than the steal in this experiment.

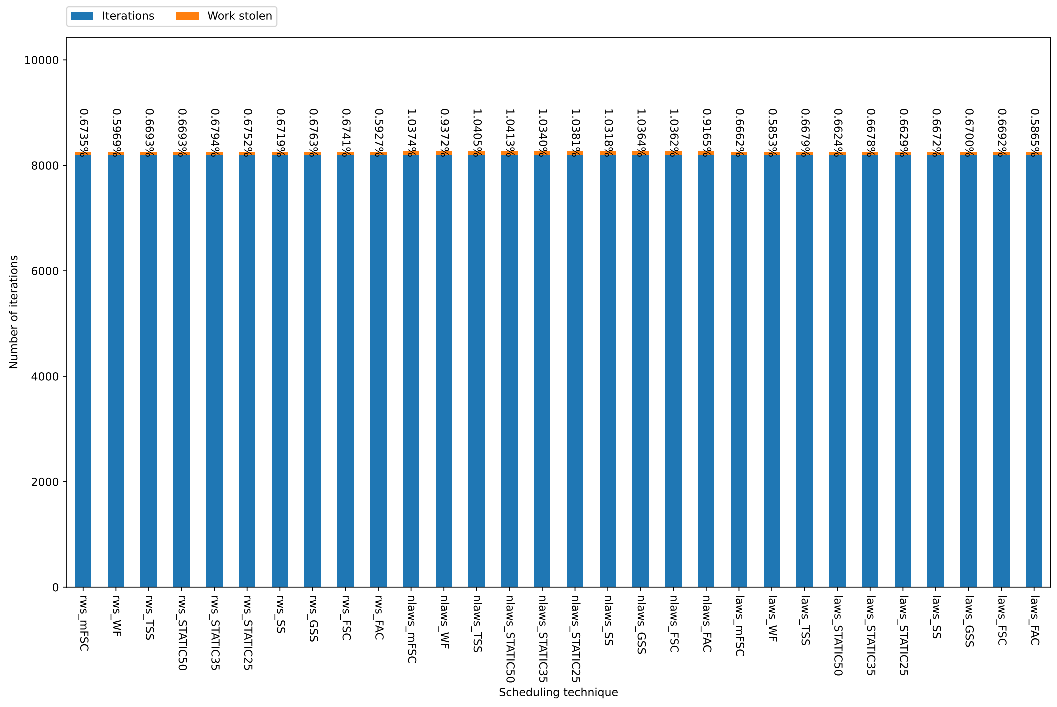


Figure 5.14: **SPH-EXA Sedov - Relation of total iterations scheduled and total work stolen:** The y-axes display the total number of iterations performed on average. The x-axes show the employed scheduling techniques. On top of the blue bars, we can see the fraction of work that has been stolen per scheduling technique with the corresponding percentage.

5.4.3 Steal attempts

A further measurement that we are interested in SPH-EXA Sedov is the successful steals operating during the experiments shown in figure 5.15. The total number of requests sent to the coordinator lies in the area of 500 for the random and locality-aware victim selection strategy. However, the naive locality-aware scheduling techniques have more additional requests sent to the coordinator. Especially, $nlaws\_WF$ and $nlaws\_FAC$ exceeded with over 800 total requests sent to the coordinator. Although having more requests to the coordinator, the naive locality-aware scheduling methods are on par with the random victim selection strategy in performance.

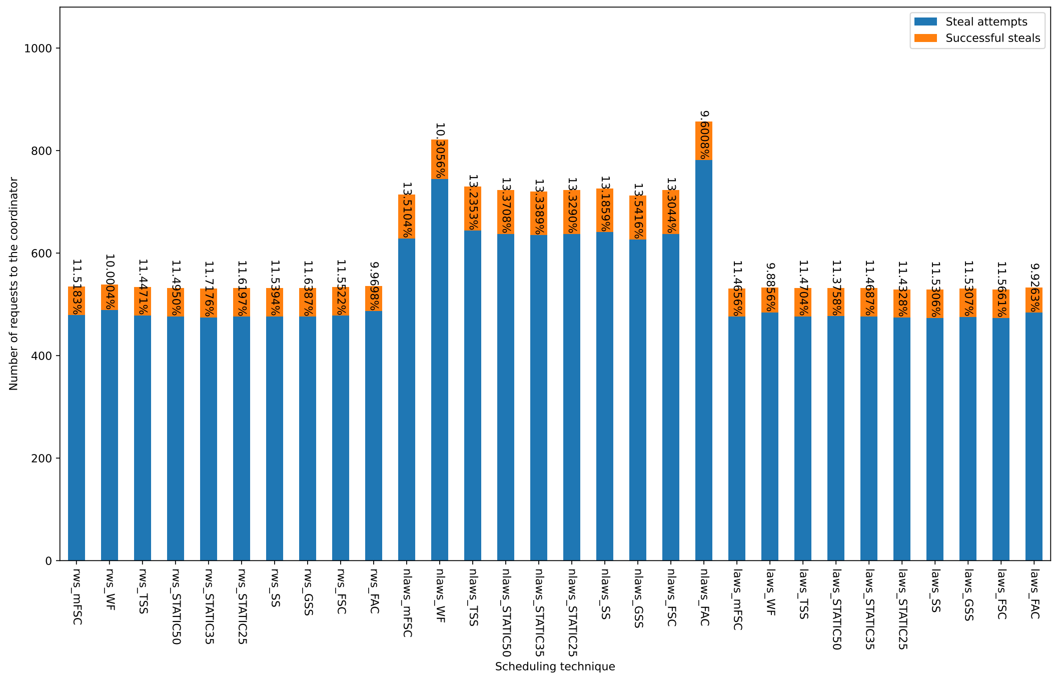
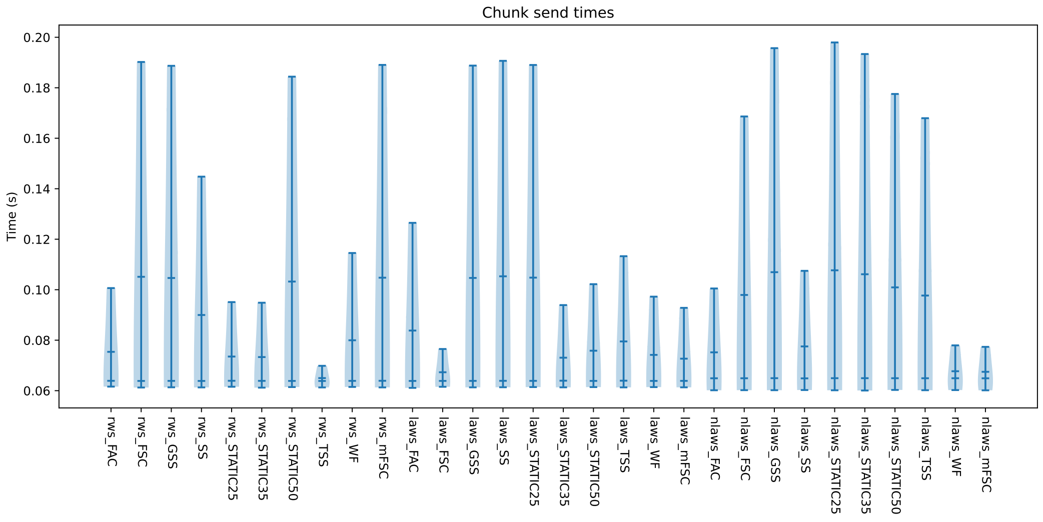


Figure 5.15: **SPH-EXA Sedov - Number of steal attempts versus successful steals per scheduling technique:** The fraction of the successful steal requests is given within the orange bars. The x-axis lists the scheduling techniques. The y-axis shows the total number of steal attempts and successful steals labeled by the fraction of successful steal requests.

5.4.4 Send and receive time per chunk

The integrated version of LB4MPI needs a serialization of the objects of SPH-EXA Sedov. Hence, we measured the serialization and the send time, respectively, the receive time and deserialization, as both are needed to deal with objects.

This can be seen in figure 5.16. The send and receive times range to 0.2 seconds for a chunk appearing at every victim selection strategy. Another factor is the dynamic and static scheduling techniques used. Here, FSC, WF, and mFSC tend to have lower send times. The static scheduling techniques with a fixed steal ratio, GSS, and TSS have longer outlying send times. However, the mean send times to send a chunk appear to be the same for all scheduling techniques. On the receiving side, the scheduling technique's mean appears to be similar. Outliers emerge at every scheduling technique, but the discrepancy between the scheduling techniques is more minor than the disparity of the send times obtained.



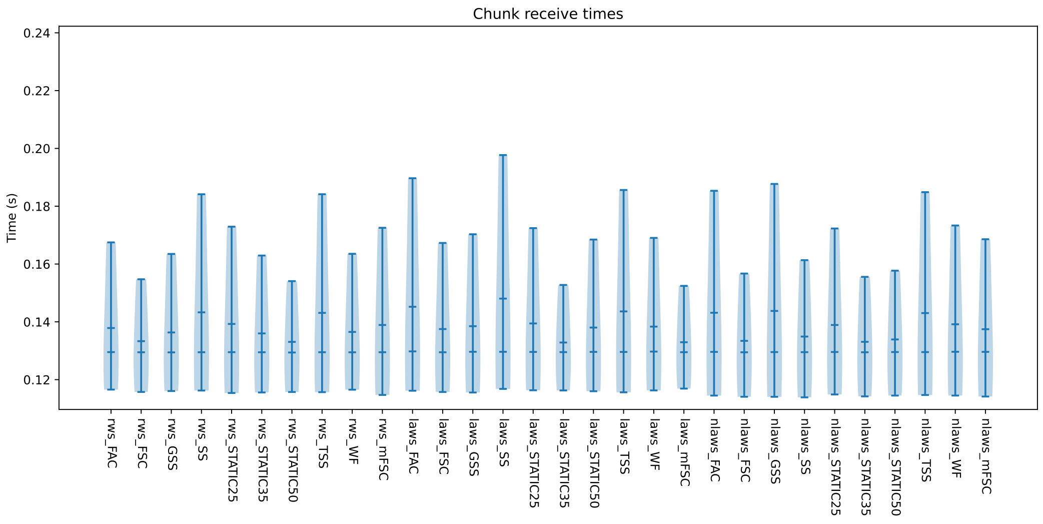


Figure 5.16: **SPH-EXA Sedov - Send and receive times of the work per iteration:** Here, a representation of the end and receive times of the work and data to be delivered from the victim to the thief is shown. On the y-axis, the needed time is shown. The scheduling techniques are listed on the x-axis. In blue, one can see the distribution of the minimum, average, and maximum time needed. The light blue area represents the occurrences of time needed.

5.5 miniAMR

The miniAMR[14] introduced in the Mantevo benchmark executes stencil calculation on a unit cube computational domain. The domains are further divided into blocks and cells to refine the objects pushed through the mesh. Furthermore, cells can communicate ghost values with adjacent blocks. Every block has an equal amount of cells in each direction, and blocks can describe distinct levels of refinement within the larger mesh with adaptive mesh refinement. Hence, the calculated grids have a fixed position in the finer mesh. The application has multiple configurable options such as block sizes, time steps, number of refinements, and numerous objects to collide with the grid calculated.

The distributed application is suitable to test and integrate the LB4MPI with cubic datatypes. The experiments have been conducted to simulate an expanding sphere on eight nodes with 16 processes per node. As we can see in the results, the application tends not to collaborate well with the work-stealing algorithm in LB4MPI.

5.5.1 Performance evaluation

In the following, we will look at an experiment where LB4MPI is not collaborating well with the application. As illustrated in figure 5.17, we can first see the low parallel execution time of the miniAMR without LB4MPI.

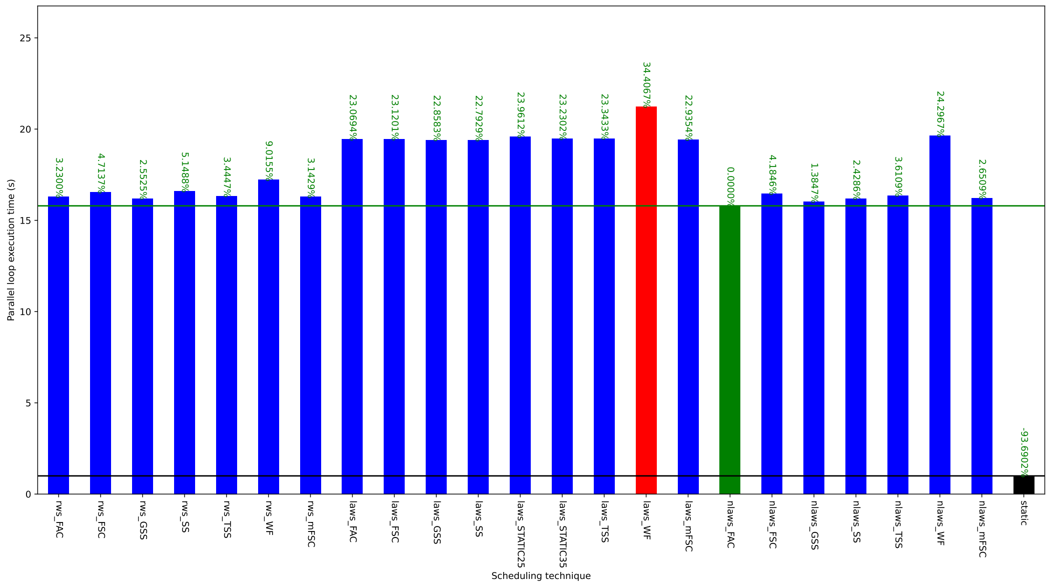


Figure 5.17: **miniAMR - Parallel loop execution time in comparison with the no LB4MPI version** On the x-axes, we show the employed scheduling techniques, whereas on the y-axes, the parallel loop execution time is shown. Furthermore, the black color represents the results of the static execution without LB4MPI. The red bar illustrates the worst-performing dynamic scheduling technique. The green bar represents the best-performing scheduling technique. The green labels above the bars are the percentage differences from the best-performing scheduling technique.

5.5.2 Limitations

Although miniAMR uses a distributed execution, the library does not elaborate well with work-stealing. The application uses adaptive mesh refinement by starting initially from the block sizes corresponding to the number of processes used. The refinement steps can generally be configured and parsed into the application, filling the finer grid into cells. By configuring a maximum amount of blocks used in the application, we can scale the experiments to our fulfillment. Nevertheless, the initialization points consist of an initial number of processes which in our experiments corresponds to 128 processes. However, when the work-stealing applies, the grid will be imbalanced, and a process will have one more cell that can be placed over other cells. In miniAMR, checksums are applied to ensure the correctness of the grid. By stealing work from other processes, these checksums cannot be calculated and traced where the grid is imbalanced. Additionally, a configurable error threshold is applied where at exceeding the value, the application will stop executing. Gathering these conditions, the current work-stealing in LB4MPI cannot be integrated correctly into miniAMR. However, the experience raised awareness of the limitations of LB4MPI. An expansion is needed to incorporate LB4MPI to work with these types of applications. An interesting aspect might be to extend the library by a work borrowing method where the thief becomes a borrower, and the victim becomes a lender. Another possibility would be to include a checksum function in LB4MPI.

