CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Scheduling of Signal Processing Tasks in a Computer Cluster

Investigating methods for scheduling dynamic data processing loads in a cluster environment using a hybrid programming model

Master's thesis in Computer science and engineering

Jonathan Eksberg, David Lidevi

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2023

Master's thesis 2023

Scheduling of Signal Processing Tasks in a Computer Cluster

Investigating methods for scheduling dynamic data processing loads in a cluster environment using a hybrid programming model

Jonathan Eksberg, David Lidevi

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

Scheduling of Signal Processing Tasks in a Computer Cluster
Investigating methods for scheduling dynamic data processing loads in a cluster environment using a hybrid programming model
Jonathan Eksberg, David Lidevi

Supervisor: Nikela Papadopoulou, Computer Science and Engineering
Advisor: Anders Åhlander, Anton Karlsson, Saab AB
Examiner: Miquel Pericas, Computer Science and Engineering

Master's Thesis 2023
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2023

Scheduling of Signal Processing Tasks in a Computer Cluster
Investigating methods for scheduling dynamic data processing loads in a cluster environment using a hybrid programming model
Jonathan Eksberg, David Lidevi
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Newer generations of radar signal processing systems have increasingly higher computational demands. This thesis aimed to investigate the impact of scheduling techniques, load balancing approaches, and parallel programming models in real-time signal processing applications utilizing a cluster environment.
To do so, a representative scenario was created, which was intended to resemble a real application scenario. Several scheduling algorithms were implemented in an iterative manner and evaluated in the representative scenario using a homogenous cluster system consisting of four nodes.
Some key findings involved the potential of dynamically varying the number of workers and their resources to better adapt to the dynamic environment of radar signal processing. These techniques could also reduce contention for memory resources and the negative impacts of simultaneous multithreading for execution times. By allocating sparingly used backup workers that ran tasks at an easier difficulty, additional increases in overall performance and robustness could be established.
The results indicate that a scheduler implemented in cluster-oriented programming models can utilize the system resources to meet the increased performance demands of signal processing systems. However, challenges such as development overhead, process allocation, and adaptation to cluster architecture must be considered for optimal performance in an arbitrary cluster environment.

Keywords: Scheduling, load balancing, real-time, signal processing, radar technology, parallelism, computer cluster, hybrid programming model, MPI, OpenMP,

Acknowledgements

First of all, we would like to thank our advisors at Saab, Anton Karlsson and Anders Åhlander. The constant day-to-day support in all matters and their involvement in the thesis work have been tremendously helpful for us and the project. We would also like to thank Jonas Lindgren, who ensured we had an excellent onboarding process and access to necessary equipment.

We also would like to express our great appreciation for our supervisor at Chalmers, Nikela Papadopoulou. The great feedback and insights we received from every meeting have been incredibly helpful for the project. Great appreciation also goes to our examiner, Miquel Pericas, whose feedback and guidance have driven the project forward.

Lastly, we would also like to say thank you to everyone else who has supported us and given us feedback during the project. The process has been much more joyful because of that.

Jonathan Eksberg, David Lidevi, Gothenburg, 2023-06-21

Contents

Glossary

AESA	*Active Electronically Scanned Array*
CFAR	*Constant False Alarm Ratio*
DBF	*Digital Beam Forming*
DD Scheduler	*Dual-Difficulty Scheduler*
DF	*Doppler Filtering*
DRA Scheduler	*Dynamic Resource Allocating Scheduler*
EDF	*Earliest-deadline-first algorithm*
EW Scheduler	*Emergency Workers Scheduler*
FT	The *Fast Time* dimension of a data block
INTI	*Integration Interval*
MIMD	*Multiple Instruction, Multiple Data*
MISD	*Multiple Instruction, Single Data*
MPD	*Medium Pulse Doppler*
MPI	*Message Passing Interface*
MPMD	*Multiple Program Multiple Data*
MTI	*Moving Target Indicator*
NUMA	*Non-Uniform Memory Access*
OpenMP	*Open Multi-Processing*
PC	*Pulse Compression*
PRF	*Pulse Repititon Frequency*
PRI	*Pulse Repititon Interval*
SIMD	*Single Instruction, Multiple Data*
SISD	*Single Instruction, Single Data*
SMP	*Symmetric Multiprocessing* or *Shared-Memory Multiprocessing*
SMT	*Simultaneous Multithreading*
SP-chain	*Signal Processing Chain*
SPMD	*Single Program Multiple Data*
ST	The *Slow Time* dimension of a data block
WI Scheduler	*Worker Initiated Scheduler*

List of Figures

List of Tables

1

Introduction

The capacity to collect large amounts of data in modern systems is rapidly increasing. However, having large amounts of data will only help if the data can be processed efficiently and within a reasonable time frame. Despite modern computing systems having access to multiple processors, fast clock speeds, and large memory sizes, serious time delays might occur if data is processed inefficiently. These delays could slow down a whole system and create bottlenecks, which is not sustainable when fast processing time is crucial.
A field in computer science that mainly involves utilizing the resources of a computer as effectively as possible is *scheduling*. A good scheduler should assign tasks to workers such that specific system-dependent goals are achieved. These goals could be, for example, maximizing throughput or meeting deadlines. Cleverly implemented scheduling can lead to significantly more performant data processing than a poorly implemented scheduler. However, the design of the scheduler is mainly dependent on the system's end goal and the nature of the tasks the system needs to process. That is partly why a universally optimal scheduler is not conceivable. It should be noted that a scheduler that operates effectively in environments with homogeneous tasks may not perform optimally when faced with tasks that exhibit a wide variation in execution time. This can result in the system failing to meet its desired performance goals.

1.1 Problem Statement

The primary stakeholder for this project was *Saab AB*. Their interest in this research originates from the challenging applications of a specific type of radar, namely an *Active Electronically Scanned Array* (**AESA**) radar. These multi-element radar systems collect large amounts of data, which need to be processed with heavy and, during specific radar modes, highly dynamic signal processing. The required computational performance has, and will continue to, increase tremendously compared to the computational demands before the newer generations of AESA-based systems. The AESA-based systems are used in different products from Saab, and the technology can be used in, e.g., aircraft, naval radars, and ground vehicles. These systems often support multi-functional processing. However, the physical computing resources inside, e.g., an aircraft, are quite constrained.

1.2 Purpose

Consequently, the purpose of this thesis was to investigate scheduling and load-balancing methods for dynamic signal processing tasks in a cluster environment. Due to the constrained set of computing resources, it was of great interest to research how to distribute these computational loads effectively to satisfy system performance requirements.
The impact of this work was derived from the analyses of methods in a specific problem area. The overall aim was to investigate the effectiveness of various methods to distribute tasks in this application scenario and to provide insights into their strengths and weaknesses. The target environment has a large real-time data inflow, and thus essential parameters of researched methods were throughput, task latency, and the ratio of met task deadlines. The contribution of this thesis lies in the investigation of effective scheduling methods in cluster real-time signal processing applications, with the added constraints such a scenario opposes. As the target environment was a computer cluster, specific tools and programming models were examined in order to make good use of the available resources.

1.3 Objectives

The purpose of the thesis can be summarized into several objectives:

1. How does the choice of scheduling technique impact the overall system latency and load balancing of a real-time signal processing application?
2. How can a scheduling approach adapt to varying and unpredictable workload conditions in real-time signal processing applications?
3. What are the possibilities and challenges associated with developing a scheduler for signal processing applications, utilizing cluster-oriented programming models?

By considering these objectives, this thesis' goal was to provide insights and recommendations for the development of efficient schedulers tailored to real-time signal processing applications.

1.4 Scope

There are many parameters and scenarios to consider in a real-application scenario for an AESA-based system. Therefore, a reasonable scope has been narrowed down, which suits the inherently limited time plan for a master thesis.

1.4.1 Goals and Challenges

This thesis involved an algorithmic approach at a high abstraction level and focused on high-performance computing at a lower abstraction level. The goal was to process

a large real-time inflow of data blocks in a cluster environment. More concretely, the primary focus points for a target system were:

- Maximize throughput
- Meet task deadlines
- Load balancing and effective use of resources
- Manage dynamic processing loads
- Minimize task latency
- Scalability analyzes

These characteristics were approached from a scheduling perspective on the cluster level, where properties of the data, the signal processing algorithms, and the cluster environment were crucial parameters in the research.

The tasks varied regarding, for instance, data dimensions and computational loads. Moreover, the computational load for a data block varies depending on its content. This means that, beyond the normal restrictions of general scheduling, several more aspects had to be considered when different methods were researched. Furthermore, the processing loads are highly dynamic, which means that the computational requirements for a task may change during run-time.
Lastly, the researched system had to be designed concerning scalability regarding processing resources. The desired outcome from the stakeholders was to find effective and practical solutions for distributing demanding and varying signal processing chains onto several processing units in a system. In addition, their interest lay in analyzing how performance in terms of throughput and meeting deadlines behaved if additional hardware were added.

The goal is that the analyses should be carried out in a lab environment, where the environment is representative of how a real-application environment could be structured.

1.4.2 Limitations

The target system was a *homogeneous* computer cluster, meaning that all processing nodes have equal processing units and memory. This project focused on a specific homogeneous target system, and as such, the implementations of researched methods were limited to a single environment.

Due to the limited time scope that a master thesis imposes, limitations on the actual signal processing needed to be established. Throughout the project, the focus revolved around a specific sequence of signal processing filters. Though this chain was representative, there exist many other sequences in a real application scenario, and they each have different system demands. To still be able to simulate a multitude of differently demanding sequences, certain increases in computational demand for this single sequence were modeled.

Another final limitation was the direct or indirect use of classified information, technologies, and data. This means that neither of these details can be portrayed in a published master thesis. This caused some delays in the project due to uncertainties regarding what was allowed to be shared publically.

2

Background

Signal processing and radar technology are information-heavy fields, and this section starts by providing relevant background knowledge within this field. This section also presents foundational concepts and research related to the thesis problem area. The presented topics have heavily influenced the project to reach the goals and meet the challenges mentioned in section 1.4.1. It is assumed that the reader has a relevant background in computer science or similar.

2.1 Radar

Radar is an acronym for *radio detection and ranging* and is mainly used for finding the range (distance), azimuth (angle), and velocity of various objects. It does so with the help of a transmitting antenna and a receiving antenna. Commonly, one single antenna is used for both transmitting and receiving signals [1].
The transmitter sends electromagnetic waves that bounce on objects and return to the receiver. The distance to an object can be calculated using the formula in Eq. 2.1 [1].

$$R = \frac{c \cdot T}{2} \tag{2.1}$$

R	:	The distance to the object	m
c	:	The speed of the light	m/s
T	:	The time from transmitting a pulse to receiving it	s

Another essential aspect to keep in mind is the detectability of objects. Since an electromagnetic wave has to hit the object in order for it to be detected, smaller objects are generally more difficult to detect. More precisely, the formula for finding the power of the echo is given in Eq. 2.2 [1].

$$P_r = \frac{P_t A_e^2 \sigma}{(4\pi)^2 \lambda^2 R^4} \tag{2.2}$$

P_r	:	Received power	W
P_t	:	Transmitted power	W
A_e	:	Effective aperture (area) of receiving antenna	m^2
σ	:	Radar cross section	m^2
λ	:	Wavelength of the signal	m
R	:	Range between target and receiver	m

The radar technology is applied in various industries to detect and track vehicles, weather formations, and terrain, among other things [1].