6 Conclusion

This work extended the LB4MPI's work-stealing algorithm, a library that can be used with distributed MPI applications using dynamic loop-scheduling techniques. It provides 15 possible loop-scheduling techniques and three different victim selection strategies, random, locality-aware, and naive locality-aware victim selection. Furthermore, it can deal with data mapping, including one-dimensional or cubic data applications. Additionally, we provided a way to serialize and deserialize data streams that come for applications using complex data types. We evaluated the performance of four applications, Pisolver, Mandelbrot, SPH-EXA Sedov, and miniAMR, executing them on a multi-core system.

Pisolver had benefitted by using LB4MPI when the input workload imbalance was higher than zero. Generally, the static scheduling techniques with a fixed steal ratio performed best by reaching a performance increase of 25.96%. Differences in the victim selection strategy have not had a significant impact. Moreover, this shows that having a certain amount of imbalance, the work-stealing might help to improve the performance even if the stolen work from the iterations is not very significant. However, Pisolver also showed that stealing 50% of the scheduled chunk can lead to a significant performance improvement.

Mandelbrot helped determine how the LB4MPI performed in an application without explicit data input. Due to the nature of Mandelbrot, which uses a fixed amount of iterations to perform a relatively simple equation, the LB4MPI performed worse by a factor of two. Yet, we think a greater workload leads to a bigger work imbalance in Mandelbrot, where LB4MPI could improve performance.

SPH-EXA Sedov incorporated with LB4MPI was a challenging application to integrate the LB4MPI in. As LB4MPI is written in C, an application-side serialization of the data is required. Hence, LB4MPI can serialize before a send by calling function pointers on the application side and send the work to another process by calling function pointers to deserialize the data. The performance was slightly worse by $10s$ compared to the application without MPI. However, this undermines the difficulty of finding applications with a high load imbalance that can be tackled with the work-stealing in LB4MPI. Contrary to the results obtained by Pisolver, the method to use here would be the random victim selection strategy with Factoring as a scheduling method.

Experiments with miniAMR have shown two future key points: Not every application can be integrated with the current version of LB4MPI, and the need to study the data dependency needs to be considered more for future work. However, it was a good sample application to include an extension of LB4MPI that can handle cubic data using the customizable $MPI\_Datatype$ function. Distributed MPI applications with high load imbalance can benefit from LB4MPI's work-stealing algorithm.

7 Future work

Work-stealing merged with dynamic loop self-scheduling techniques is an excellent way to deal with MPI applications' load imbalances. During this work, the performance evaluations depended on applications using different data types within their calculation. However, choosing the proper application to experiment on can be difficult, as lines of code, the complexity of the domain, and code interpretation are critical points for integrations. So, to evaluate the LB4MPI work-stealing algorithm with cubic data types and objects, further distributed applications need to be chosen. By providing a variety of self-scheduling techniques and victim selection strategies, the exploration with different applications can undermine the benefits of LB4MPI.
Another exciting aspect would be to use different scheduling techniques in an experiment. For example, process A executes with GSS, process B executes with a FAC, while the steal amount depends on the victim process.
In Pisolver, we showed that LB4MPI achieves a significant performance increase. Further, using a fixed steal-ration of 50% was more beneficial. It would be interesting to see how the fixed steal ratio performs with a scheduling technique.
As previously shown, the LB4MPI has limitations when dealing with applications that rely on checksums. So, to make the LB4MPI more adaptable to different distributed applications, we can propose two ways to collaborate with those applications. One point would be to extend a checksum function that checks the correctness of an application's data. Another point might be to extend the method by using process-level work-borrowing, such that LB4MPI can deal with another technique used in distributed systems.
To achieve a generic application that can deal with almost all data types, a $C++$ implementation of LB4MPI would be needed. While having the advantage in $C++$ to use object-oriented programming, in-built callable functions, and generic data types, comprehensive coverage of distributed applications can be achieved.

Bibliography

[1] I. Banicescu and S.F. Hummel. Balancing processor loads and exploiting data locality in n-body simulations. In *Supercomputing '95:Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, pages 43–43, 1995.

[2] Ali Mohammed, Ahmed Eleliemy, Florina M. Ciorba, Franziska Kasielke, and Ioana Banicescu. An approach for realistically simulating the performance of scientific applications on high performance computing systems. *Future Generation Computer Systems*, 111:617–633, 2020.

[3] Constantine D. Polychronopoulos and David J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, C-36(12):1425–1439, 1987.

[4] Susan Flynn Hummel, Edith Schonberg, and Lawrence E. Flynn. Factoring: A method for scheduling parallel loops. *Commun. ACM*, 35(8):90–101, aug 1992.

[5] David S Johnson. The np-completeness column: an ongoing guide. *Journal of Algorithms*, 6(3):434–451, 1985.

[6] Ioana Banicescu and Vijay Velusamy. Load balancing highly irregular computations with the adaptive factoring. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, IPDPS '02, page 195, USA, 2002. IEEE Computer Society.

[7] MPI. Mpi: A message-passing interface standard version. https://www.mpi-forum.org/docs/. Accessed: 2022-12-12.

[8] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, sep 1999.

[9] Jean-Noël Quintin and Frédéric Wagner. Hierarchical work-stealing. In Pasqua D'Ambra, Mario Guarracino, and Domenico Talia, editors, *Euro-Par 2010 - Parallel Processing*, pages 217–229, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[10] Swann Perarnau and Mitsuhisa Sato. Victim selection and distributed work stealing performance: A case study. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 659–668, 2014.

[11] Gian-Andrea Wetten. Dynamic scheduling in hpc using a distributed data approach. 2022.

[12] Cabezón, R. M., García-Senz, D., and Figueira, J. Sphynx: an accurate density-based sph method for astrophysical applications. *A&A*, 606:A78, 2017.

[13] Danilo Guerrera, Aurélien Cavelan, Rubén M. Cabezón, David Imbert, Jean-Guillaume Piccinali, Ali Mohammed, Lucio Mayer, Darren Reed, and Florina M. Ciorba. Sph-exa: Enhancing the scalability of sph codes via an exascale-ready sph mini-app, 2019.

[14] Aparna Sasidharan and Marc Snir. Miniamr - a miniapp for adaptive mesh refinement. 2016.

[15] Hui Li, Sudarsan Tandri, Michael Stumm, and Kenneth C. Sevcik. Locality and loop scheduling on numa multiprocessors. In *1993 International Conference on Parallel Processing - ICPP'93*, volume 2, pages 140–147, 1993.

[16] C.P. Kruskal and A. Weiss. Allocating independent subtasks on parallel processors. *IEEE Transactions on Software Engineering*, SE-11(10):1001–1016, 1985.

[17] T.H. Tzen and L.M. Ni. Trapezoid self-scheduling: a practical scheduling scheme for parallel compilers. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):87–98, 1993.

[18] Susan Flynn Hummel, Jeanette Schmidt, R. N. Uma, and Joel Wein. Load-sharing in heterogeneous systems via weighted factoring. In *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '96, page 318–328, New York, NY, USA, 1996. Association for Computing Machinery.

[19] Torben Hagerup. Allocating independent tasks to parallel processors: An experimental study. *Journal of Parallel and Distributed Computing*, 47(2):185–197, 1997.

[20] I. Banicescu and V. Velusamy. Performance of scheduling scientific applications with adaptive weighted factoring. In *Parallel and Distributed Processing Symposium, International*, volume 1, pages 791,792,793,794,795,796,797,798,799,800,801, Los Alamitos, CA, USA, apr 2001. IEEE Computer Society.

[21] Yizhuo Wang, Yang Zhang, Yan Su, Xiaojun Wang, Xu Chen, Weixing Ji, and Feng Shi. An adaptive and hierarchical task scheduling scheme for multi-core clusters. *Parallel Computing*, 40(10):611–627, 2014.

[22] Ryusuke Nakashima, Hiroshi Yoritaka, Masahiro Yasugi, Tasuku Hiraishi, and Seiji Umatani. Extending a work-stealing framework with priorities and weights. In *2019 IEEE/ACM 9th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, pages 9–16, 2019.

[23] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The data locality of work stealing. In *Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '00, page 1–12, New York, NY, USA, 2000. Association for Computing Machinery.

[24] Quan Chen and Minyi Guo. Locality-aware work stealing based on online profiling and auto-tuning for multisocket multicore architectures. *ACM Trans. Archit. Code Optim.*, 12(2), jul 2015.

[25] Andi Drebes, Karine Heydemann, Nathalie Drach, Antoniu Pop, and Albert Cohen. Topology-aware and dependence-aware scheduling and memory allocation for task-parallel languages. *ACM Trans. Archit. Code Optim.*, 11(3), aug 2014.

[26] Ioana Banicescu and Susan Flynn Hummel. Balancing processor loads and exploiting data locality in n-body simulations. In *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, Supercomputing '95, page 43–es, New York, NY, USA, 1995. Association for Computing Machinery.

[27] Samuel Russ, Ioana Banicescu, Mark Bilderback, and Sheikh Ghafoor. A fault-tolerant system for balancing the load of data-parallel applications a manuscript submitted for consideration to the distributed systems engineering journal special issue on dependable distributed systems. 01 1999.

[28] Jonas H. Müller Korndörfer, Mario Bielert, Laércio L. Pilla, and Florina M. Ciorba. Mapping matters: Application process mapping on 3-d processor topologies. 2021.

[29] Prof. Dr. Florina M. Ciorba. minihpc: Small but modern mini hpc. https://hpc.dmi.unibas.ch/en/research/minihpc/. Accessed: 2023-08-28.

[30] Benoit B. Mandelbrot. Fractal aspects of the iteration of z $\rightarrow \Lambda$z(1- z) for complex Λ and z. *Annals of the New York Academy of Sciences*, 357(1):249–259, 1980.

A Appendix

A.1 Extended results Pisolver

A.1.1 Comparison of varying workload imbalance

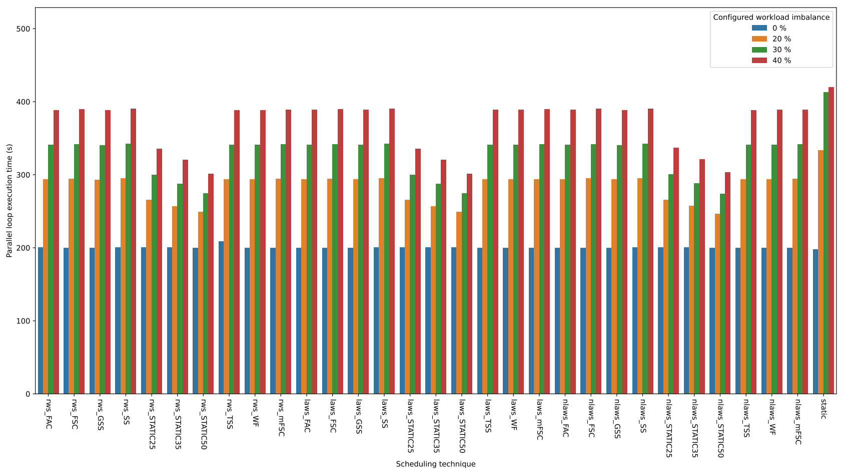


Figure A.1: **Comparison of the parallel execution time about the scheduling techniques used and categorized by the different input workload imbalance:** In the x-axis, we can see all the scheduling techniques employed during execution. The y-axis corresponds to the parallel execution time in seconds. Furthermore, the plot is categorized by the configured workload imbalance of 0%, 20%, 30%, and 40%.

A.1.2 Results with 0% induced workload imbalance

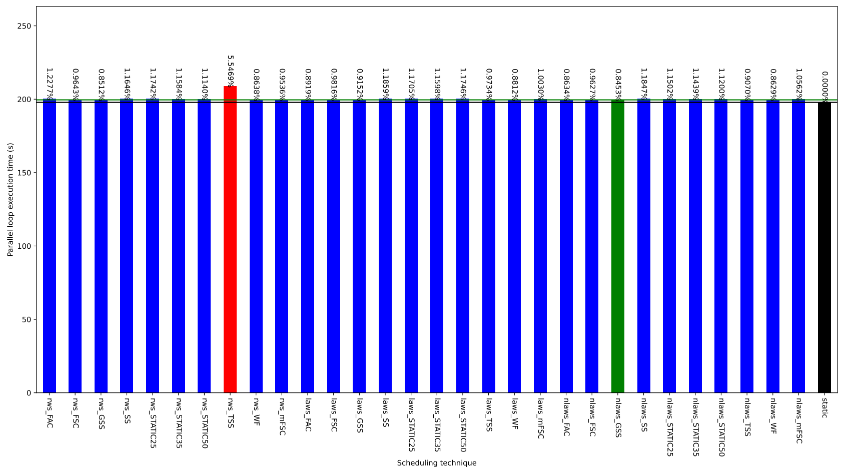


Figure A.2: **Pisolver executed with** 0% **workload imbalance:** On the x-axes, we show the employed scheduling techniques, whereas on the y-axes, the parallel loop execution time is shown. Furthermore, the black color represents the results of the static execution without LB4MPI. The red bar illustrates the worst-performing dynamic scheduling technique. The green bar represents the best-performing scheduling technique. The black labels above the bars are the percentage differences from the static version without LB4MPI.

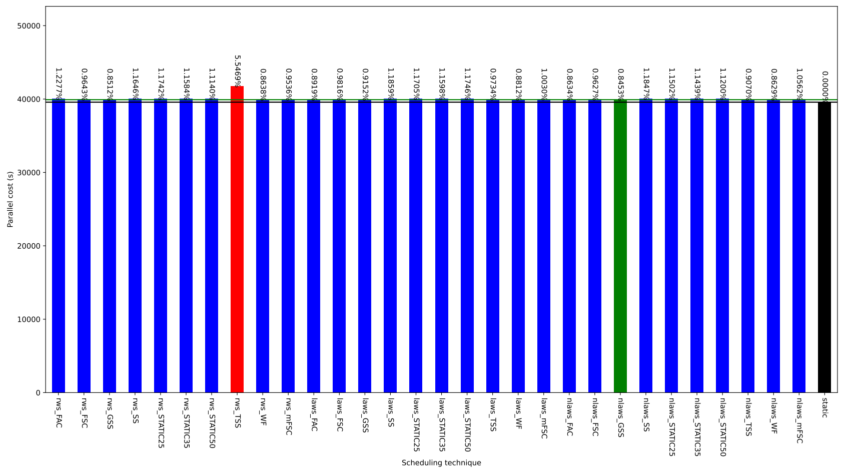


Figure A.3: **Pisolver executed with** 0% **workload imbalance:** On the x-axes, we show the employed scheduling techniques, whereas on the y-axes, the parallel loop execution cost is shown. Furthermore, the black color represents the results of the static execution without LB4MPI. The red bar illustrates the worst-performing dynamic scheduling technique. The green bar represents the best-performing scheduling technique. The black labels above the bars are the percentage differences from the static version without LB4MPI.

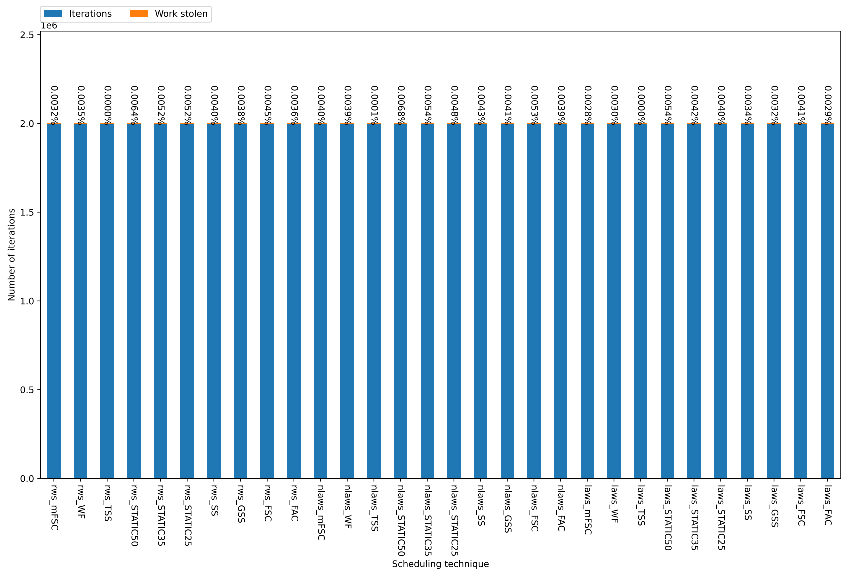


Figure A.4: **Pisolver executed with 0% workload imbalance - Relation of total iterations scheduled and total work stolen:** The y-axes display the total number of iterations performed on average. The x-axes show the employed scheduling techniques. On top of the blue bars, we can see the fraction of work that has been stolen per scheduling technique with the corresponding percentage.

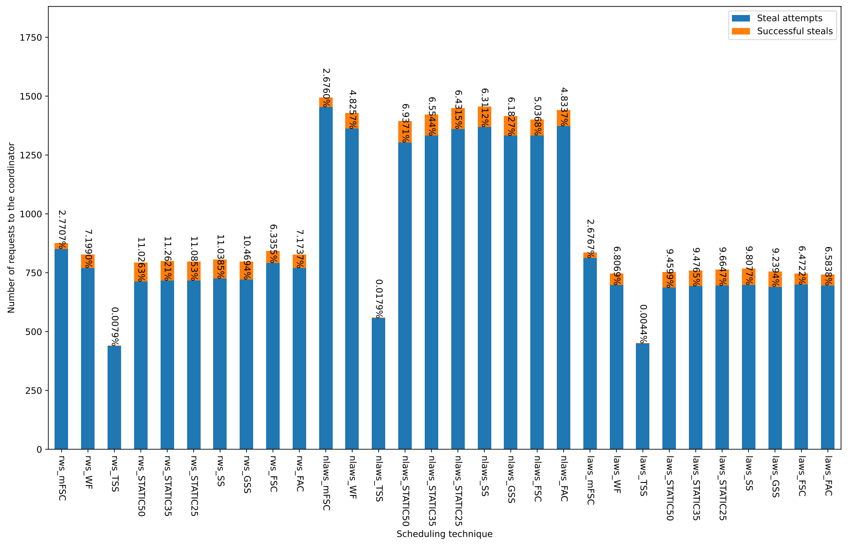


Figure A.5: **Pisolver executed with 0% workload imbalance - Number of steal attempts versus successful steals per scheduling technique:** The fraction of the successful steal requests is given within the orange bars. The x-axis lists the scheduling techniques. The y-axis shows the total number of steal attempts and successful steals labeled by the fraction of successful steal requests.

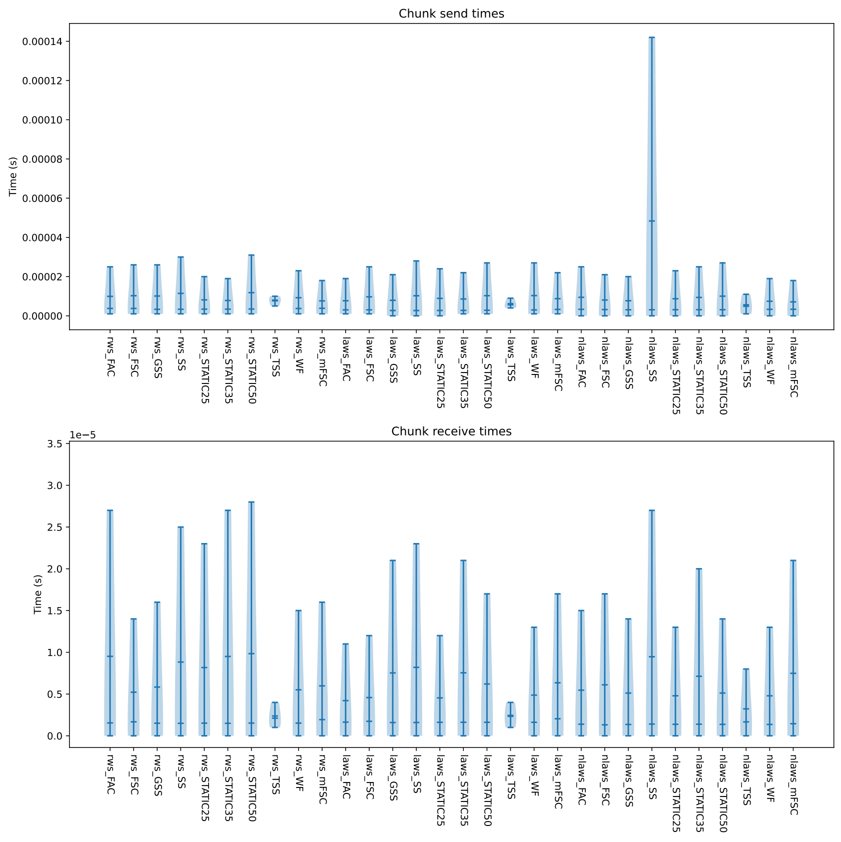


Figure A.6: **Pisolver executed with 0% workload imbalance - Send and receive times of the work per iteration:** Here, a representation of the end and receive times of the work and data to be delivered from the victim to the thief is shown. On the y-axis, the needed time is shown. The scheduling techniques are listed on the x-axis. In blue, one can see the distribution of the minimum, average, and maximum time needed. The light blue area represents the occurrences of time needed.

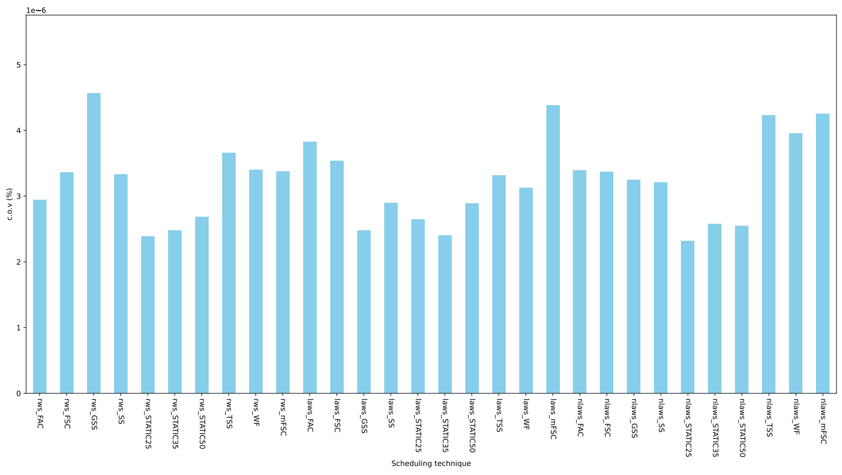


Figure A.7: **Pisolver executed with 0% workload imbalance - Calculated c.o.v of the parallel loop execution times obtained:** On the x-axis, the scheduling techniques are listed. On the y-axis, one can see the c.o.v. of the parallel loop execution time

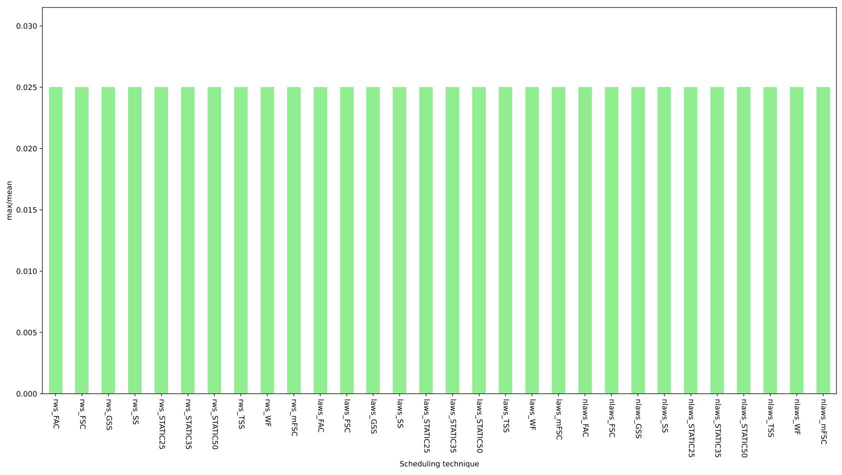


Figure A.8: **Pisolver executed with 0% workload imbalance - Imbalanced factor max/mean** On the x-axis, the scheduling techniques are listed. On the y-axis, one can see the imbalanced factor of the parallel loop execution time

A.1.3 Results with 20% induced workload imbalance

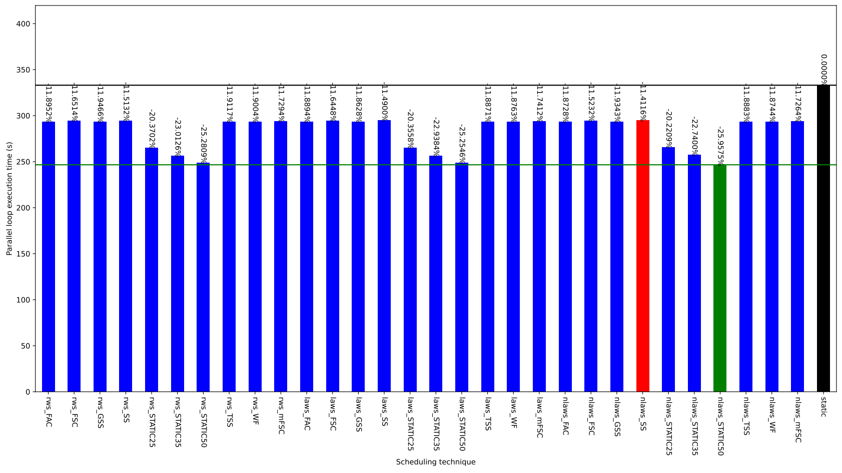


Figure A.9: **Pisolver executed with 20% workload imbalance:** On the x-axes, we show the employed scheduling techniques, whereas on the y-axes, the parallel loop execution time is shown. Furthermore, the black color represents the results of the static execution without LB4MPI. The red bar illustrates the worst-performing dynamic scheduling technique. The green bar represents the best-performing scheduling technique. The black labels above the bars are the percentage differences from the static version without LB4MPI.

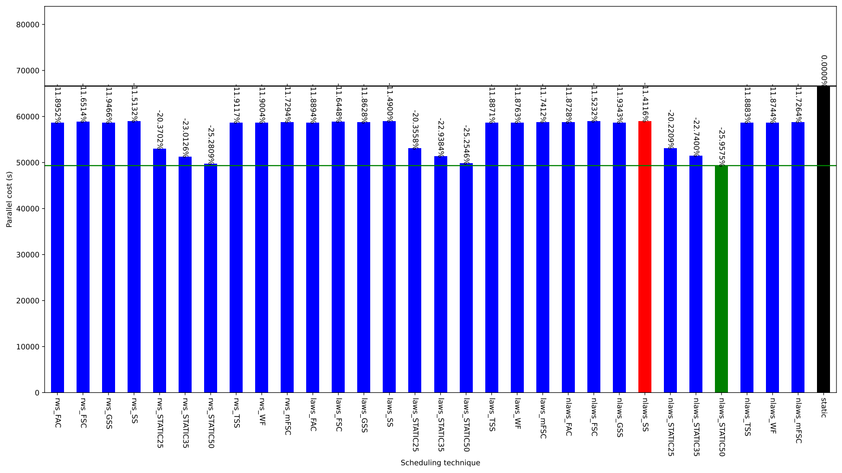


Figure A.10: **Pisolver executed with 20% workload imbalance:** On the x-axes, we show the employed scheduling techniques, whereas on the y-axes, the parallel loop execution cost is shown. Furthermore, the black color represents the results of the static execution without LB4MPI. The red bar illustrates the worst-performing dynamic scheduling technique. The green bar represents the best-performing scheduling technique. The black labels above the bars are the percentage differences from the static version without LB4MPI.

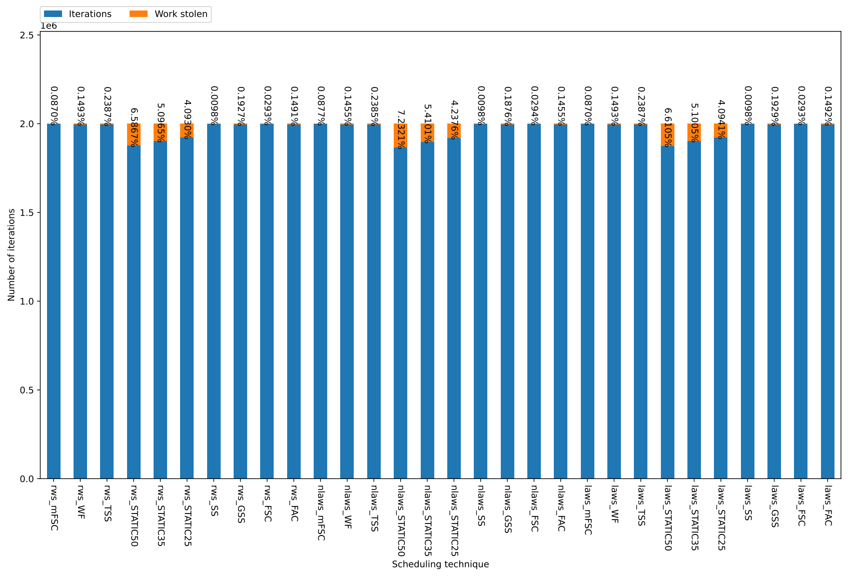


Figure A.11: **Pisolver executed with 20% workload imbalance - Relation of total iterations scheduled and total work stolen:** The y-axes display the total number of iterations performed on average. The x-axes show the employed scheduling techniques. On top of the blue bars, we can see the fraction of work that has been stolen per scheduling technique with the corresponding percentage.