2.1.1 AESA Radar

An AESA radar is an electronically steered radar with several antenna elements. Compared to mechanically steered antenna systems that require physical rotation of the radar, the AESA radar can, through constructive interference, listen at different angles. This is done through a technique called *beam forming* [2]. Another characteristic of AESA is the possibility to group antenna elements, effectively resulting in separate radar systems that can gather information for different purposes. The newer generations of AESA-based systems typically consist of thousands of individual antenna elements [2].

Figure 2.1: An AESA antenna

2.2 Data Acquisition

The newer generations of AESA radars gather data faster than the previous generations [2]. Each antenna element of the AESA gathers data in a matrix with the dimensions [#samples per pulse, #pulses], based on a *sampling frequency* and a *Pulse Repetition Frequency* (**PRF**). The sampling frequency is the frequency at which a sample from a pulse is collected. The PRF is the frequency at which a pulse is sent from the antenna elements. There is also the *Pulse Repetition Interval* (**PRI**)

which is calculated as one over the PRF. The dimensions of the radar data matrix are often referred to *Fast Time* (**FT**) and *Slow Time* (**ST**). FT is the dimension in which samples are collected per each PRI. ST refers to the dimension built up by the number of PRIs collected for a specific data matrix. An *Integration Interval* (**INTI**) refers to the duration at which a single data matrix is collected [3].

The newer generations of AESA radar typically consist of hundreds to thousands of antenna elements, resulting in a data block with dimensions [#ST, #FT, #antenna channels] [2].

The data blocks are gathered quickly (typically in the order of tens of milliseconds), which results in a large amount of data per time unit [2]. These data blocks are then fed through a signal processing chain, dependent on the characteristics of the data. This combined puts new demands on the system such that it can process the data blocks efficiently.

2.3 Signal Processing

Signal processing chains (**SP-chain**) can be structured in many different ways and with various filters. An example of an SP-chain is shown in Fig. 2.2.

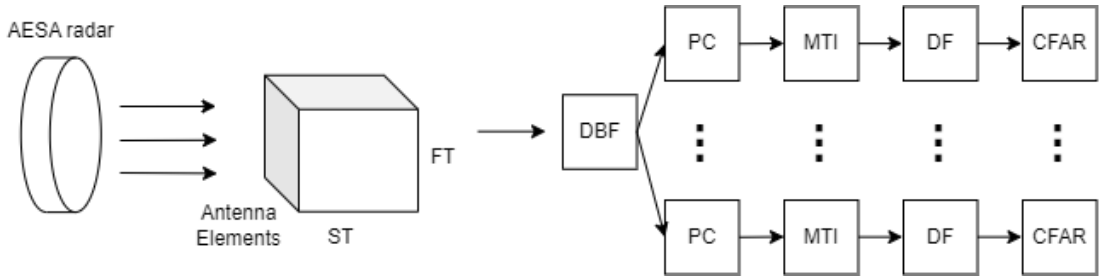


Figure 2.2: An example of a signal processing chain.

In Figure 2.2 above, the data is fed through a signal processing chain consisting of the filters:

- **Digital Beam Forming (DBF):** Multiple beams are formed using constructive interference, which makes it possible to steer toward different directions.
- **Pulse Compression (PC):** Amplifies the returning energy that matches the transmitted waveform.
- **Moving Target Indicator (MTI):** A high pass filter that effectively removes the signals returning from static targets
- **Doppler Filtering (DF):** Move from time to frequency domain that can be translated into target velocity
- **Constant False Alarm Ratio (CFAR):** A detection scheme that is used to determine if and where there are targets without too many false alarms

The data block is processed through the chain, and the output depends mainly on what signal processing is done. After the signal processing step, information about, e.g., detections can be extracted [3].

The different filters of a signal processing chain mainly consist of vastly varying matrix operations. Furthermore, some filters operate on the data blocks row-wise while others operate on the data block column-wise. This creates *data dependencies*, e.g., if the data needs to be processed row-wise, there is a horizontal data dependency [3].

2.4 Parallelism

Parallelism is the concept of performing computations on multiple tasks or data at precisely the same time. This stands in contrast to concurrency, where the objective is that multiple working units should progress simultaneously and not necessarily execute simultaneously. In order to effectively utilize different types of parallelism in a complete system, there is a need for good interplay between hardware architecture and software.

2.4.1 Parallel Architecture

Parallel computer architectures are designed for different types of tasks and are often categorized according to *Flynn's Taxonomy* [4]. These categorizations are based on the type of instructions that should be run and the incoming data streams. The taxonomy categorizes parallel computer architectures into four specific categories:

- **SISD** - *Single Instruction, Single Data*; Uniprocessor systems
- **SIMD** - *Single Instruction, Multiple Data*; Vector processor and parallel processor system
- **MISD** - *Multiple Instruction, Single Data*; Pipelined computers
- **MIMD** - *Multiple Instruction, Multiple Data*; Clusters and multiprocessor system

SISD systems include uniprocessor systems that perform single instructions on a single data stream. The first computer systems following the von Neuman architecture were often designed this way. The SISD architecture is often cheap and has low power consumption, but they have limited speed due to being uniprocessors. This architecture can be found in, e.g., microcontrollers and older mainframes.

The SIMD architecture is mainly prominent in modern GPUs, where a single instruction is executed on multiple and different data. They are efficient when performing the same instruction on large amounts of data but are limited to specific applications as they are not general processing units.

Systems with Multiple Instruction Streams, Single Data Stream (**MISD**) architecture are designed to enable parallel execution of multiple instructions simultaneously on the same data stream. Despite its limited application scope, this architecture can

be a suitable choice for pipelining systems. One use case for this architecture is the design of space shuttle flight control systems, where high reliability and fault tolerance are crucial.

Finally, the MIMD architecture can perform different instructions on multiple and different data streams. MIMD systems are often further categorized into *shared-memory MIMD* and *distributed-memory MIMD*. In a shared-memory MIMD system, the processing nodes are connected to a globally shared memory, and communication is done via modifications of said memory. In contrast, the processing nodes in a distributed-memory MIMD have access to local memory. The inter-node communication is instead established via an interconnection network. A distributed-memory MIMD is more prone to scale well than a shared-memory MIMD due to the memory conflicts that inevitably appear when additional processing units are added to a shared-memory MIMD system. These systems are great where multitasking is required, but the downside is that the architecture is more complex and expensive, which in turn causes development to be complicated and expensive. MIMD architecture is found in most parallel computers, which includes modern PCs and smartphones.

The shared-memory MIMD architecture is often referred to as *multiprocessor*, while the distributed-memory MIMD architecture is commonly referred to as a *computer cluster*. These architectures aim to utilize multiple processing resources, but they do so in slightly different ways since they are designed for different purposes. Table 2.1 compares the different architectures [5].

Item	**Multiprocessor**	**Computer Cluster**
Node configuration	CPU	CPU, RAM, net interface
Node peripherals	All shared	Shared exc. maybe disk
Location	Same rack	Same room
Internode communication	Shared RAM	Dedicated interconnect
Operating systems	One, shared	Multiple, same
File systems	One, shared	One, shared

Table 2.1: Some characteristics of the different MIMD architectures

2.4.2 Interconnection

In any parallel architecture, there needs to be some way to share information across nodes. In computer clusters, this is accomplished with an interconnection network, consisting mainly of links and switches arranged in a specific manner [6]. This network then allows multiple processors or nodes to communicate with each other through, e.g., message passing. How the network is designed will significantly impact the performance of a computer cluster since there is an unavoidable communication overhead that comes with the interconnection network. Therefore, there is a significant consideration towards the network's topology, which describes the geometric structure used to connect nodes and memory modules [6]. An interconnection network can be designed with a star topology, where a central node (or switch) connects to every

other node in the cluster. The communication in this topology is guaranteed to be efficient, with a constant diameter of 2. However, the major drawback of the star topology is the single point of failure for the central node [7]. This serves to show that the topology design of an interconnection network will vary depending on the intended use cases and requirements of the cluster and the application.

2.4.3 Processes and Threads

The concepts of *processes* and *threads* are foundational in parallel programming models, as they can describe different control flows executing on multiple cores. A major goal for using these abstractions from the perspective of parallel execution is to obtain an overall shorter execution time.

A process is defined as an abstraction of a program that is executing. A process contains all the information necessary for executing that program, such as data and program counters. Each process also has its own private address space. If two or more processes want to exchange data, this must be done through explicit communication channels [7]. For instance, communication could be done via shared memory or message passing.
Processes are executed on resources such as processors or cores, and if multiple resources are available, the processes can execute truly in parallel. Processes are often given a time slice in which they can be executed in the system's resources. However, there needs to be a way for a process to store its state since when context-switching occurs; the process needs to be able to continue its work later on.

The concept of threads is an extension of the concept of a process, where a process may have multiple threads that have their independent control flows. A main advantage of spawning multiple threads rather than multiple processes is that threads are lightweight in comparison. The threads of a process do not need to copy any address space since they all share the same address space within a process. This in turn requires threads to have access to a physically shared memory, which is the case for multiprocessors [7]. Hence, threads can offer the same type of advantages as processes concerning parallel execution while simultaneously being more flexible. More specifically, in a multicore processor, the threads of a particular process can be assigned to different cores, allowing for parallelism within the same process.

When designing and implementing a parallel program with multiple processes with threads, several essential things must be considered to gain any performance increase. Firstly, a suitable number of processes and their threads needs to be determined, which are application-dependent parameters. The degree of parallelism and how the execution time scales with the number of processes and threads are important parameters to consider, as well as the execution resources available. An ideal parallel application should spawn processes and threads such that all cores in an execution environment are performing work to use the resources available efficiently.
However, there is an overhead included with process and thread creation, management, and termination, which one wants to be kept low. If the task's granularity is too fine, the overhead will outweigh the benefits of parallelism. Moreover, an oversubscribed system will suffer performance degradation due to time-sharing of the same processing

units, as well as from memory collisions. Yet another vital thing to consider when implementing a parallel program is to, as much as possible, avoid sequentialization by the use of synchronization operations such as locks. If one is not careful with how the synchronization operations are used in the application, one may not see performance increases [7].

2.4.4 Data Parallelism

To achieve good performance on any parallel architecture system, *data parallelism* is of interest. Data parallelism is mainly revolving around the concept of a single operation that can be carried out in parallel on multiple subsets of the data [8]. The data is thus distributed over some nodes in the parallel architecture. A simple example of a data parallel task could be summing an array of integers, where slices of the array are distributed across multiple nodes, which in turn computes their local sum. Once all nodes are done, the results can be aggregated to achieve a final result. Data parallelism is especially useful when the size of the data sets can be sliced to fit the size of a node's local memory locations, e.g., its private memory or caches. However, there is a need to consider the overhead that comes with inter-node communication and data copying. If the overhead grows too large, then the excess step of trying to exploit data parallelism might not be worth it in the end.

2.4.5 Amdahl's law

Amdahl's law is a formula for measuring the maximum theoretical speed-up a program can achieve. It is defined as shown in equation 2.3.

$$S(f, n) = \frac{1}{(1 - f) + \frac{f}{n}} \tag{2.3}$$

S : The theoretical speed-up of the program
f : The proportion of the program that can be made parallel
n : The number of processors

Comparing the actual speed-up with the maximum theoretical speed-up can be an effective way of evaluating the potential success of a parallelized program. However, it requires one to accurately determine f, which often is non-trivial.

2.4.6 Simultaneous Multithreading

Tasks in a computer system generally present a mixture of being both compute- and I/O-bound. Consequently, there will be numerous times where a task occupies a processor core while, e.g., performing I/O operations. The objective of *simultaneous multithreading* (**SMT**) is to increase the utilization of the processor cores when tasks cause, for instance, lengthy memory latencies [9]. This is done by presenting a physical processor core as multiple logical processor cores, which allows a single physical core to share the physical execution resources for a number of threads. This can cause a speedup in a program, as a thread that is, e.g., waiting for an I/O

operation can lend its resources to another thread. This will effectively result in a latency hiding ability. This, however, does not imply that threads run in parallel for a single CPU, which could be obtainable with additional physical CPUs.

2.5 Memory

Processors are gaining increasingly greater performance and speed simultaneously as applications become more demanding for utilizing the available resources. However, the rate at which memory speed and bandwidth grow is not quite able to keep up with the development of processing resources. Therefore many systems saw the advent of memory hierarchies, for instance, caches. Hence, different architectures utilize these hierarchies in different ways.

2.5.1 Symmetric Multiprocessing

Symmetric Multiprocessing or *Shared-Memory Multiprocessing* (**SMP**) is the definition of two or more CPUs that together share a memory, a system bus, I/O devices, and an operating system that treats all CPUs equally. Traditionally the CPUs of an SMP architecture were single-core processors. As multi-core processors are becoming the de facto standard in modern computers, from the SMP's point of view, it treats the cores as separate and equal CPUs.
More correctly, an additional definition to SMP systems is that "the more precise description of what is intended by SMP is a shared memory multiprocessor where the cost of accessing a memory location is the same for all processors; that is, it has uniform access costs when the access actually is to memory. If the location is cached, the access will be faster, but cache access times and memory access times are the same on all processors" [10].
A drawback of this approach is that all CPUs compete for the same system bus to access the memory. Although caches for each CPU may be in place, the single bus impacts the scalability of SMPs. After some threshold, the more CPUs added, the more trouble each CPU will have to reach the main memory. In practice, an SMP design is limited to a maximum of a few dozen CPUs [5].

2.5.2 Non-Uniform Memory Access

Like in the case with SMP, all processors in *Non-Uniform Memory Access* (**NUMA**) systems have access to shared memory. Unlike SMPs, there is not necessarily the same contention for a single system bus in NUMA systems. NUMA systems split the memory into multiple *NUMA nodes*, each with separate memory busses for subsets of processors. Therefore, the memory access speeds are non-uniform, and the memory access to local memory modules is considerably faster than access to remote memory modules [5]. Since the requirement of the same memory access speed for all processors to all memory is dropped in NUMA systems, the scalability of this design to more processors is improved.
NUMA systems largely benefit from close-by memory, and to further promote this, the concept of *processor affinity* or *CPU-pinning* is relevant. Moreover, multiple

tasks that communicate greatly benefit from being allocated to the same NUMA node. Then, no inter-node communication is necessary, which could slow down the overall run time.
Ensuring a task executes on the same CPU during its lifetime can drastically change the run time using local memory locations. There is a high probability that the data the task uses is already present in local memory locations, and then the task does not have to go to the main memory to fetch the data again.

2.6 Load Balancing

The concept of *load balancing* is broad and applicable on various levels within computer science. One example is that load balancing can handle a large dynamic load of internet traffic for a web server. Another example is to utilize load balancing to distribute workloads to multiple processing nodes in an HPC cluster. In general, load balancing aims to distribute tasks fairly between processing units, so optimal resource utilization, response time, and throughput can be approached [11]. The decision of when to allocate tasks to processing units can be made before program execution or while a system is running. Therefore, the two categories *static load balancing* and *dynamic load balancing* are often considered when designing a system that wants to employ load balancing.

Static load balancing occurs before program execution. The effectiveness of such a scheme relies heavily on the accuracy of the assignment function, which assigns tasks to processor nodes [11]. A relevant requirement for a system with static load balancing is that there should be a rather computationally uniform incoming load. Since the load-balancing decisions are already made before program execution, there is no need for real-time communication between the processing units. *Round Robin* could be suitable as a simple static load balancing assignment function.

In contrast, dynamic load balancing occurs while the program is executing. Therefore, dynamic load balancing may be more suitable for a system with a higher fluctuation in the incoming load. Thus, it requires real-time communication to distribute and balance the work.

There are two common approaches to distributed and dynamic load balancing, both of which are often threshold based. That is, given a processor and its private queue containing tasks, dynamic load balancing decisions are made based on the queue status [12]. One of the approaches is the *sender-initiated* strategy, where processors can give away work if they are overwhelmed. This puts the burden of moving tasks on the already overloaded processor. Another common approach within dynamic load balancing is the *receiver-initiated* strategy, often referred to as *work-stealing* [12].

Work-stealing, in general, is a distributed dynamic load-balancing technique used in multiprocessor and cluster environments. In work-stealing, idle processors search for work from other processors who may be heavily loaded. This is beneficial for the system as a whole, as the work of finding and moving tasks is put onto the processors

that do not progress. This consequently minimizes the overhead for a processor already making progress on tasks [13].

Regardless of which dynamic load balancing technique is used, it is beneficial to consider to what degree dynamic load balancing could be done within a shared-memory node in a computer cluster. More significant communication overheads and higher overall latency may occur if global operations are used in order to load balance [13].

2.7 Software

This section provides relevant information about the software, libraries, and programming languages used in the target system and in the different scheduling implementations.

2.7.1 Julia

Julia is a programming language developed at the Massachusetts Institute of Technology (MIT). The language provides high-level expressive syntax, which makes it relatively easy to learn and use. Julia supports concurrent parallel and distributed computing, and the language is also known to be efficient when performing operations on a large amount of data.
In 2020, Karlsson conducted a master's thesis examining the efficiency of using Julia as a programming language for high-speed AESA signal processing chains [14]. Karlsson mainly compared Julia with C++. His findings were that while C++ most often outperforms Julia in execution time, Julia offers more intuitive syntax while still being competitive in speed.
The thesis concluded that Julia is perfectly suitable for prototyping parallel programs.

2.7.2 Message Passing Interface

Message Passing Interface (**MPI**) is a standardized and portable message-passing specification used for parallel computing. MPI declares a set of routines and functions for two-sided communication between multiple processes, possibly running on separate interconnected computers or within a single machine. MPI allows these processes to work together to solve a common problem by exchanging data and coordinating their activities. The processes of an MPI program have access to private address spaces, and communication between processes needs to be designed explicitly. MPI is designed to be flexible, efficient, and scalable, making it suitable for a wide range of parallel computing applications, from small clusters to large-scale supercomputers. MPI provides both *point-to-point* and *collective* communication operations, allowing for a wide range of communication patterns between processes. Overall, MPI is a widely-used and well-established parallel computing interface that provides a high-level interface for distributed computing.
The scalability, robustness, and ease of use make MPI an appropriate interface for parallelism and communication between processors in a system. Moreover, MPI

supports different types of parallel programming paradigms, namely *Single Program Multiple Data* (**SPMD**) and *Multiple Program Multiple Data* (**MPMD**). The latter allows for executing different programs, such that one can run processes with different responsibilities, all in the same communication world.

As MPI is just a specification, different implementations and wrappers are built to realize MPI in multiple languages. Two major implementations are *MPICH* and *OpenMPI*, targeting languages such as C, C++, and Fortran [15], [16]. Moreover, there exists a wrapper for MPI written in Julia [17]. When programming an MPI application as MPMD, these implementations and wrappers allow the application to utilize different programming languages for the different MPI processes.

Allocation of MPI processes to nodes in a cluster environment can be done statically via, e.g., a list of hosts' IP addresses, where each process can be mapped to a specific node if needed. The number of MPI processes per node varies greatly depending on the cluster architecture and the program to be executed. As a concrete example, consider a multi-threaded program that can make effective use of eight threads per process. Then the number of processes per node can be calculated as: $N\_processes\_per\_node = N\_CPUs/N\_threads\_per\_process$ [18].
Utilizing simultaneous multithreading in MPI programs may yield performance gains, as the number of processes could be effectively doubled. However, the possible benefits are greatly application- and architecture-dependent, and there are some caveats to be aware of before adopting logical cores. The contention for memory and communication resources will increase with more processes per node, which will affect performance. Furthermore, logical cores will compete with cache resources, thus resulting in a higher cache-miss ratio. Therefore, extensive testing is required in order to determine if the use of simultaneous multithreading is beneficial for the application at hand [19].

2.7.3 OpenMP

Open Multi-Processing (**OpenMP**) is an API for parallelizing computations primarily designed for shared-memory parallel systems. OpenMP is not a programming language. Instead, it heavily uses compiler directives to structure parallel regions, which multiple threads can execute in parallel. OpenMP focuses on portability, user-friendliness, and efficiency to parallelize applications and programming with OpenMP is done in C, C++, or Fortran [20].

OpenMP is based on shared address spaces, and the parallel execution comes from executing multiple threads in a multiprocessor system. One of the major focuses of OpenMP constructs is the concept of *work sharing*, where each thread can execute its task on different shares of the input data. To spawn these threads and create parallel regions of an application, OpenMP utilizes a fork-join programming model. The master thread forks multiple threads for execution and then synchronizes with an implicit barrier at the end of a parallel region. This may yield performance gains; however, it might be the case that one thread's execution time is consistently the bottleneck of the total execution time [20].

2.8 Hybrid Programming Model

Many high-performance computing environments today are designed hierarchically. A cluster often consists of multiple shared-memory nodes, each with multiple cores, connected via a high-speed interconnection. This design makes the cluster suitable for multiple levels of parallelism for different levels of the environment. Inside each node, shared memory parallelization can be utilized to effectively communicate and compute using the resources of a single node. There is also the possibility for parallelization between each cluster node, that is, on a distributed memory level [21].

The trend of this hierarchical design for computing clusters seems likely to continue for some time. Due to this, a natural approach is to utilize different levels of parallelism, where suitable parallelization APIs are used at different levels of a cluster. For example, inside each node, one could use OpenMP for shared memory parallelization and **MPI** for communication and parallelization at the distributed memory level, leading to a clear separation of concerns. This parallel programming model is often referred to as a *hybrid programming model*, and utilizing these APIs is often referred to as *hybrid MPI+OpenMP* [21].