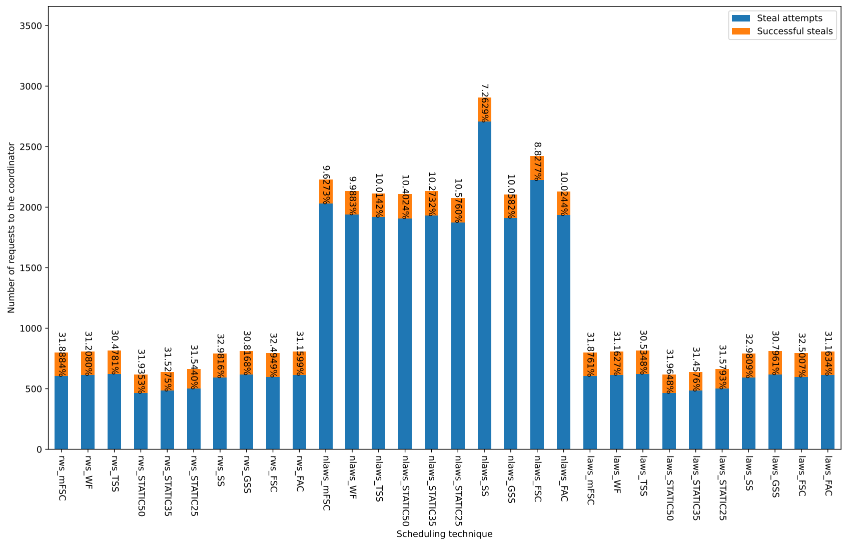


Figure A.12: **Pisolver executed with 20% workload imbalance - Number of steal attempts versus successful steals per scheduling technique:** The fraction of the successful steal requests is given within the orange bars. The x-axis lists the scheduling techniques. The y-axis shows the total number of steal attempts and successful steals labeled by the fraction of successful steal requests.

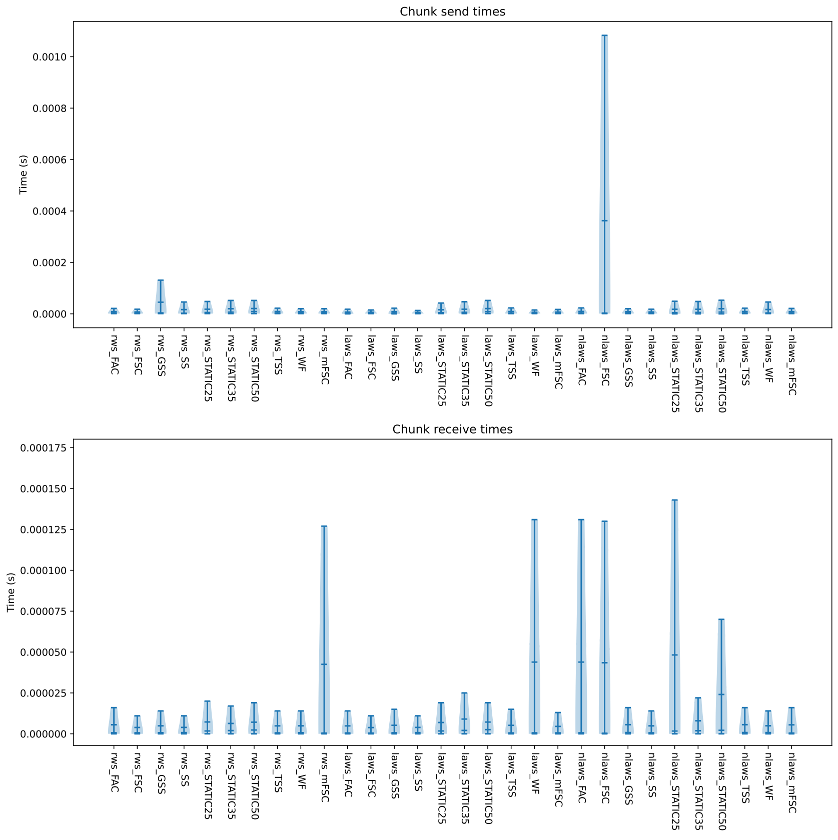


Figure A.13: **Pisolver executed with 20% workload imbalance - Send and receive times of the work per iteration:** Here, a representation of the end and receive times of the work and data to be delivered from the victim to the thief is shown. On the y-axis, the needed time is shown. The scheduling techniques are listed on the x-axis. In blue, one can see the distribution of the minimum, average, and maximum time needed. The light blue area represents the occurrences of time needed.

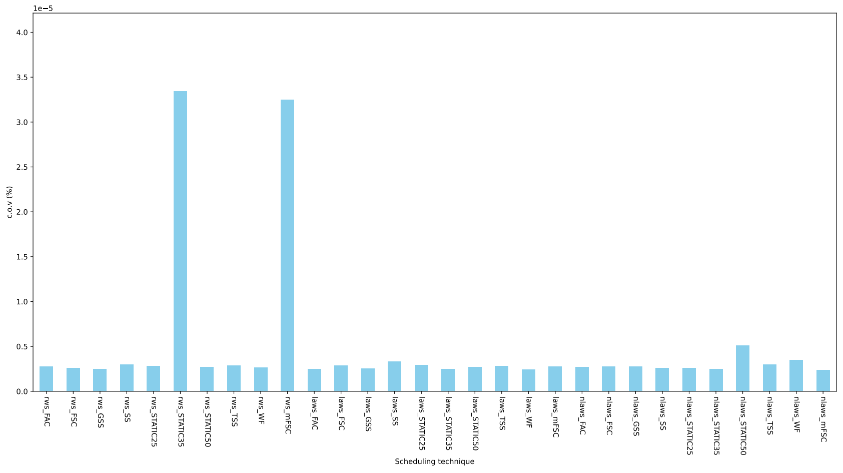


Figure A.14: **Pisolver executed with 20% workload imbalance - Calculated c.o.v of the parallel loop execution times obtained:** On the x-axis, the scheduling techniques are listed. On the y-axis, one can see the c.o.v. of the parallel loop execution time

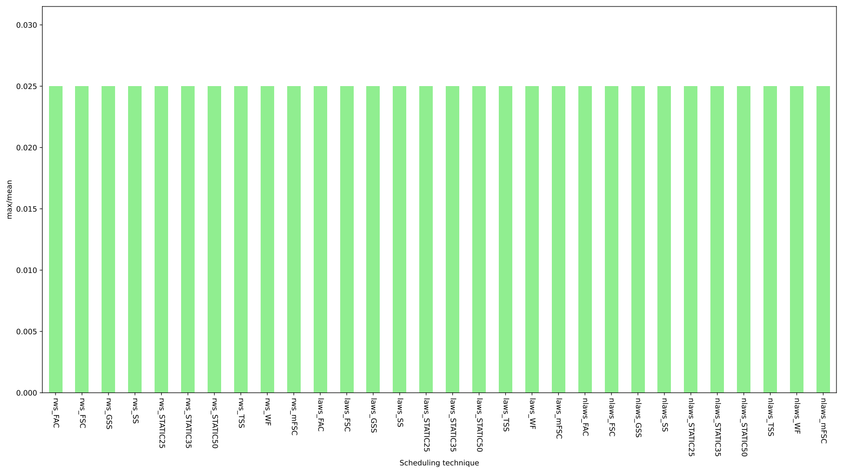


Figure A.15: **Pisolver executed with 20% workload imbalance - Imbalanced factor max/mean** On the x-axis, the scheduling techniques are listed. On the y-axis, one can see the imbalanced factor of the parallel loop execution time

A.1.4 Results with 30% induced work imbalance

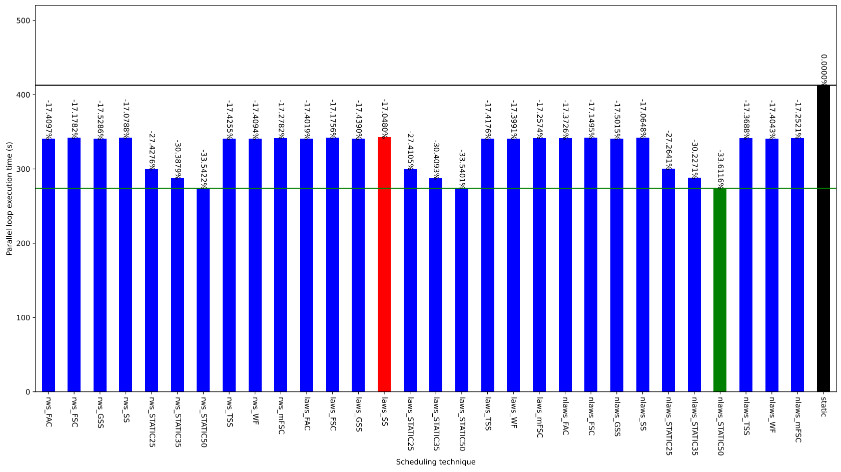


Figure A.16: **Pisolver executed with 30% workload imbalance:** On the x-axes, we show the employed scheduling techniques, whereas on the y-axes, the parallel loop execution time is shown. Furthermore, the black color represents the results of the static execution without LB4MPI. The red bar illustrates the worst-performing dynamic scheduling technique. The green bar represents the best-performing scheduling technique. The black labels above the bars are the percentage differences from the static version without LB4MPI.

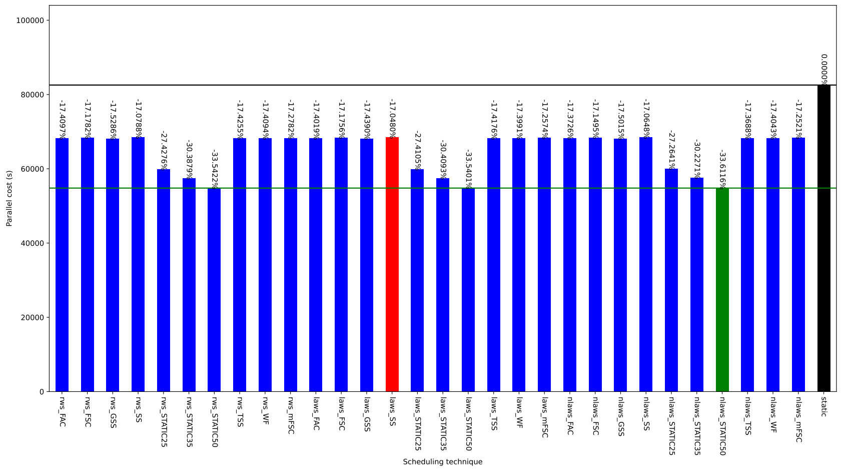


Figure A.17: **Pisolver executed with 30% workload imbalance:** On the x-axes, we show the employed scheduling techniques, whereas on the y-axes, the parallel loop execution cost is shown. Furthermore, the black color represents the results of the static execution without LB4MPI. The red bar illustrates the worst-performing dynamic scheduling technique. The green bar represents the best-performing scheduling technique. The black labels above the bars are the percentage differences from the static version without LB4MPI.

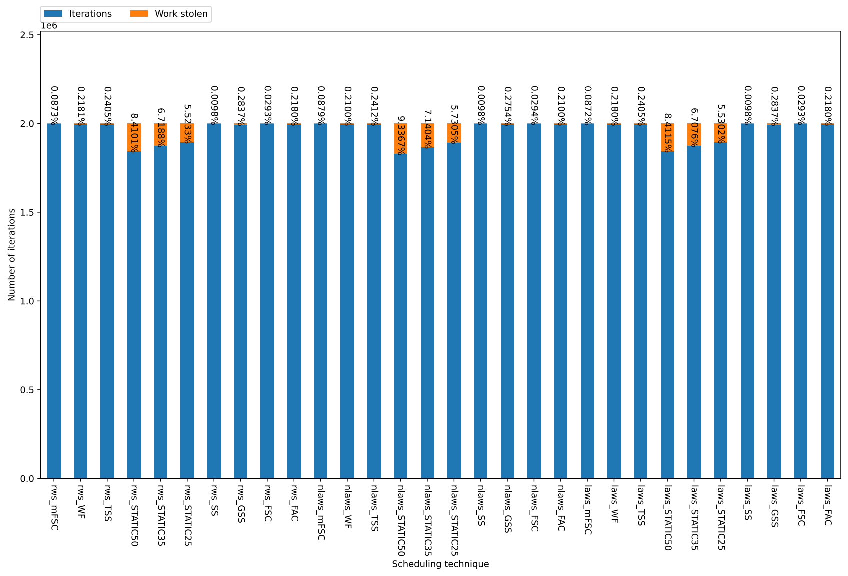


Figure A.18: **Pisolver executed with 30% workload imbalance - Relation of total iterations scheduled and total work stolen:** The y-axes display the total number of iterations performed on average. The x-axes show the employed scheduling techniques. On top of the blue bars, we can see the fraction of work that has been stolen per scheduling technique with the corresponding percentage.

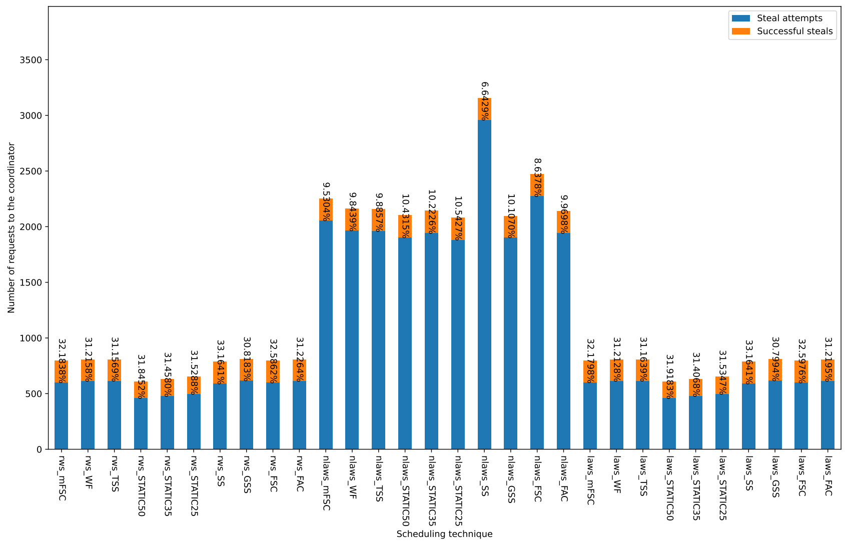


Figure A.19: **Pisolver executed with 30% workload imbalance - Number of steal attempts versus successful steals per scheduling technique:** The fraction of the successful steal requests is given within the orange bars. The x-axis lists the scheduling techniques. The y-axis shows the total number of steal attempts and successful steals labeled by the fraction of successful steal requests.