A system with a hybrid model will encounter some challenges that need to be addressed, and there is no best solution for all types of applications. One of these challenges is mapping MPI processes and OpenMP threads to the nodes of a cluster. One approach is the *fully hybrid* model, where one MPI process is mapped to each node, and OpenMP is used internally in a multi-core node. This approach only uses message passing as inter-node communication and intra-node communication is done via shared memory. This may be beneficial as the threads of a process can share the data without distributing it across processes. However, this approach has some drawbacks. One issue is in regard to applications employing a master-only style of MPI communication, where only the master thread of a process communicates via MPI, and all MPI communication occurs outside of a parallel region [21]. A single communicating thread might not very well utilize the bandwidth of the interconnection network. Another thing that impacts performance is that whenever the master thread communicates with MPI, all other threads need to be idle, which means that the scalability within each node will be limited [21]. An argument against this design may also be that it goes against the original programming model of MPI. MPI was designed when having single-core and single-processor nodes was more common. Therefore, MPI is more tailored towards having one process per each processing unit.

Additional MPI processes per node can be allocated if, e.g., the utilization of resources could be more satisfied with more than one MPI process per node. This is referred to as a hybrid *mixed model* [21]. This model improves on the fully hybrid model partly due to the possibility of better utilizing the network bandwidth because of a larger number of master-only processes.

Regardless of which approach for a hybrid model is used, there is the added complexity of managing two levels of parallelism. Code has to be correct regarding MPI and threading, which will lead to more complex development [21]. Moreover, a hybrid

programming model is generally as performance-portable as possible, but it will require a system-tailored setup for it to be efficient in an applicable environment.

When opting for a hybrid model, there is a need to tell the MPI implementation the thread level requested in the application to ensure that a multi-threaded application does not interfere with the MPI calls. The MPI specification lists four classes of thread safety for an application:

- **MPI_THREAD_SINGLE**: Only a single thread will execute within the MPI application.
- **MPI_THREAD_FUNNELED**: The MPI processes can be multi-threaded, but all MPI calls are done via the master thread.
- **MPI_THREAD_SERIALIZED**: The MPI processes can be multi-threaded, and all threads can communicate with MPI. However, they can not communicate concurrently.
- **MPI_THREAD_MULTIPLE**: The MPI processes can be multi-threaded, with no limitations on how to structure MPI calls.

A threaded MPI program can request a suitable thread level and will be provided the same level if there is support for it in the current environment [22]. To clarify further, these thread levels allow a blocking MPI call to only block the calling thread while other threads can still progress their tasks.

2.9 Scheduling

Scheduling, in general, revolves around the mechanism of mapping tasks to processing units following a certain scheduling scheme. Scheduling is a complex problem to solve optimally, and as such, multiple scheduling algorithms exist for different environments. Furthermore, these algorithms are often tailored toward specific systems (such as real-time systems), towards uniprocessor or multiprocessor systems, or both simultaneously. As such, the choice of scheduling algorithm depends on the requirements of the system environment at hand.

Scheduling algorithms may rely on additional data about a task to determine how to schedule tasks onto processors. An example would be that tasks can be assigned different *priorities*. For instance, if a system is uniprocessor, and a task set is given, a scheduler could schedule the tasks in order of priority onto the processor until all tasks are completed.

2.9.1 Scheduling in Real-time systems

In *real-time systems* with vast amounts of incoming data, it is crucial to form tasks from the data and process them as quickly as possible. These tasks also have a set deadline. Generally, there are two classes of real-time systems - *soft* real-time systems and *hard* real-time systems. Missing a deadline in a hard real-time system is unacceptable, contrary to a soft real-time system where it might be inconvenient

but still acceptable. An algorithm that schedules real-time tasks onto computational units is called a *Real-time Scheduling* algorithm.

A foundational paper in the field of scheduling in real-time systems was published by Liu and Layland [23]. This paper presents a task model based on five distinct assumptions that characterize a task in a hard real-time system. With this model defined, they continue to prove a series of theorems leading to three different scheduling algorithms. The algorithms presented are based on task priority and preemption and mainly differ in how task priorities are assigned. First, the *rate-monotonic scheduling algorithm* is presented for fixed priorities assigned beforehand. Then, if the priorities can be dynamically assigned, Liu and Layland present the *deadline driven scheduling algorithm* (also known as *earliest-deadline-first* (**EDF**) algorithm). Finally, the *mixed scheduling algorithm* is presented, which combines the two types of priority assignments to get their best properties. Despite the age of this work, these algorithms are still widely used today in real-time systems, and the paper laid the foundation for much future research in the field.

Liu and Layland considered a uniprocessor environment in this paper, but modern systems usually utilize multiple processors for the parallel execution of tasks. It turns out to be a much more complicated problem in a multiprocessor system since few of the results collected for a uniprocessor system generalize directly to a multiprocessor system [24]. Although the fixed scheduling algorithm proved to be optimum, it is still a complex problem to determine beforehand if a set of tasks can be scheduled (all tasks meet their deadlines) or not in a real-time multiprocessor setting. As such, Leung and Whitehead [25] researched the complexity of determining this concerning fixed-priority assignment. They concluded that scheduling a set of periodic, real-time tasks onto $m \geq 1$ equivalent processors is NP-hard to decide if the set is schedulable.

2.9.2 Multiprocessor Scheduling

The complexity of scheduling real-time tasks in a multiprocessor environment grows surprisingly compared to scheduling in a uniprocessor system. This is the case even if a multiprocessor system is homogeneous. In addition to answering the question of priority; “which task should run next?”, scheduling in a multiprocessor system has to answer the question of allocation; “where should this task run next?”. However, two major approaches to multiprocessor scheduling in real-time systems are often considered, namely *global scheduling* and *partitioned scheduling*.

Global scheduling (or dynamic task assignment) encompasses the concept that an instance of any task can execute on any processor in the system. Global scheduling is employing a single global ready queue of tasks. These tasks can then be dispatched to any processor in the system [26]. Using global scheduling in a multiprocessor system effectively load balances the system automatically since there is no need to balance any task queues between the processors. Thus, global scheduling also may be a simpler implementation than other multiprocessor scheduling approaches.
However, some drawbacks will most certainly affect the system’s performance. For example, synchronization overhead due to a shared data structure is one of the things that will scale poorly. Moreover, global scheduling does not consider cache affinity

since any processor may execute any task from the ready queue. Hence, system performance will be degraded due to a high rate of cache misses.

Partitioned scheduling is another approach to multiprocessor scheduling. Instead of a global shared ready queue for all tasks, each task is assigned to a processor's private ready queue. In this scenario, tasks never migrate between processors, and as such, cache affinity is taken into account. Moreover, once tasks are assigned to a processor, uniprocessor run-time scheduling is possible, which opens the possibility to use established uniprocessor analysis [26]. However, in contrast to global scheduling, a prominent issue with partitioned scheduling is the load imbalance between the processors.

3

Methodology

The methodology used in the thesis is described below and should give a reliable foundation such that the reader could recreate or re-implement parts of the project. The following sections are ordered chronologically concerning the project's timeline.

3.1 Literature Studies

Firstly, literature studies were conducted to establish a robust and reliable foundation for the research and the implementations. The literature study involved a review of relevant academic works, both theoretical and empirical, that are related to this research topic. The purpose of this study was to gain a thorough understanding of the current research in the field of cluster real-time scheduling. Another objective of the literature study was to determine the capabilities and responsibilities of a scheduler in representative scenarios and the computing environment in which it should operate.
While researching scheduling in cluster environments, there was much consideration toward throughput, latency, meeting task deadlines, and utilization of available hardware resources. Since a large amount of data needs to be processed quickly, the overhead of a scheduler should not outweigh the benefits from when, e.g., inter-node communication is used.

The studies provided valuable knowledge in the field of real-time scheduling as well as an intuition about the current research in the field. During the literature studies, it became evident that there are immoderately many possible target systems, each with its unique characteristics. Hence, many variations of real-time scheduling algorithms exist, and few of them were directly applicable to the target system in this project. The literature with the most relevance and influence on the project was gathered and is presented in chapter 2.
The literature studies were not only done during the initial part of the project but rather as a continuous process during almost the entire time span.

3.2 Target Environment

To efficiently implement and subsequently evaluate different scheduling systems, there was a need for careful consideration of the target environment. Consequently, a thorough analysis of the structure of the target environment was needed.

The target environment in this project was a computer cluster consisting of four identical nodes. Each node has an identical 28-core processor (**SMT** enabled) with a shared L3 cache and access to 196GB RAM via NVMe. Moreover, each node consists of a single **NUMA** node. The interconnection topology is designed as a star network, with a network switch as the central node. The interconnect network is established with Ethernet. Fig. 3.1 shows a simplified overview of the topology in the target system, while Appendix A describes each node's architecture in more detail.

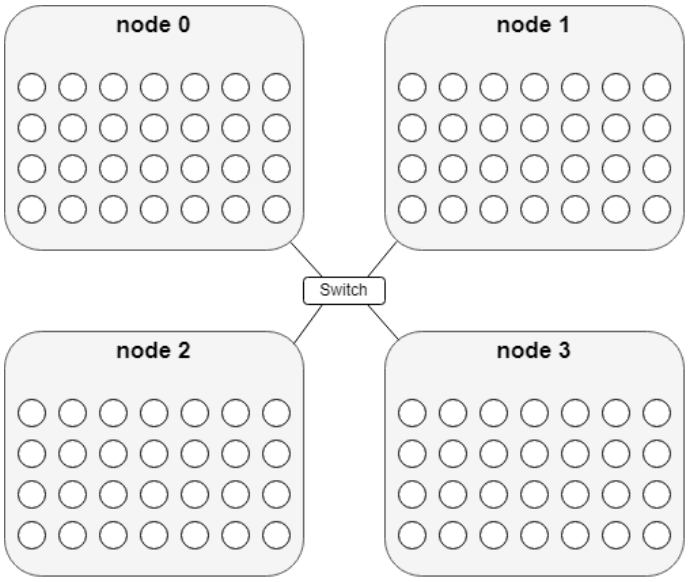


Figure 3.1: A simplified overview of the target system, each node has one socket with one 28-core processor

3.3 Representative Fictive Scenarios

The project initially began by working on fictive, yet for the project, representative scenarios. These scenarios enabled the research to be narrowed down to the main topics of this project. Moreover, considering a confined set of scenarios, comparisons between prototypes could be more accurate. Since the project revolved around the computational impacts of varying and dynamic workloads, the fictive data was randomized, and the representative signal processing chains were arbitrarily chosen. However, they were representative concerning representative dimensions and operations per data item as a real application scenario. With these scenarios, a suitable way was achieved to measure computational capacity, test implementations, evaluate and compare different implementations, and run benchmarks. A more consistent way to collect measurements and compare those to quantitative goals was also obtained with this approach, as the computational demands for a signal processing application are largely data dependent.

The main components in establishing a representative scenario were:

- Creating the data loads
- Modeling the inflow of data

- Modeling the task deadlines
- Modeling the dynamic environment
- Modeling the signal processing

These together formed a representative scenario, and they had to be implemented to resemble a real application radar system. To evaluate prototypes and how different methods could be applied in these scenarios, they were analyzed in the intended target environment.

3.3.1 Data Loads

The data blocks collected from the radar and fed through signal processing chains are of varying dimensions in a real application scenario. The dimensions of the data can vary due to a change of radar mode or a change of **PRF**. Therefore, a representative data load needs to model these changes accordingly.

Some default ranges of values for different parameters were needed to model the representative data loads. The sampling frequency was set at a constant 5 MHz to represent a reasonable scenario. The PRF is modeled to change for successive data blocks and is within the range of 10 kHz to 20 kHz. These frequencies are in the ranges of a *Medium Pulse Doppler* (**MPD**) radar, which is representative of this project's scope. Furthermore, the slow time (ST) dimension of data blocks is in the interval of 64 to 512, with each increment being a power of two relative to the preceding step.
The PRF changes for each consecutive data block while the ST dimension stays the same for a period of time before changing. This pattern repeats as long as needed for a representative scenario.

Once the above parameters are modeled, they can be used to compute the fast time (FT) dimension for each incoming data block. The fast time and slow time dimensions can then be used to calculate the integration interval (**INTI**) for each block:

$$FT = \frac{f_{sampling}}{PRF} \tag{3.1}$$

$$INTI = \frac{FT \cdot ST}{f_{sampling}} \tag{3.2}$$

FT	: The fast time dimension size of a data block
$f_{sampling}$	: The sampling frequency
PRF	: Pulse Repetition Frequency
ST	: The slow time dimension size of a data block
$INTI$	: Integration Interval

The INTIs computed are utilized to simulate the data acquisition rate obtained from a radar system. More specifically, the influx of data will vary with successive data

dimensions to model a dynamic real application scenario.

ST	MIN (ms)	MAX (ms)
64	3.2	6.4
128	6.4	12.8
256	12.8	25.6
512	25.6	51.2

Table 3.1: How the INTI (data acquisition rate) changes for different ST-dimensions

3.3.2 Influx of Data

The influx of data was modeled by introducing a provider responsible for sending the data blocks to the scheduler process. The provider is located on the same cluster node as the scheduler, which enables the communication of data blocks via shared memory. More concretely, both the provider and scheduler get access to a threadsafe FIFO queue to which the provider writes, and the scheduler reads. The provider sends the data blocks in time intervals that are calculated using Eq. 3.2.
The deadline for a data block is set right before the provider sends it to the scheduler, meaning that a certain part of the latency might be spent waiting for access to the queue. However, this is justified by acknowledging that there is some latency between an AESA radar and a scheduler process in a real application scenario as well.

3.3.3 Task Deadlines

There are different ways to compute the deadlines for the data blocks depending on the requirements for the specific context. One way to compute the deadlines for a data block is according to the formula shown in Eq. 3.3.

$$Lat_i = a \cdot \frac{FT_i \cdot ST_i}{f_{sampling}} \tag{3.3}$$

Lat_i : Deadline for data block i
FT_i : The fast time dimension size of data block i
ST_i : The slow time dimension size of data block i
$f_{sampling}$: The sampling frequency
a : Latency constant

This formula is equivalent to the INTI calculation shown in Eq. 3.2 except that the deadline formula also contains the a constant. The a constant describes how many times slower it is allowed for the block to be processed compared to how fast it was acquired.
However, this formula is tailored for an extremely demanding scenario where high-resolution scanning with minimal delays is critical. Implementing a system-level scheduler that meets such deadlines requires a significantly longer project timeframe and is not feasible in this thesis. To counteract this while maintaining a representative deadline formula, the formula shown in Eq. 3.4 was used.

$$Lat_i = a \cdot \frac{FT_{max} \cdot ST_{max}}{f_{sampling}} \tag{3.4}$$

Lat_i	:	Deadline for data block i
FT_{max}	:	Fast time dimension size of the largest data block
ST_{max}	:	Slow time dimension size of the largest data block
$f_{sampling}$	:	The sampling frequency
a	:	Latency constant

This means that every data block has an equally long period of time before its deadline, independent of its dimension sizes. This is justified by the fact that smaller data blocks are acquired faster, leading to a larger amount of data blocks in a shorter period of time. The latency constant a is often in the range of 3 to 5 and should be the same for every data block.

3.3.4 Dynamic Environment

In general, the computational demands of a signal processing chain depend on the algorithms involved in the chain, but it may also largely depend on the number of detections contained in a data block. This leads to the SP-chains having a wide range of execution times that are difficult to predict. Since the numeric data of the modeled data blocks are randomly generated, there will most likely be zero detections and consequently no dynamic behaviors.
To model this dynamic behavior, a static SP-chain with a fixed amount of filters is introduced in combination with two variables; *intensity* and *difficulty*. The intensity is an integer in the range [1,15], and the difficulty is either of `EASY` or `HARD`. These two variables together decide how to modify the computational complexity of a single task. How this works can be seen in Figure 3.2.

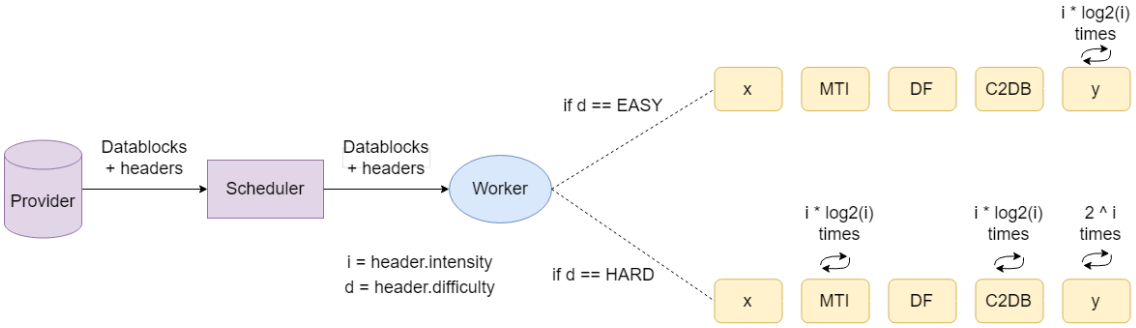


Figure 3.2: The connection between intensity and difficulty.

The intensity variable affects the computational demands of an SP-chain depending on the level of difficulty. Some filters in the parallel sequences are repeated based on the intensity variable, while a serial sequence is repeated at the end. This part is concerned with the detections found in a data block, and since there will likely be zero detections in randomly generated data, this is also affected by the intensity variable.

The intensity variable represents the computational load of the data block and is set separately for each data block by the provider. To model the unpredictability of the signal processing chain, the intensity variable is invisible to the scheduler meaning that no decisions can be made by using the value of the intensity variable. The provider can set the intensity of each block in different ways; the most representative way of picking the intensity is by sampling from a right-skewed normal distribution. This is because too low intensities may result in scenarios that are not challenging enough. The mean of this distribution was set to 7 and the standard deviation to 2.5.
The difficulty variable represents different versions of signal processing chains where a higher difficulty could lead to a higher resolution of the detections or the possibility of seeing objects farther away. The difficulty is set by the scheduler and should be ideally set in a way such that the deadlines are met for *every* data block but, at the same time, favor `HARD` difficulties. All deadlines should optimally be met since, for instance, an operator of a radar system should be able to detect objects around her at all times.

3.3.5 Signal Processing

Each data block that is produced and enters the system should be processed through a signal processing chain. To model a real-application scenario, the SP-chains are structured as multiple sequences, each consisting of multiple filters in a specified and fixed order. These sequences can be run in parallel. A data block is split, and each slice is distributed to a sequence. The data dependencies that might exist within a data block are taken care of by merging the block at specified instances of the sequences. Thus, each sequence can be executed in parallel, where an implicit barrier is placed whenever the data slices need to be merged again. A simplified example is shown in Fig. 3.3.

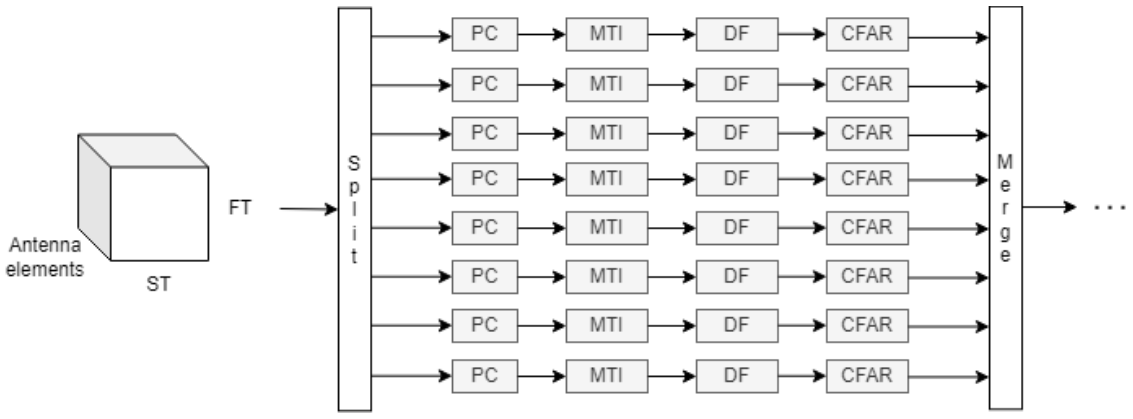
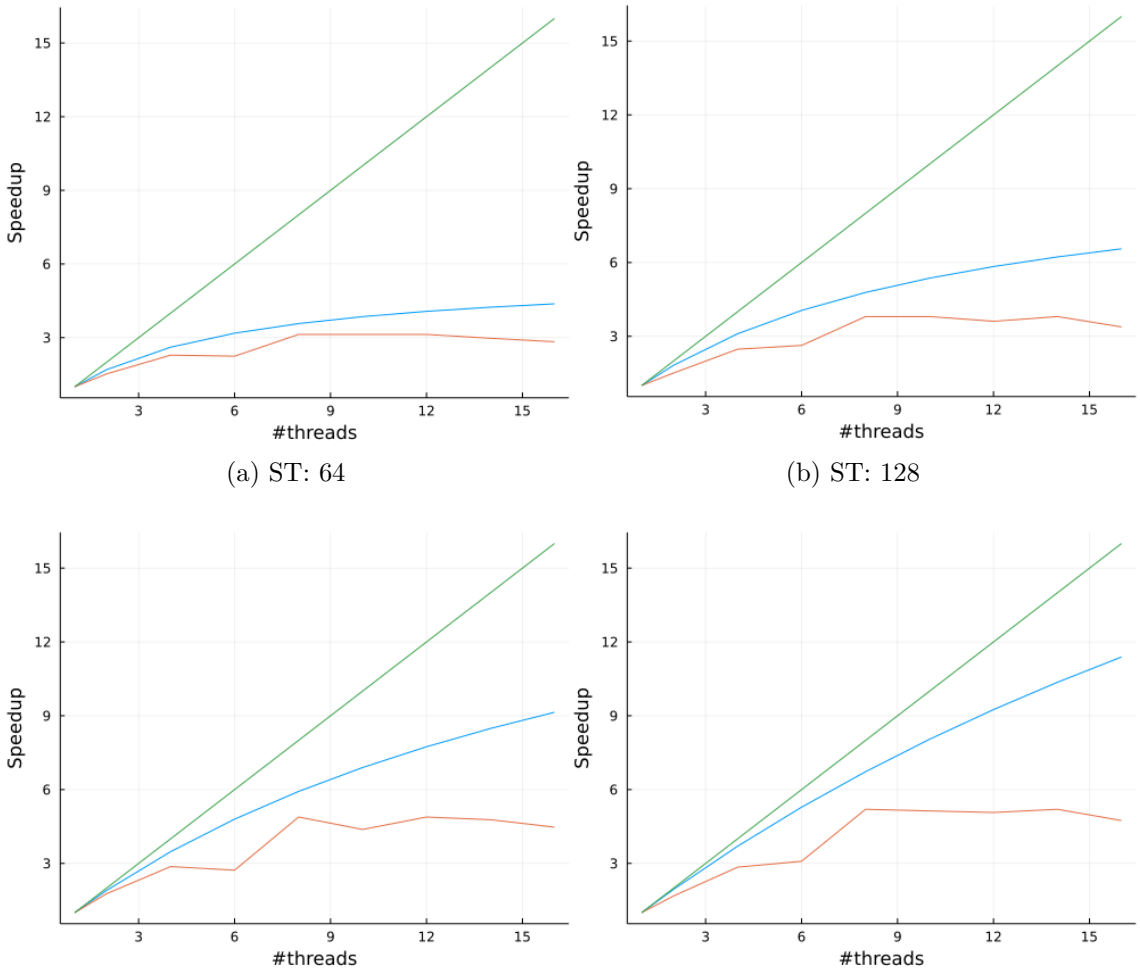


Figure 3.3: A simplified view of parallel sequences within a signal processing chain.