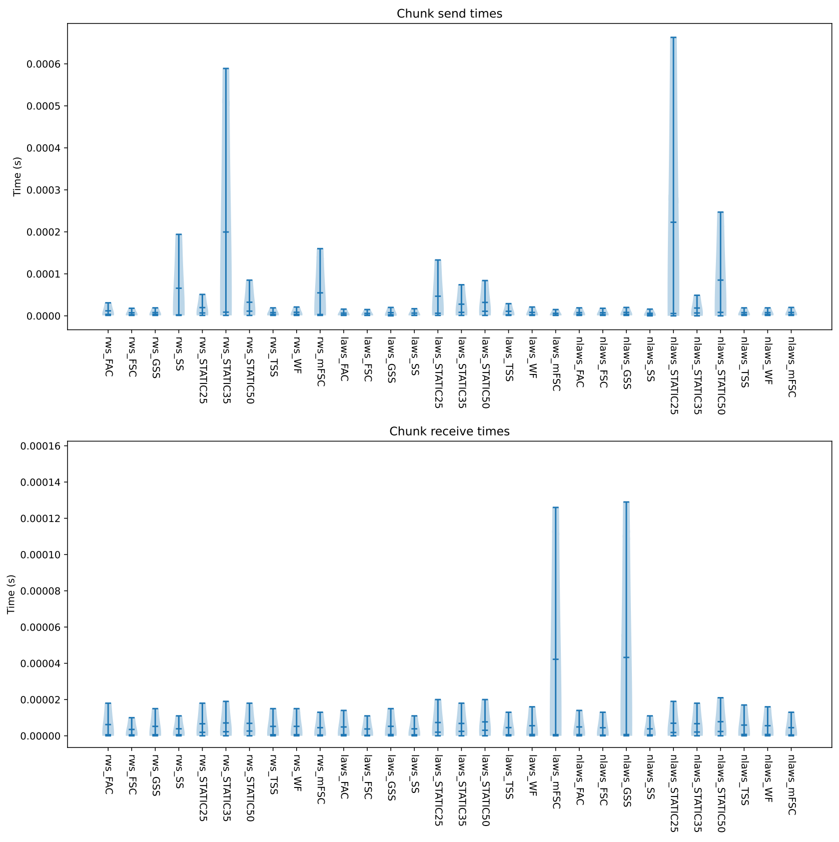


Figure A.20: **Pisolver executed with 30% workload imbalance - Send and receive times of the work per iteration:** Here, a representation of the end and receive times of the work and data to be delivered from the victim to the thief is shown. On the y-axis, the needed time is shown. The scheduling techniques are listed on the x-axis. In blue, one can see the distribution of the minimum, average, and maximum time needed. The light blue area represents the occurrences of time needed.

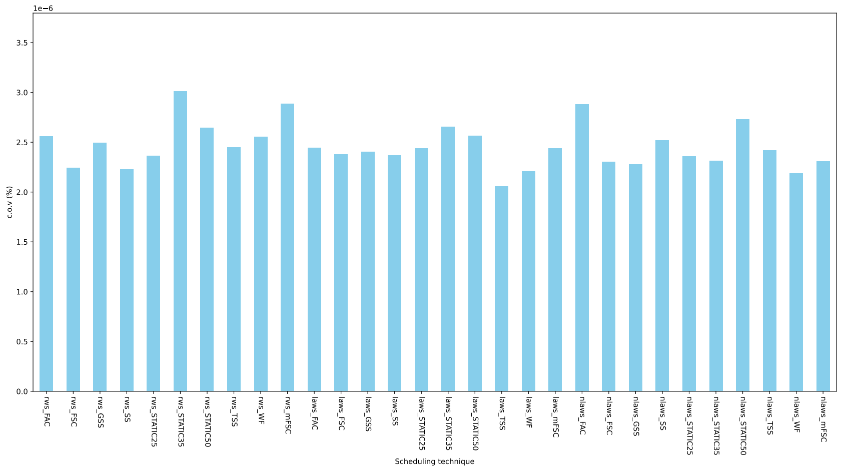


Figure A.21: **Pisolver executed with** 30% **workload imbalance - Calculated c.o.v of the parallel loop execution times obtained:** On the x-axis, the scheduling techniques are listed. On the y-axis, one can see the c.o.v. of the parallel loop execution time

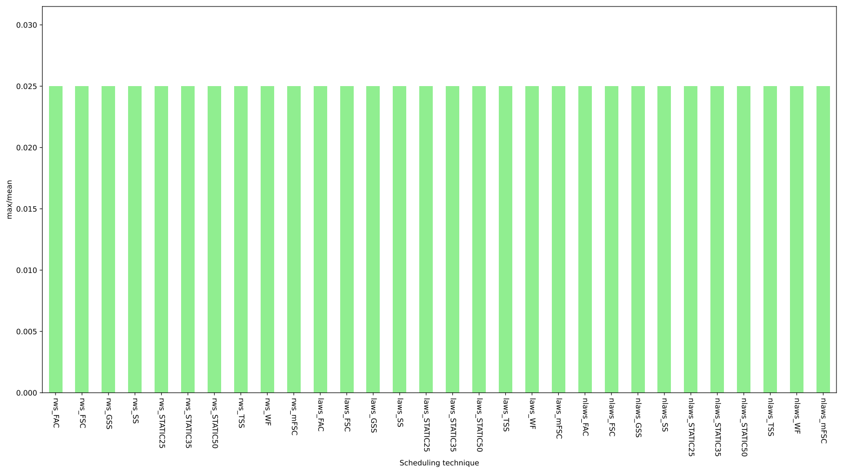


Figure A.22: **Pisolver executed with** 30% **workload imbalance - Imbalanced factor max/mean** On the x-axis, the scheduling techniques are listed. On the y-axis, one can see the imbalanced factor of the parallel loop execution time

A.1.5 Results with 40% induced workload imbalance

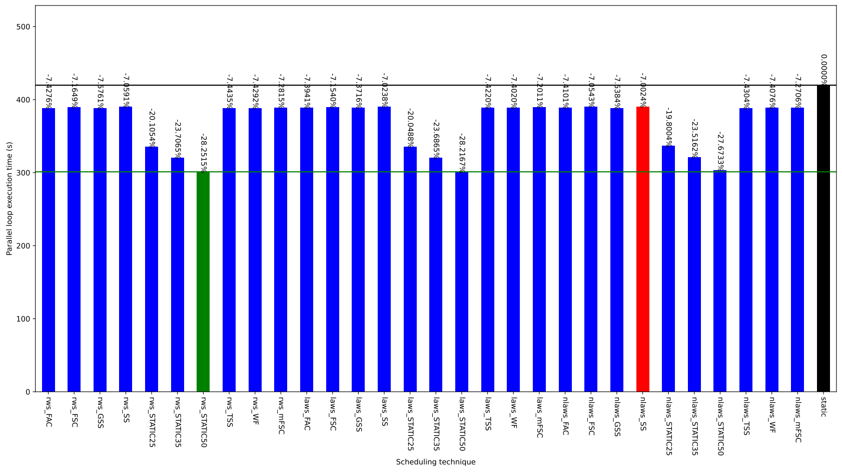


Figure A.23: **Pisolver executed with** 40% **workload imbalance:** On the x-axes, we show the employed scheduling techniques, whereas on the y-axes, the parallel loop execution time is shown. Furthermore, the black color represents the results of the static execution without LB4MPI. The red bar illustrates the worst-performing dynamic scheduling technique. The green bar represents the best-performing scheduling technique. The black labels above the bars are the percentage differences from the static version without LB4MPI.

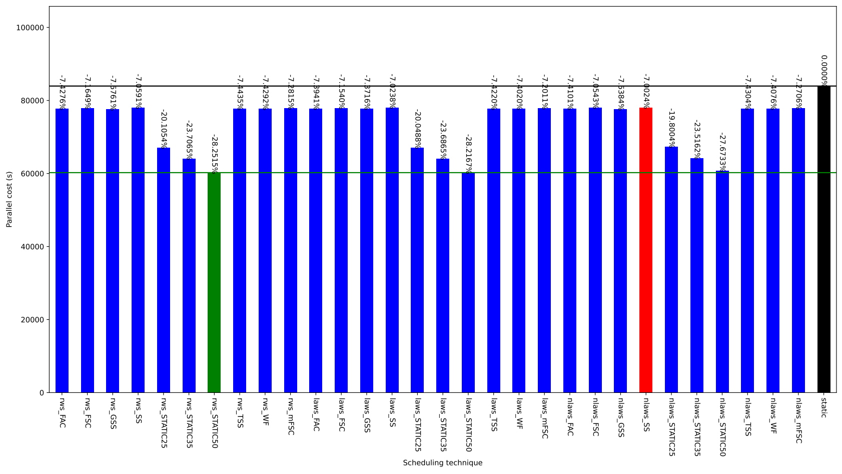


Figure A.24: **Pisolver executed with** 40% **workload imbalance:** On the x-axes, we show the employed scheduling techniques, whereas on the y-axes, the parallel loop execution cost is shown. Furthermore, the black color represents the results of the static execution without LB4MPI. The red bar illustrates the worst-performing dynamic scheduling technique. The green bar represents the best-performing scheduling technique. The black labels above the bars are the percentage differences from the static version without LB4MPI.

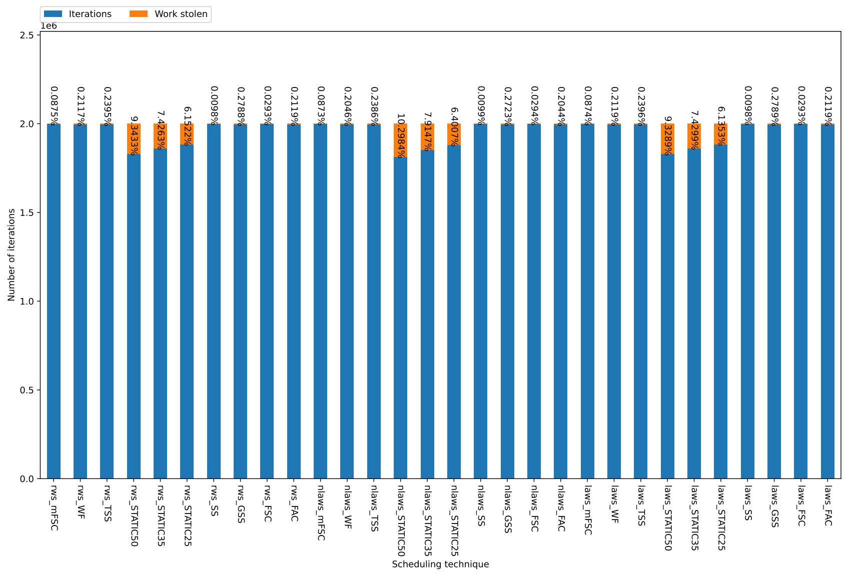


Figure A.25: **Pisolver executed with 40% workload imbalance - Relation of total iterations scheduled and total work stolen:** The y-axes display the total number of iterations performed on average. The x-axes show the employed scheduling techniques. On top of the blue bars, we can see the fraction of work that has been stolen per scheduling technique with the corresponding percentage.

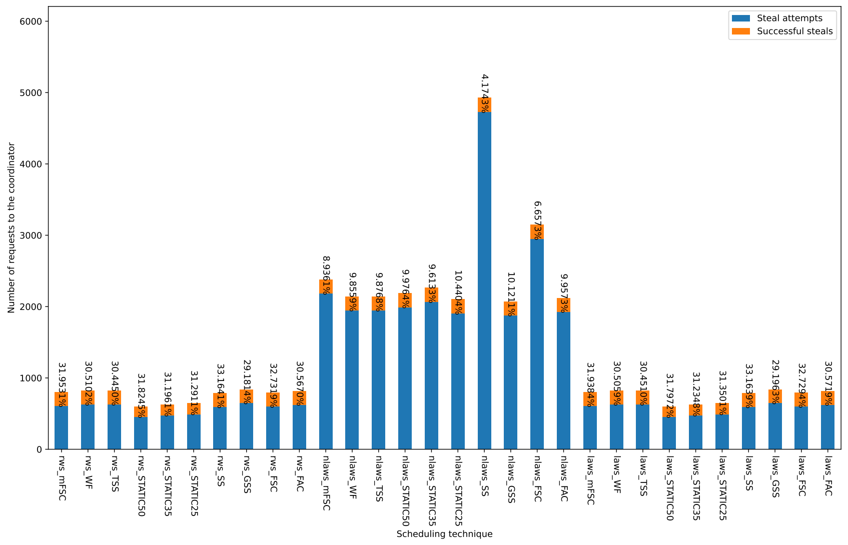


Figure A.26: **Pisolver executed with 40% workload imbalance - Number of steal attempts versus successful steals per scheduling technique:** The fraction of the successful steal requests is given within the orange bars. The x-axis lists the scheduling techniques. The y-axis shows the total number of steal attempts and successful steals labeled by the fraction of successful steal requests.