In these representative scenarios, an SP-chain consists of eight sequences that can be run in parallel. More concretely, a process that should compute the SP-chain can effectively use up to eight threads. Due to the varying data block dimensions, the fraction f present in Amdahl's law (Eq. 2.3) varies as well. Figure 3.4 shows the

speedup for the different ST dimensions of the signal processing tasks according to Amdahl's law. As can be seen, there is generally increased performance with respect to the execution time of up to 8 threads for a single task. It can also be observed that there is a marginally larger possible speedup for large ST dimensions compared to smaller ST dimensions. Nevertheless, due to the varying execution times of the representative scenarios, it is not always necessary to utilize eight threads to complete a task before its deadline.



(a) ST: 64

(b) ST: 128

(c) ST: 256

(d) ST: 512

Figure 3.4: Speedup graphs (Amdahl's law) for the signal processing tasks. Blue: Theoretical speedup, Orange: Actual speedup, Green: Ideal (linear) speedup

3.4 Baseline Scheduler

A baseline scheduler was deemed useful for its importance in evaluation, even though a naive implementation caused apparent bottlenecks in the system. The implemented Baseline Scheduler simply distributes tasks to workers in a round-robin fashion with blocking MPI communication.

The model components of the Baseline Scheduler consisted of the following:

- **Provider:** Generates data (see section 3.3.2). Implemented in Julia.
- **Scheduler:** Schedules tasks to workers. Implemented in Julia.
- **Workers:** Executes the assigned tasks. Implemented in C++.
- **Collector:** Collects the output from workers. Implemented in Julia.

Most of the components were written in Julia, as the language has a well-supported MPI wrapper, parallel programming support, and a high-level expressive syntax that makes it a good candidate for prototyping. All this while still being competitive in terms of speed compared to other purely compiled languages. The worker components were written in C++. The workers are more performance-critical, and thus their behavior is required to be fast, memory-efficient, precise, and repeatable. There was also reusable functionality for the workers provided by the stakeholder that was utilized in this project's representative scenarios. Furthermore, since MPI is a specification with multiple wrappers and implementations, it is possible to develop MPI processes using different programming languages.

The inter-node communication in the model consists of point-to-point communication via MPI (see section 2.7.2). The distribution of MPI processes works as follows:
One MPI process is spawned for the collector and one process for the scheduler (which also runs the provider in parallel). The number of worker MPI processes varied between evaluation runs of the Baseline Scheduler, and how worker processes are allocated was of great interest since the overall performance for the different numbers of workers were tested and evaluated.
The Baseline Scheduler distributes the tasks to the workers in a round-robin fashion. Furthermore, if the next worker in line is not done with their previous task, the scheduler waits until that worker is ready for a new task before continuing. It is also worth mentioning that this scheduling implementation leads to the workers performing equally (or as equally as possible) many tasks regardless of the individual workloads of the tasks. Fig. 3.5 shows an overview of how the Baseline Scheduler behaves.

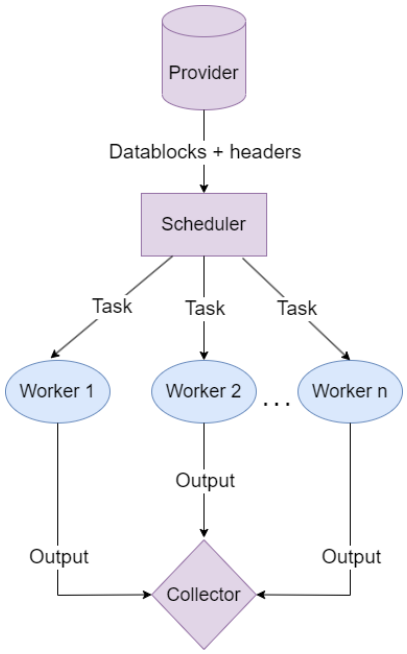


Figure 3.5: An overview of the Baseline Scheduler.

The collector's purpose is to perform timings and calculate metrics such as throughput, latency, and the rate of met deadlines. To allow such calculations, the workers store the necessary data in an output header and send it to the collector after performing their task. It is worth mentioning that some additional randomly generated detections are sent from the workers to the collector. This is to simulate a similar communication load on the system as in a real application scenario.

The most obvious flaw of the Baseline Scheduler was the static load balancing of tasks between the workers and the blocking communication from the scheduler, which resulted in large total idle times for the workers. An example scenario of the Baseline Scheduler with four workers and 12 different tasks can be seen in figure 3.6.

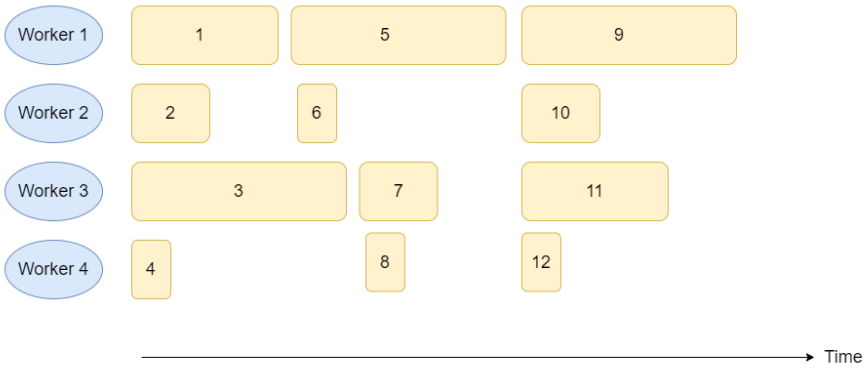


Figure 3.6: An example of the static load balancing of the Baseline Scheduler.

This static load balancing, in combination with the total idle time of the workers, lead to undesired throughput and a large ratio of missed deadlines. To avoid this, a

scheduler that achieves a more dynamic load balancing was desired.

3.5 Worker Initiated Scheduler

A new scheduling implementation was desired that should fulfill specific requirements. Firstly, the scheduler should be able to utilize the available processors fairly. The processors were aimed to be load-wise balanced in such a way that no processor was constantly doing most of the processing. Secondly, the prototype aimed to manage dynamic data loads, i.e., data loads with varying dimensions, computational demands, and inflow rates. These tasks will vary greatly in latency, and the system should not bottleneck due to certain workers being busy processing long tasks.

The implemented scheduler provided dynamic load balancing through *receiver-initiated* communication between the scheduler and workers. The idea is that the communication initiative is reversed compared to the Baseline Scheduler. That is, workers send requests to the scheduler when they are available for work, and the scheduler responds with a task to the requesting worker. If all workers are busy, the scheduler will simply block while waiting for a new request. This leads to a dynamic distribution of tasks since the task assignment depends on the tasks' latency. This implementation was named *Worker-Initiated Scheduler* (**WI Scheduler**), and an overview of this implementation can be seen in figure 3.7.

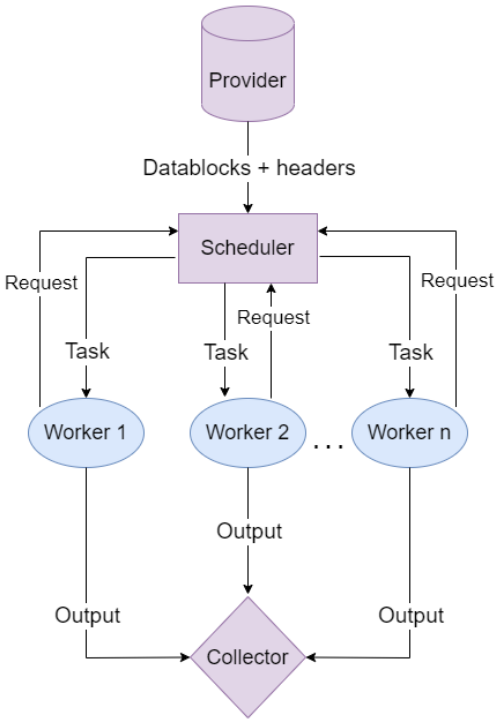


Figure 3.7: The worker-initiated scheduling implementation.

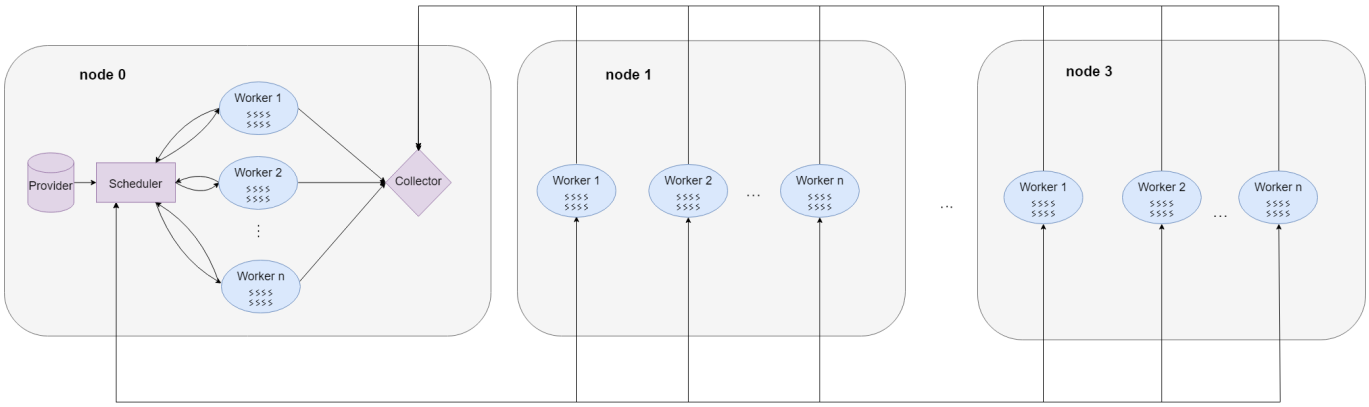


Figure 3.8: The new scheduling implementation on the system level.

Figure 3.8 shows how the WI Scheduler is realized in the computer cluster. The jagged lines on each worker process denote the threads each worker spawns, described in more detail in section 3.3.5.
The WI Scheduler achieved a more satisfactory load balancing and less idle time for the workers. An example of this load balancing can be seen in Figure 3.9, using the exact prerequisites as in Figure 3.6.

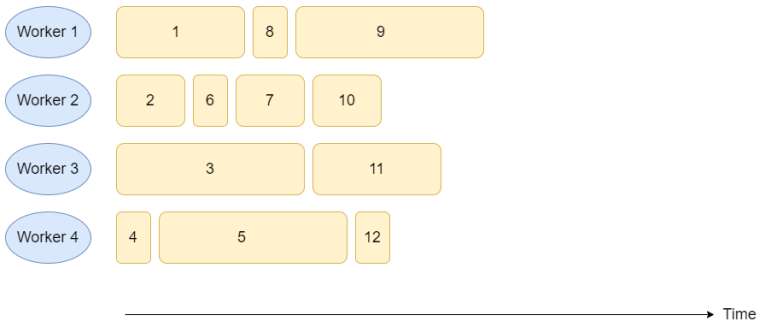


Figure 3.9: An example of the dynamic load balancing of the WI scheduling implementation

3.6 Investigative Analysis

An analysis step followed after the Baseline Scheduler and the WI Scheduler were implemented to draw conclusions from the outcomes. The analysis was the foundational factor for judging how the implemented schedulers corresponded to the quantitative goals of this thesis, mentioned in 1.4.1. Moreover, the results of this analysis steered the iterative refinement process.

An applicable approach to evaluate the goals of the thesis was benchmarking. For instance, the Baseline Scheduler provided a benchmark that could be compared with the WI Scheduler. Comparing the benchmarks of different implementations was a major method of evaluation.

The analysis was structured in such a way that the comparisons between the implementations could be justified. For instance, during benchmarking, the implementations had the same prerequisites in terms of data inflow and resources available. By performing benchmarks for both the Baseline Scheduler and the WI Scheduler, certain analyses could be concluded.

First and foremost, there is a great performance increase when establishing worker-initiated communication since the burden of seeking new tasks is distributed to the already idle workers. Another meaningful insight was how worker processes could be allocated to the nodes in the cluster environment. The worker processes do not require any inter-process communication amongst themselves, so the placement of the processes regarding their respective MPI rank does not cause any significant impact.
The analysis also revealed that there seemed to be some contention for memory and processing resources. This caused individual workers' execution time to degrade because more MPI processes (not oversubscribing) were allocated to the same node in the cluster.

The analysis also provided interesting insights regarding the use of logical cores and not just physical cores. The utilization of logical cores caused the worker's execution times to increase, as seen in Figure 3.10. The CPU-pinning strategies are described in more detail in Appendix B.1

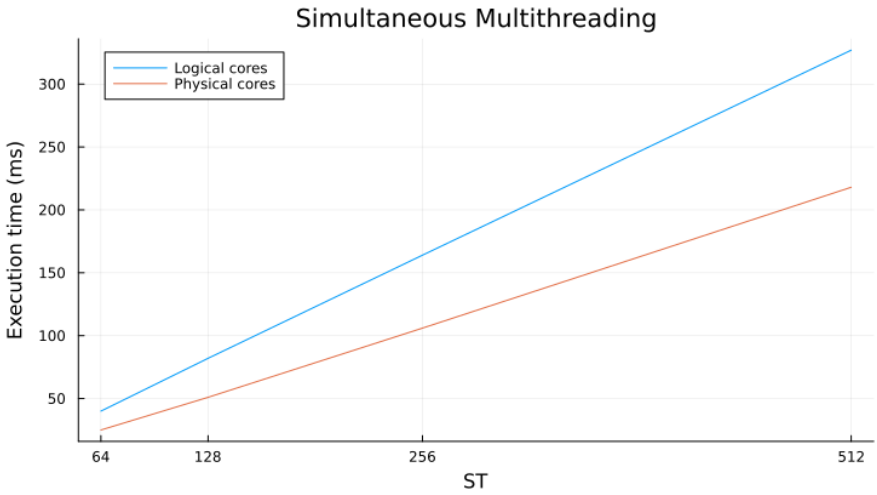


Figure 3.10: Worker execution times using different CPU-pinning strategies, one node, one worker (MPI process).

While the worker's execution times increase with the use of logical cores, the overall system latency decreases. Hence, the conclusion was that the system might benefit from additional workers, although using logical cores causes increased worker execution times. However, there is a limit for the number of workers per node so as not to introduce oversubscription, which degrades the performance significantly. For instance, a node in the target system has access to 56 logical cores, which equates to a maximum of seven workers if they each utilize eight threads simultaneously.

Based on the insights from the analysis, two interesting questions arose which would

inspire further iterative improvements. One question is related to strategies for placement and the number of MPI processes on each node and its relation to memory contention. The other question is related to the potential of dynamically allocating resources to the workers.

3.7 Iterative Improvements I

Consequently, further system improvement work was founded on the conclusions found in the first investigative analysis described in Section 3.6.

3.7.1 Subgroups Scheduler

It became apparent during the analysis that running more MPI processes on the same node leads to slower execution times for each worker, although a node was not oversubscribed. This was the case even when the processes and their threads were CPU pinned. Figure 3.11 more concretely shows this behavior.

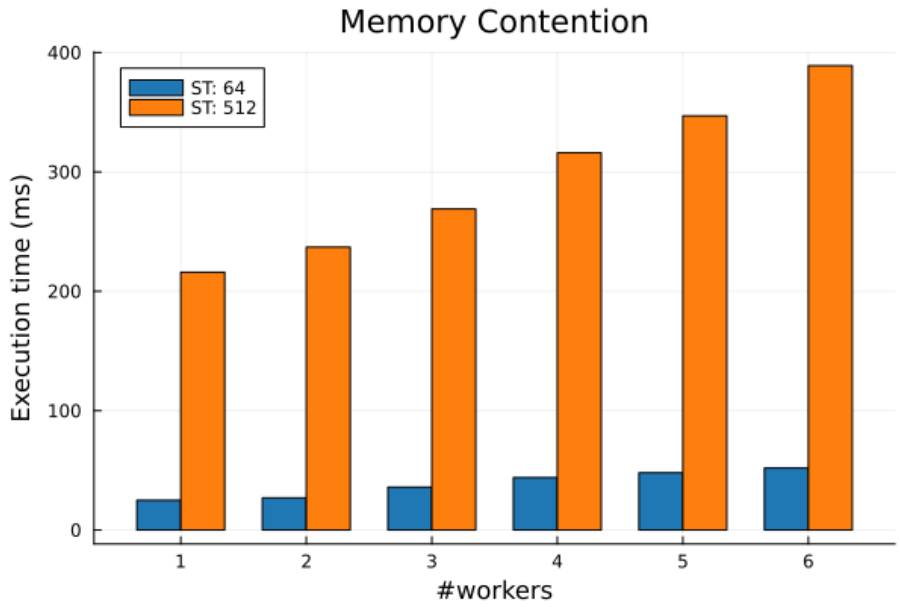


Figure 3.11: Average worker execution times of a task, for a different number of worker processes on one cluster node, for ST dimensions 64 and 512

This led to the following question: Is it advantageous to have, e.g., two groups of workers on each node where one group consists of priority workers that always receive tasks, and the other group consists of non-priority workers that only receive tasks when the system needs additional workers? This way, when the number of workers in the priority group is enough to handle the data inflow rate, the bottleneck is in the worker's execution time, which could be optimized by minimizing contention for memory resources. This implementation also relies on the observation that blocking worker processes on the same node does not cause any significant slowdown for other progressing workers' execution times.

The result of this question was the *Subgroups Scheduler*, and the implementation is described in pseudo-code in Algorithm 1.

Algorithm 1: Subgroups Scheduler

```
subgroup_workers ← A subset of all available workers
helper_workers ← The rest of the workers that are not in subgroup_workers
queue ← Task queue, which the provider sends tasks to
waiting_helpers ← Collection for temporarily storing incoming requests
run_helper_workers(header) ← Condition for if helper_workers should be
 utilized or not

while Running do
    (header, data) = take(queue)
    if (!run_helper_workers) then
        while (True) do
            worker = receive_work_request()
            if (worker ∈ helper_workers) then
                waiting_helpers.append(worker)
            end
            else
                send_task(worker, header, data)
                break
            end
        end
    end
    else
        if (length(waiting_helpers) > 0) then
            worker = pop(waiting_helpers)
            send_task(worker, header, data)
        end
        else
            worker = receive_work_request()
            send_task(worker, header, data)
        end
    end
end
```

For this scheduler, there were at least two important topics to investigate further for performance gains. Firstly, since the sizes of the different groups of workers have to be set statically, appropriate sizes for the groups were needed. A subgroup needed to be large enough to manage the inflow of data reasonably well while still being as small as possible to reduce contention for memory resources, consequently reducing worker execution times.
Secondly, the condition for deciding when to use the helper workers also played a prominent role in the overall performance. A fruitful observation from the evaluation step was that different slow time dimensions require different numbers of workers to

manage the data inflow. Smaller ST dimensions generally require more workers than larger ST dimensions to be most performant in terms of the ratio of met deadlines. This can be observed in Table 3.2.

	ST Dimension			
#workers	**64**	**128**	**256**	**512**
1	0.09%	0.3%	0%	0.2%
2	0.4%	1.1%	0.2%	0.8%
4	1.45%	2.3%	1.9%	10.9%
6	2.4%	25.3%	95.0%	81.2%
8	4.9%	97.4%	96.0%	84.1%
10	93.1%	97.7%	94.7%	79.2%
12	97.7%	97.2%	94.3%	77.8%
14	97.5%	95.8%	93.1%	75.8%

Table 3.2: Ratio of met deadlines per ST dimensions, for varying numbers of workers

These results imply potential performance gains in varying the number of workers depending on the current ST dimension. For example, it can be observed that the best ratio of met deadlines is achieved with 10-12 workers for the smallest ST dimensions. For the larger ST dimensions, running 12 workers leads to a significant performance loss compared to the optimal setting of 8 workers. This finding serves as additional motivation for developing a dynamic scheduler capable of adjusting the number of workers.

Moreover, the inflow of data is periodic, where data with the same slow time dimension arrives for a period before switching to another ST dimension. In a real application scenario, this periodicity could be caused due to a radar mode being used for a period, before switching to a different mode. Some modes produce smaller dimensions with a higher frequency, while others produce larger ones at a slower frequency. Therefore, if the scheduler detects a change in the ST dimension, a decision could be made on whether helper workers are needed. A condition then could base its decision on the current ST dimension of the incoming blocks. The behavior of incoming ST dimensions can be seen in Figure 3.12.