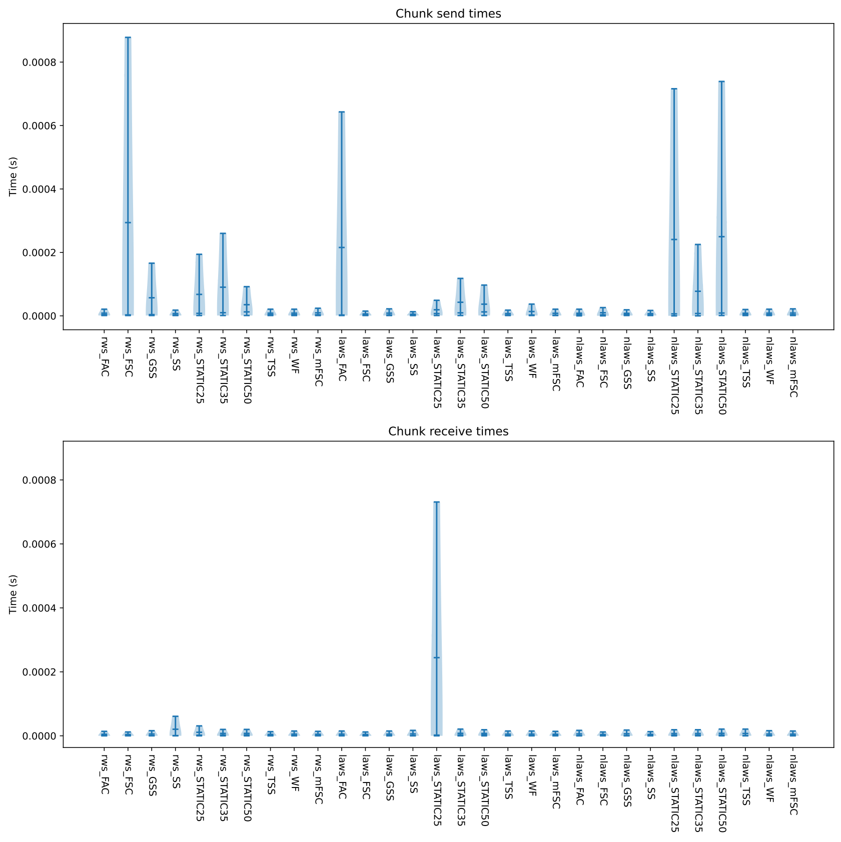


Figure A.27: **Pisolver executed with 40% workload imbalance - Send and receive times of the work per iteration:** Here, a representation of the end and receive times of the work and data to be delivered from the victim to the thief is shown. On the y-axis, the needed time is shown. The scheduling techniques are listed on the x-axis. In blue, one can see the distribution of the minimum, average, and maximum time needed. The light blue area represents the occurrences of time needed.

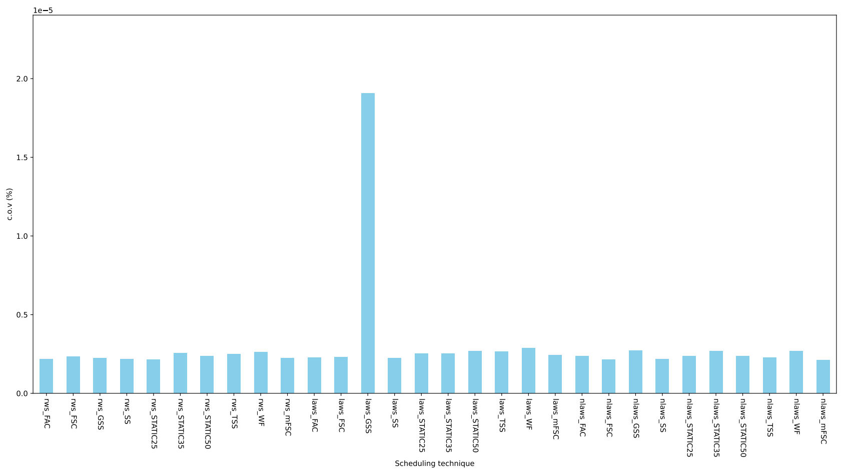


Figure A.28: **Pisolver executed with 40% workload imbalance - Calculated c.o.v of the parallel loop execution times obtained:** On the x-axis, the scheduling techniques are listed. On the y-axis, one can see the c.o.v. of the parallel loop execution time

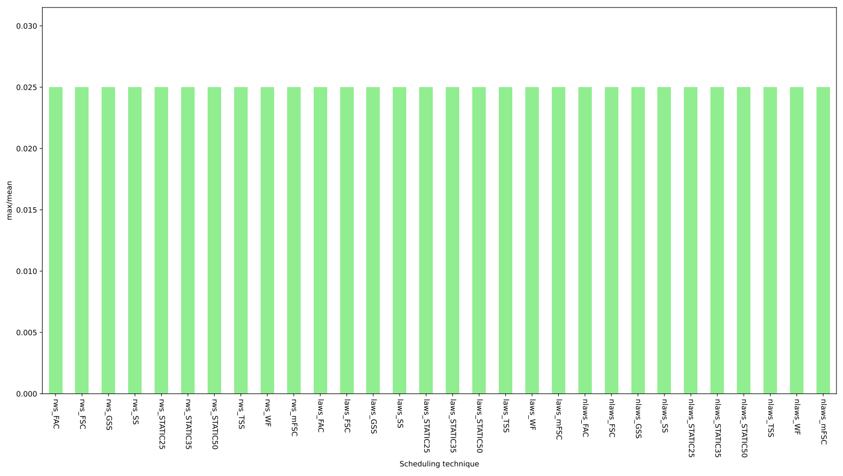


Figure A.29: **Pisolver executed with 40% workload imbalance - Imbalanced factor max/mean** On the x-axis, the scheduling techniques are listed. On the y-axis, one can see the imbalanced factor of the parallel loop execution time

A.2 Extended results Mandelbrot

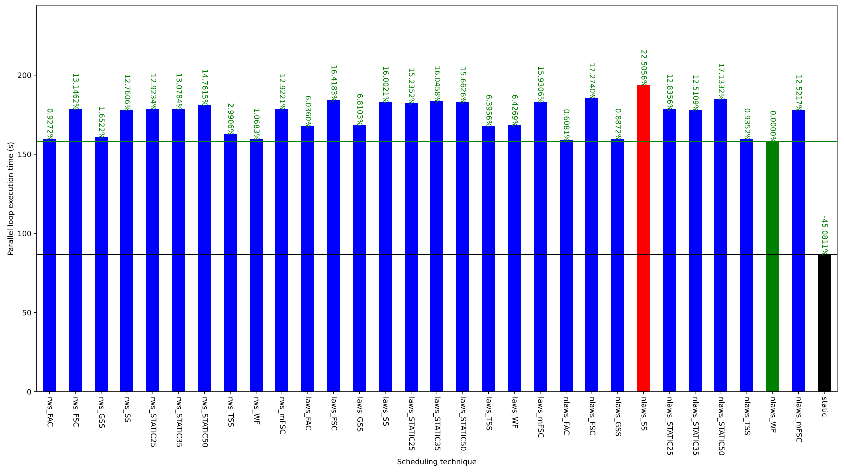


Figure A.30: **Mandelbrot:** On the x-axes, we show the employed scheduling techniques, whereas on the y-axes, the parallel loop execution time is shown. Furthermore, the black color represents the results of the static execution without LB4MPI. The red bar illustrates the worst-performing dynamic scheduling technique. The green bar represents the best-performing scheduling technique. The black labels above the bars are the percentage differences from the static version without LB4MPI.

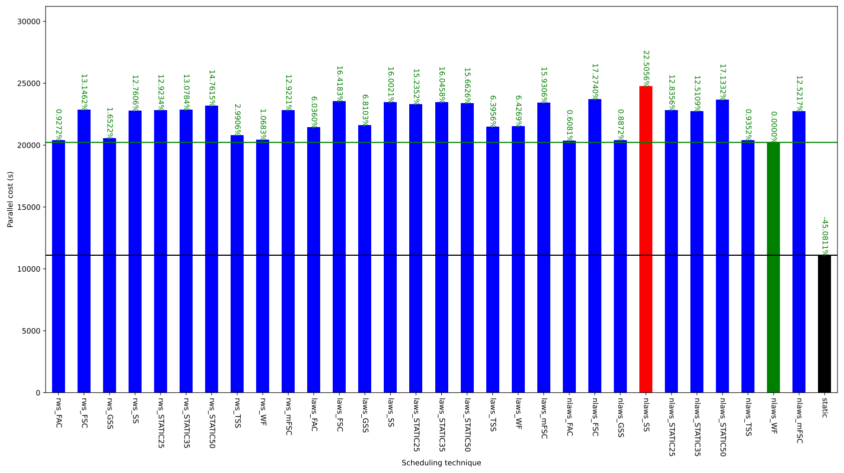


Figure A.31: **Mandelbrot:** On the x-axes, we show the employed scheduling techniques, whereas on the y-axes, the parallel loop execution cost is shown. Furthermore, the black color represents the results of the static execution without LB4MPI. The red bar illustrates the worst-performing dynamic scheduling technique. The green bar represents the best-performing scheduling technique. The black labels above the bars are the percentage differences from the static version without LB4MPI.

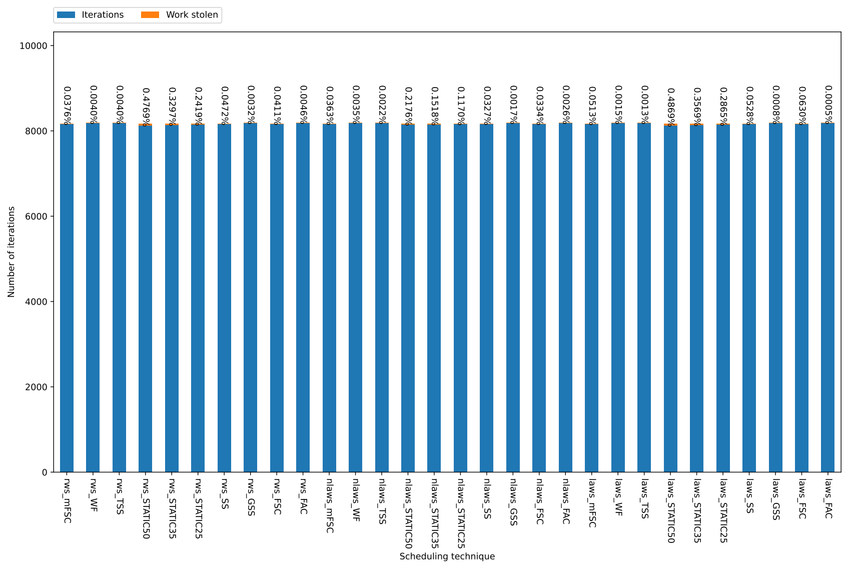


Figure A.32: **Mandelbrot - Relation of total iterations scheduled and total work stolen:** The y-axes display the total number of iterations performed on average. The x-axes show the employed scheduling techniques. On top of the blue bars, we can see the fraction of work that has been stolen per scheduling technique with the corresponding percentage.

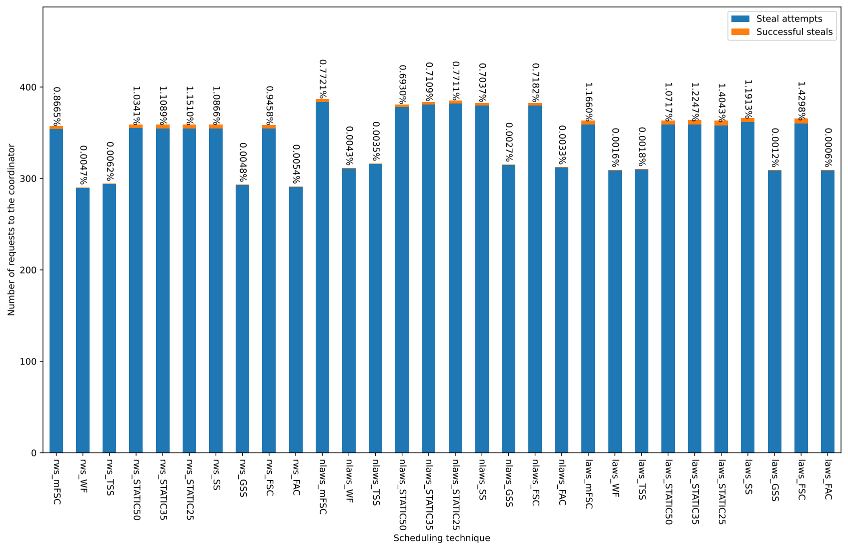


Figure A.33: **Mandelbrot - Number of steal attempts versus successful steals per scheduling technique:** The fraction of the successful steal requests is given within the orange bars. The x-axis lists the scheduling techniques. The y-axis shows the total number of steal attempts and successful steals labeled by the fraction of successful steal requests.

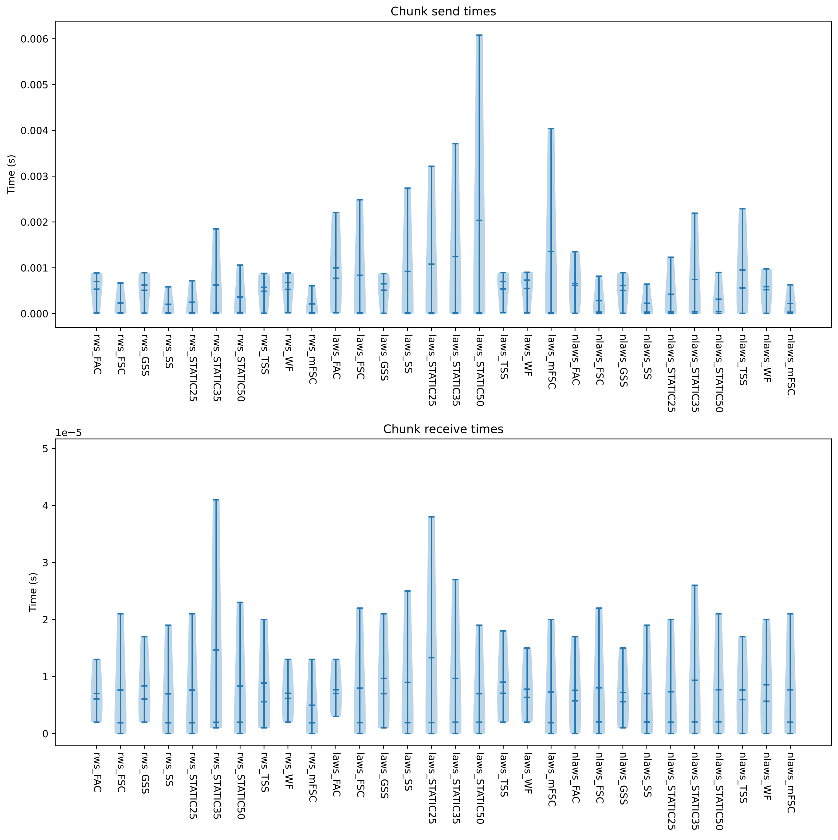


Figure A.34: **Mandelbrot - Send and receive times of the work per iteration:** Here, a representation of the end and receive times of the work and data to be delivered from the victim to the thief is shown. On the y-axis, the needed time is shown. The scheduling techniques are listed on the x-axis. In blue, one can see the distribution of the minimum, average, and maximum time needed. The light blue area represents the occurrences of time needed.

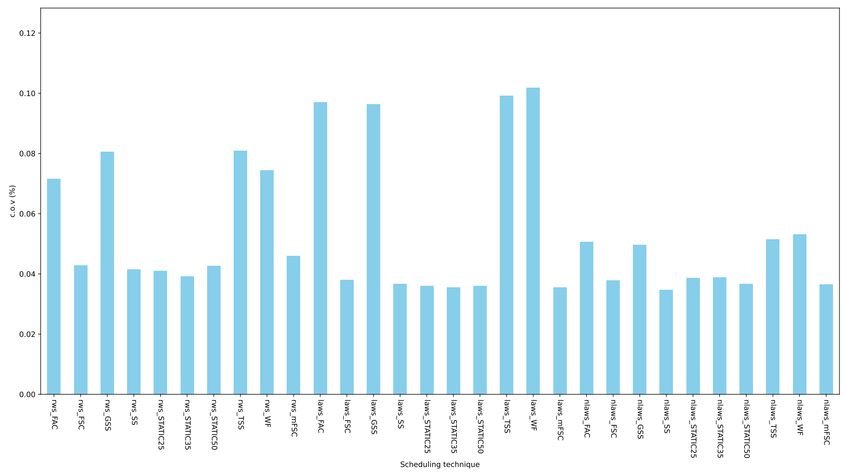


Figure A.35: **Mandelbrot - Calculated c.o.v of the parallel loop execution times obtained:** On the x-axis, the scheduling techniques are listed. On the y-axis, one can see the c.o.v. of the parallel loop execution time

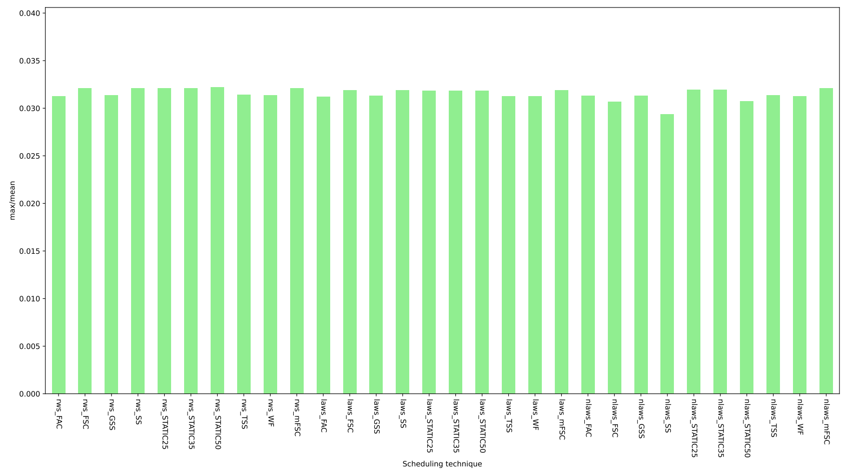


Figure A.36: **Mandelbrot - Imbalanced factor max/mean** On the x-axis, the scheduling techniques are listed. On the y-axis, one can see the imbalanced factor of the parallel loop execution time

A.3 Extended results SPH-EXA (Sedov)

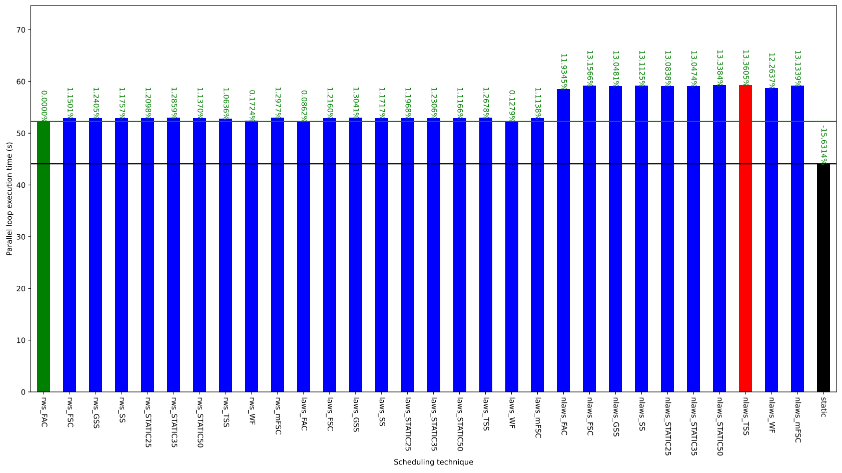


Figure A.37: **SPH-EXA Sedov:** On the x-axes, we show the employed scheduling techniques, whereas on the y-axes, the parallel loop execution time is shown. Furthermore, the black color represents the results of the static execution without LB4MPI. The red bar illustrates the worst-performing dynamic scheduling technique. The green bar represents the best-performing scheduling technique. The black labels above the bars are the percentage differences from the static version without LB4MPI.

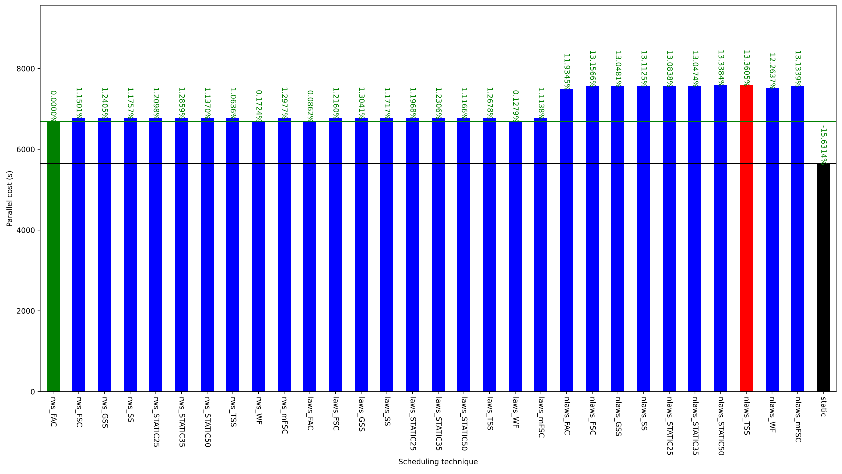


Figure A.38: **SPH-EXA Sedov:** On the x-axes, we show the employed scheduling techniques, whereas on the y-axes, the parallel loop execution cost is shown. Furthermore, the black color represents the results of the static execution without LB4MPI. The red bar illustrates the worst-performing dynamic scheduling technique. The green bar represents the best-performing scheduling technique. The black labels above the bars are the percentage differences from the static version without LB4MPI.

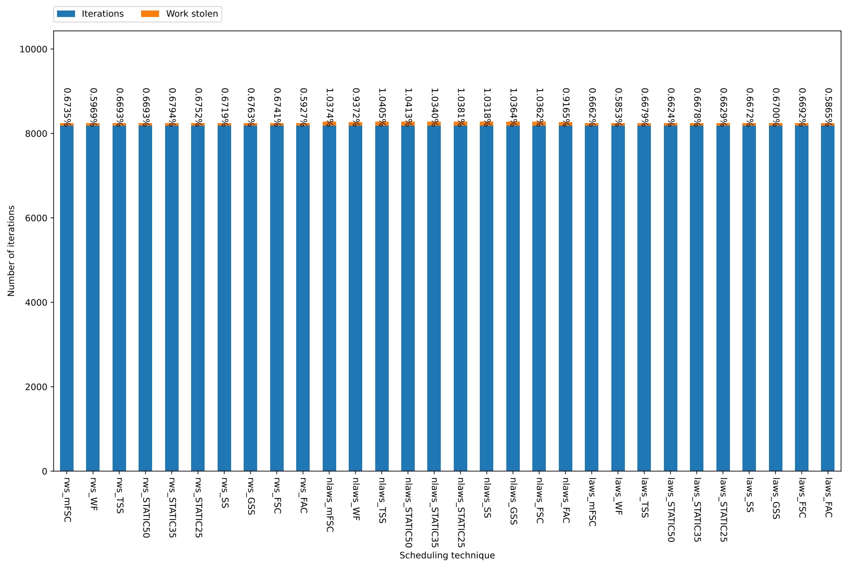


Figure A.39: **SPH-EXA Sedov - Relation of total iterations scheduled and total work stolen:** The y-axes display the total number of iterations performed on average. The x-axes show the employed scheduling techniques. On top of the blue bars, we can see the fraction of work that has been stolen per scheduling technique with the corresponding percentage.

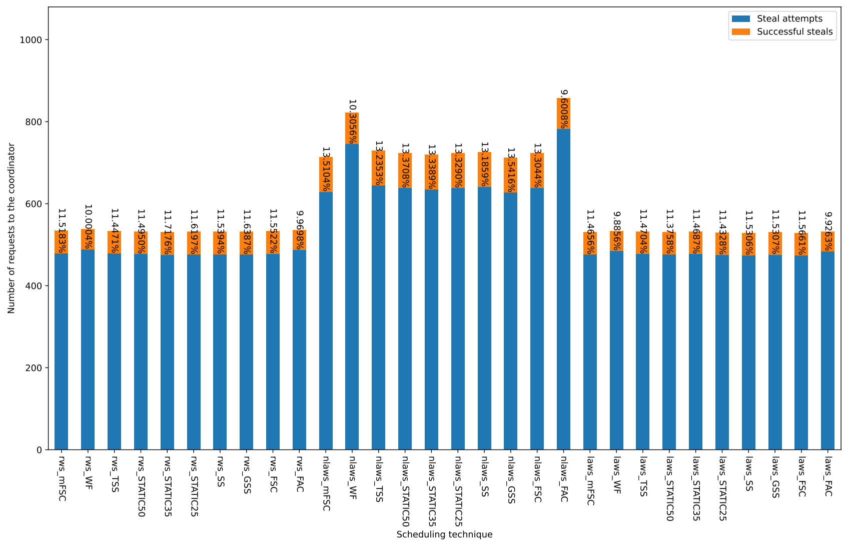


Figure A.40: **SPH-EXA Sedov - Number of steal attempts versus successful steals per scheduling technique:** The fraction of the successful steal requests is given within the orange bars. The x-axis lists the scheduling techniques. The y-axis shows the total number of steal attempts and successful steals labeled by the fraction of successful steal requests.

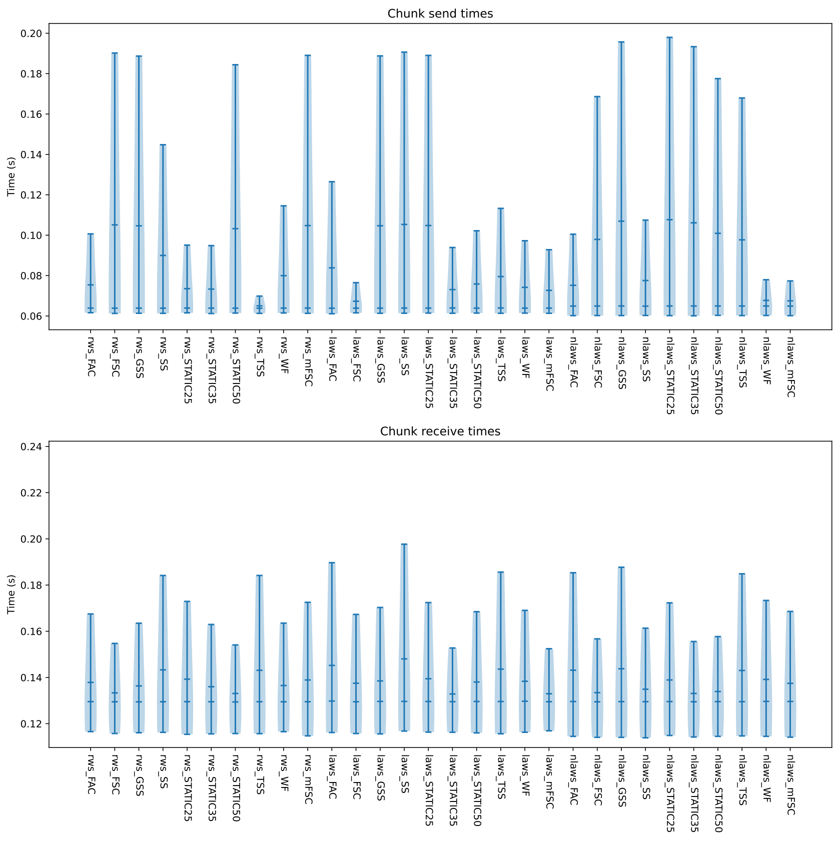


Figure A.41: **SPH-EXA Sedov - Send and receive times of the work per iteration:** Here, a representation of the end and receive times of the work and data to be delivered from the victim to the thief is shown. On the y-axis, the needed time is shown. The scheduling techniques are listed on the x-axis. In blue, one can see the distribution of the minimum, average, and maximum time needed. The light blue area represents the occurrences of time needed.

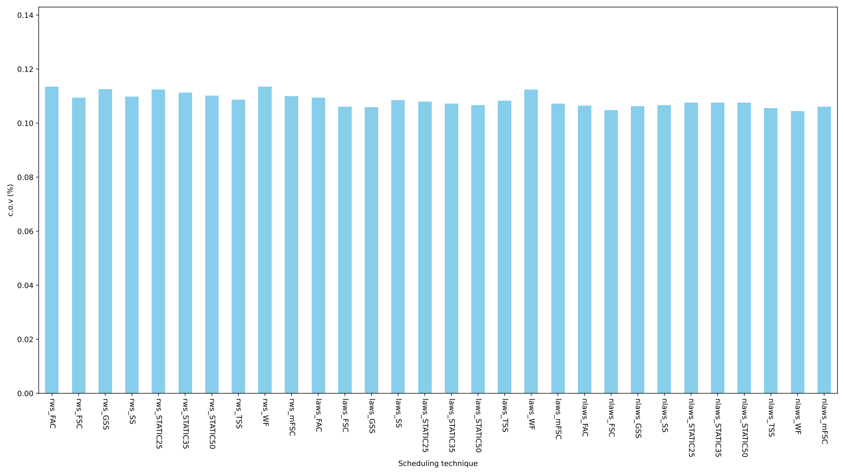


Figure A.42: **SPH-EXA Sedov - Calculated c.o.v of the parallel loop execution times obtained:** On the x-axis, the scheduling techniques are listed. On the y-axis, one can see the c.o.v. of the parallel loop execution time

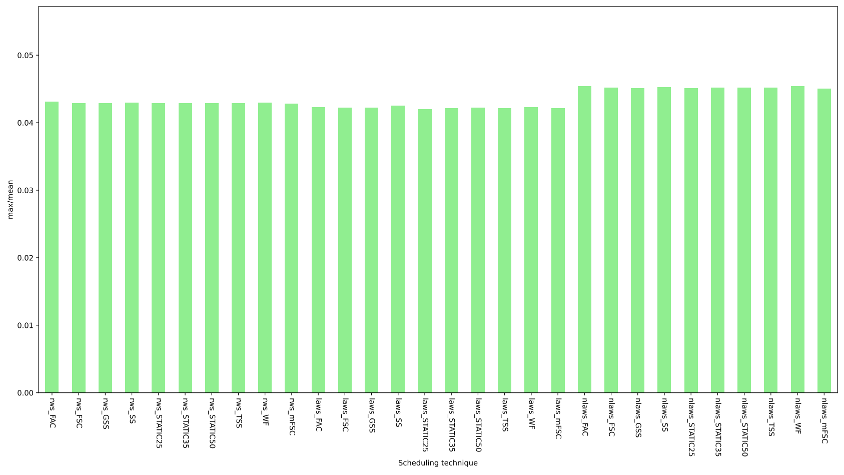


Figure A.43: **SPH-EXA Sedov - Imbalanced factor max/mean** On the x-axis, the scheduling techniques are listed. On the y-axis, one can see the imbalanced factor of the parallel loop execution time

A.4 Extended results miniAMR

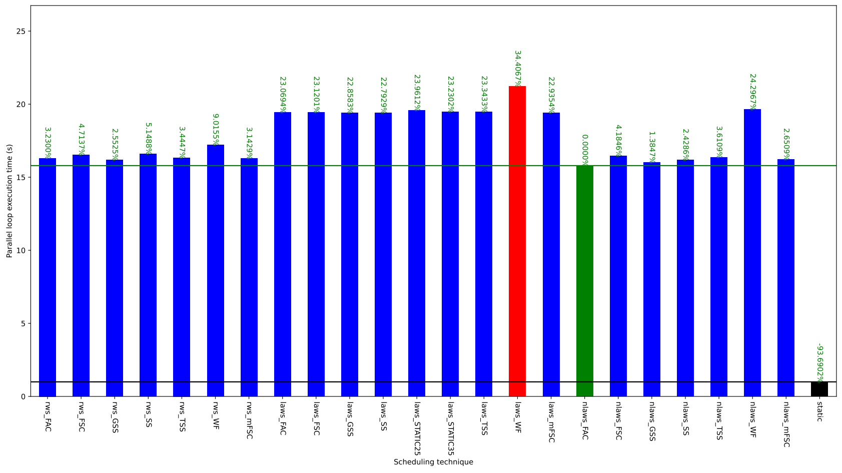


Figure A.44: **miniAMR:** On the x-axes, we show the employed scheduling techniques, whereas on the y-axes, the parallel loop execution time is shown. Furthermore, the black color represents the results of the static execution without LB4MPI. The red bar illustrates the worst-performing dynamic scheduling technique. The green bar represents the best-performing scheduling technique. The black labels above the bars are the percentage differences from the static version without LB4MPI.

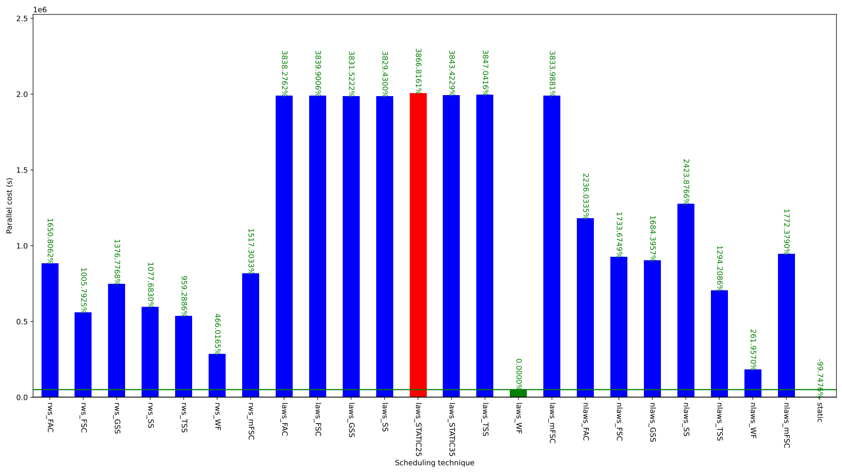


Figure A.45: **miniAMR:** On the x-axes, we show the employed scheduling techniques, whereas on the y-axes, the parallel loop execution cost is shown. Furthermore, the black color represents the results of the static execution without LB4MPI. The red bar illustrates the worst-performing dynamic scheduling technique. The green bar represents the best-performing scheduling technique. The black labels above the bars are the percentage differences from the static version without LB4MPI.

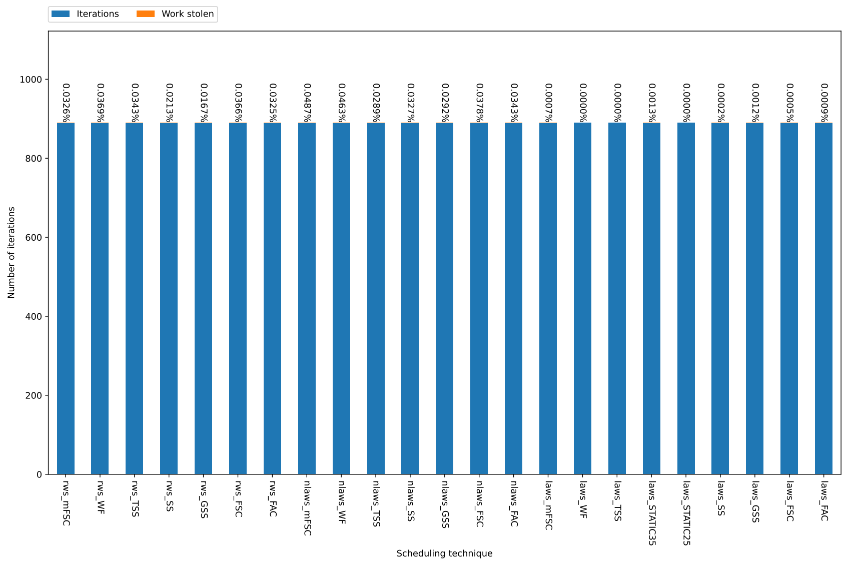


Figure A.46: **miniAMR - Relation of total iterations scheduled and total work stolen:** The y-axes display the total number of iterations performed on average. The x-axes show the employed scheduling techniques. On top of the blue bars, we can see the fraction of work that has been stolen per scheduling technique with the corresponding percentage.

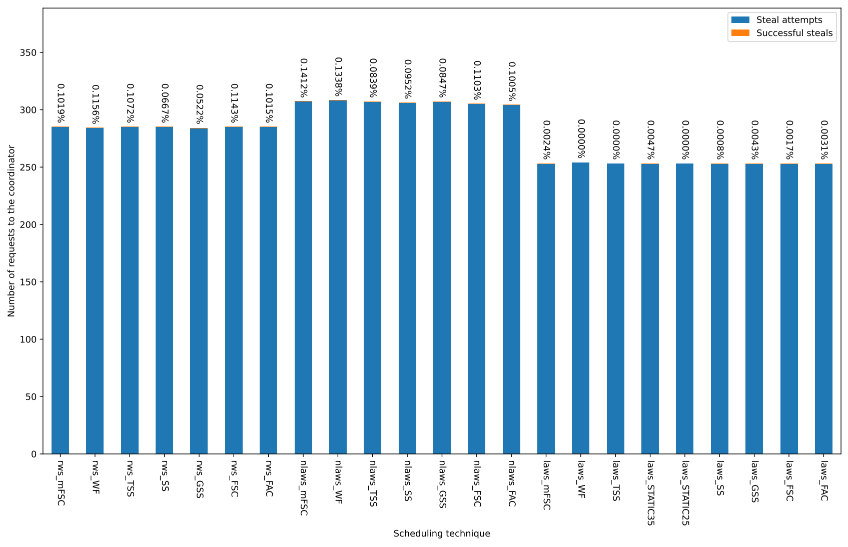


Figure A.47: **miniAMR - Number of steal attempts versus successful steals per scheduling technique:** The fraction of the successful steal requests is given within the orange bars. The x-axis lists the scheduling techniques. The y-axis shows the total number of steal attempts and successful steals labeled by the fraction of successful steal requests.

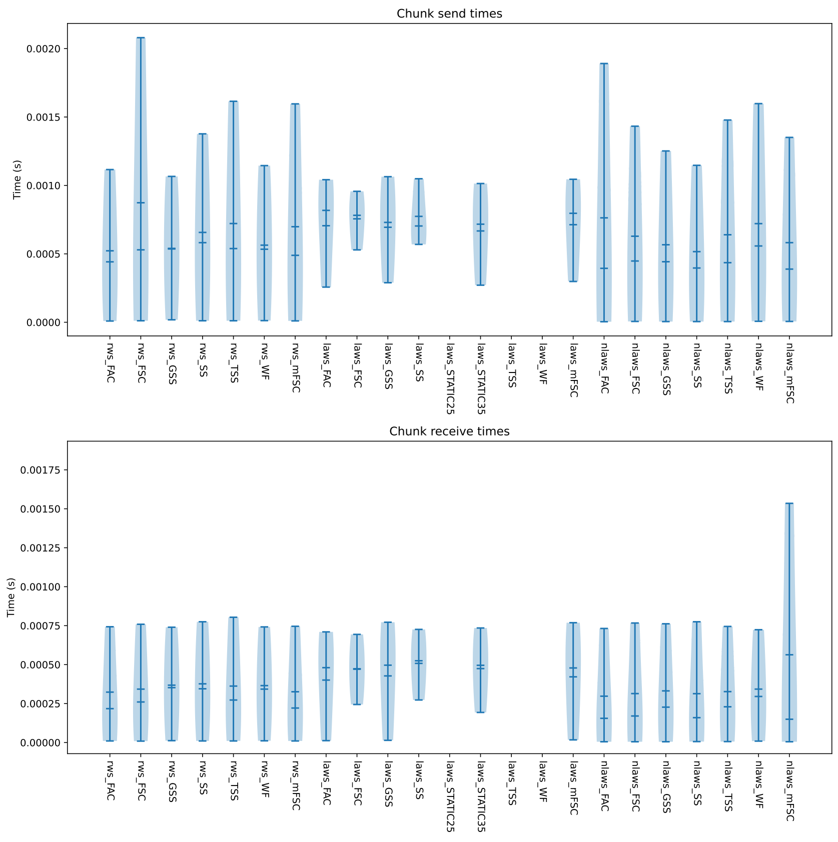


Figure A.48: **miniAMR - Send and receive times of the work per iteration:** Here, a representation of the end and receive times of the work and data to be delivered from the victim to the thief is shown. On the y-axis, the needed time is shown. The scheduling techniques are listed on the x-axis. In blue, one can see the distribution of the minimum, average, and maximum time needed. The light blue area represents the occurrences of time needed.

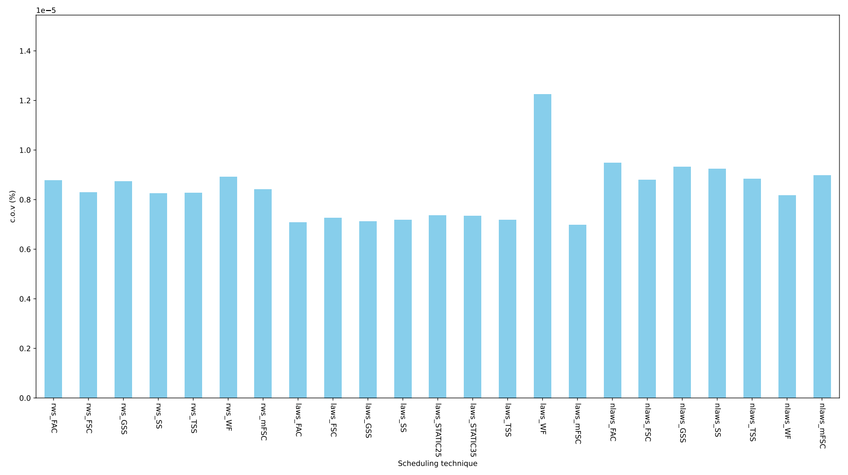


Figure A.49: **miniAMR - Calculated c.o.v of the parallel loop execution times obtained:** On the x-axis, the scheduling techniques are listed. On the y-axis, one can see the c.o.v. of the parallel loop execution time

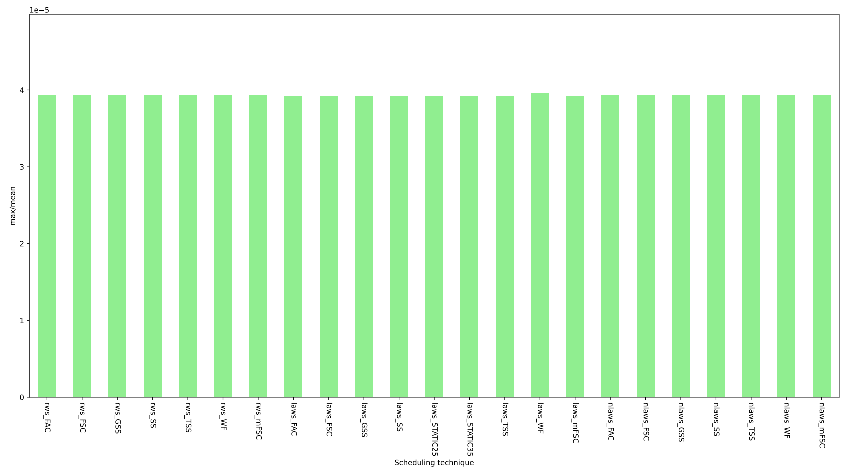


Figure A.50: **miniAMR - Imbalanced factor max/mean** On the x-axis, the scheduling techniques are listed. On the y-axis, one can see the imbalanced factor of the parallel loop execution time

University of Basel

Faculty of Science

Declaration on Scientific Integrity
(including a Declaration on Plagiarism and Fraud)
Translation from German original

Title of Thesis:	Dynamic Loop Self-scheduling with Distributed Data for MPI Applications
Name Assessor:	Prof. Dr. Florina M. Ciorba
Name Student:	Nderim Shatri
Matriculation No.:	HS16-062-234

With my signature I declare that this submission is my own work and that I have fully acknowledged the assistance received in completing this work and that it contains no material that has not been formally acknowledged. I have mentioned all source materials used and have cited these in accordance with recognised scientific rules.

Place, Date: Basel, den 31.08.2023 Student: [illegible]

Will this work, or parts of it, be published?

○ No

[✓ Yes. With my signature I confirm that I agree to a publication of the work (print/digital) in the library, on the research database of the University of Basel and/or on the document server of the department. Likewise, I agree to the bibliographic reference in the catalog SLSP (Swiss Library Service Platform). (cross out as applicable)]

Publication as of:

Place, Date: Basel, den 31.08.2023 Student: [illegible]

Place, Date: Assessor:

Please enclose a completed and signed copy of this declaration in your Bachelor's or Master's thesis