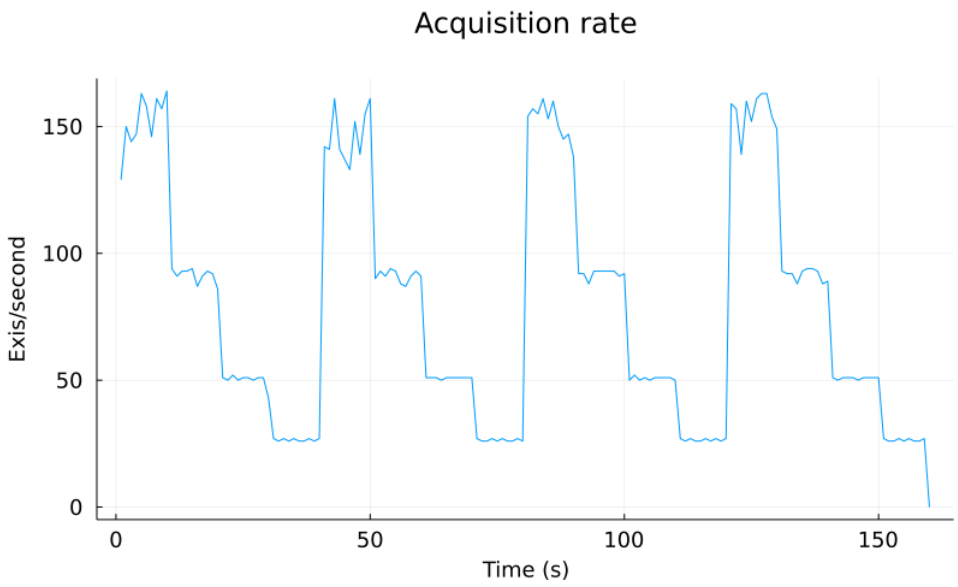


Figure 3.12: Behavior of incoming ST dimensions

3.7.2 Dynamic Resource Allocating Scheduler

Data blocks with smaller dimension sizes are acquired faster and thus have a higher inflow rate. In addition to this, data blocks of different ST dimensions benefit differently from different numbers of threads (as can be seen in Fig. 3.4). Recalling that the inflow of distinct ST dimensions follows a periodic pattern, could it be beneficial to have more workers with fewer threads per worker during periods of small data blocks and fewer workers with more threads per worker during periods of large data blocks?
Consequently, the idea was to investigate the potential of a scheduler that balances the number of workers and their allocated resources to better adapt to the characteristics of the data.

The *Dynamic Resource Allocating Scheduler* (**DRA Scheduler**) was implemented, which has the same worker-initiated communication as the **WI Scheduler** but with some additional functionality. This functionality can be seen in Algorithm 2.

Algorithm 2: Dynamic Resource Allocating Scheduler

```
worker_nodes ← Dictionary mapping workers to their nodes
worker_avail ← Dictionary mapping workers to their availability
worker_threads ← Dictionary mapping workers to their current thread usage
node_threads ← Dictionary mapping nodes to their number of available threads
queue ← Task queue, which the provider sends tasks to

while Running do
    (header, data) = take(queue)
    n_threads = required_n_threads(header)

    /* Look for an already waiting worker with enough free
       resources on its node, (-1,-1) is returned if none is found
       */
    args = [worker_avail, n_threads, node_threads, worker_nodes]
    (worker, node) = find_waiting_worker(args)

    if worker ≠ −1 then
        send_task(worker, header, data, n_threads)
        node_threads[node] − = n_threads
        worker_threads[worker] = n_threads
        worker_avail[worker] = False
        continue
    end

    /* Receive requests until a worker on a node which has enough
       free resources is found                                    */
    while True do
        worker = receive_work_request()
        node = worker_nodes[worker]
        node_threads[node] + = worker_threads[worker]
        worker_threads[worker] = 0

        if node_threads[node] >= n_threads then
            send_task(worker, header, data, n_threads)
            node_threads[node] − = n_threads
            worker_threads[worker] = n_threads
            worker_avail[worker] = False
            break
        end
        else
            worker_avail[worker] = True
        end
    end
end
```

The number of worker MPI processes should be set according to the lowest thread usage. For instance, if the lowest thread usage is two threads per worker, the amount of worker MPI processes should be set such that:

$$n\_worker\_processes = \frac{total\_n\_threads}{2}$$

Here, *total_n_threads* is the number of threads available on all nodes combined.

When using a different number of workers with a different number of threads in various time intervals, the nodes become more prone to oversubscription. To avoid this, the DRA Scheduler keeps track of the available threads on each node and the number of threads each worker is currently using. When the scheduler receives a request from a worker, the scheduler can compute which node this worker belongs to and free the number of threads the worker previously used on the said node.
When the scheduler receives a request from a worker, but the number of available threads on the worker's node is insufficient for the current task, the scheduler stores the worker's availability and does not send the task to the worker. The number of threads needed for a task is mainly based on its ST dimension size.
Lastly, a worker waiting for a task sleeps for short time intervals to give up its resources to any other worker that might need them. Figure 3.13 illustrates the workers on one node from the target system with 28 cores when two threads each are used for the tasks.

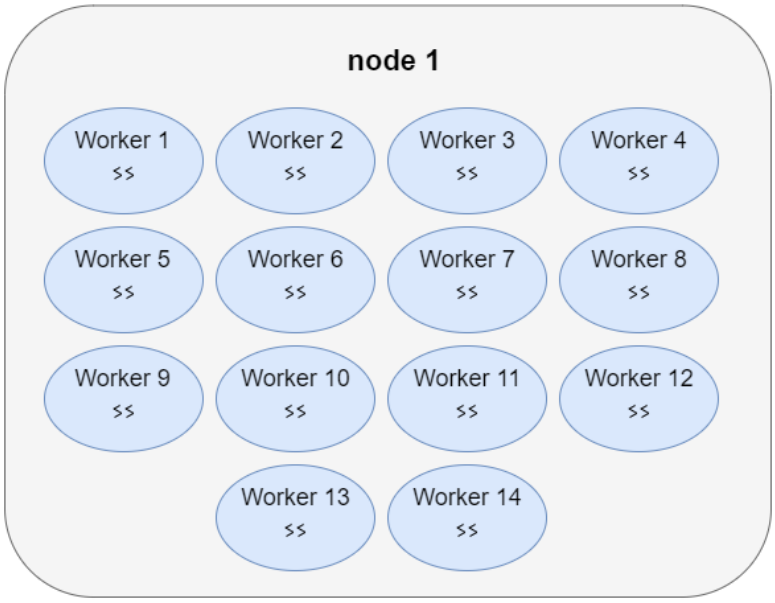


Figure 3.13: Example of workers when two threads each are used for the tasks.

The workers, when the tasks instead need eight threads each, are shown in Figure 3.14.

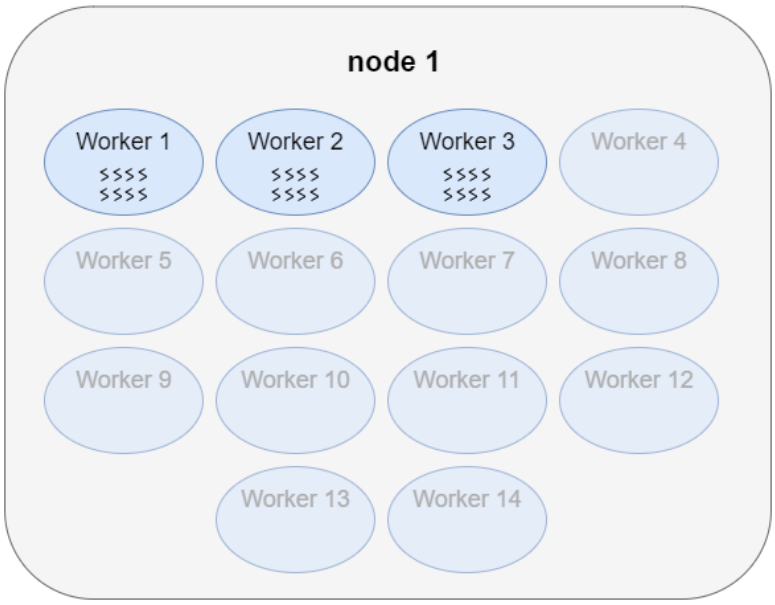


Figure 3.14: Example of workers when eight threads each are used for the tasks. Three workers are active, and the rest of the workers are idle.

3.7.3 Investigative Analysis of Iterative Improvements I

The research of the Subgroups Scheduler was concerned with how the latency behaves when adopting the subgroup strategy. In other words, how does the system latency behave if a scheduler optimizes worker execution time by running as few processes per node as possible?
The implementation of the Subgroups Scheduler provided certain observations. It managed to outperform the WI Scheduler slightly. Nonetheless, the resulting magnitude of the results, which can be seen in Table 3.2, was lower than expected. An insight from this implementation was that, in general, it seems to be preferable to run as few processes per node as possible.

When analyzing the results of the DRA Scheduler presented in Chapter 4, it was stated that there were no benefits of dynamically adjusting the number of workers and their number of threads in the current scenario. However, there were clear benefits of doing so in scenarios where resources were restricted and had to be used with a more conservative approach. In addition to this, a more computationally demanding **SP-chain** would also imply that the resources have to be used more conservatively. The benefit of using dynamic thread allocation under these circumstances was a discovery that proved valuable in a scheduling algorithm in the next iteration.

3.8 Iterative Improvements II

Iteration I resulted in valuable insights, which served as a foundation for the next iteration of improvements. There are certain benefits and drawbacks with a dynamic scheduler which were further researched in this section, and these implementations

also utilized the possibility of sending backup tasks.

3.8.1 Dual-Difficulty Scheduler

The tasks vary in data dimensions and computational demands. This consequently leads to some tasks failing to be completed before their respective deadlines, though given optimal prerequisites. Moreover, in a real application scenario, it is not always the case that the inflow of data is well-behaved. Outliers will likely occur. Outliers here are defined as tasks whose overall computational demands significantly deviate from the majority of nearby tasks with, e.g., the same block dimensions. The outliers in these representative scenarios were modeled as tasks with max intensity, and they could occur sporadically or in groups.

In the representative scenarios, the scheduler process allows for modifying the task's difficulty. This implies that a data block that would not succeed in meeting its deadline, though given optimal prerequisites, could instead run an easier difficulty and then be completed before its deadline. However, this was not desirable as a default behavior as the trade-off with running a less computationally intensive task represents that a real application scenario would produce less useful results. That is why the quota of tasks that run the hard difficulty had to be maximized.

Detecting outliers at the scheduling process is generally impossible in this application scenario, as the required computation for a task is largely data-dependent and unknown until a worker executes the task. Therefore, the Dual-Difficulty Scheduler (**DD Scheduler**) runs each data block with hard and easy difficulty and keeps the result from the hard worker only if it completes before its deadline. The DD Scheduler is described in pseudo-code in Algorithm 3.

Algorithm 3: Dual-Difficulty Scheduler

```
hard_comm ← MPI communicator for workers running difficulty=hard
easy_comm ← MPI communicator for workers running difficulty=easy
queue ← task queue, which the provider sends tasks to
time_slice ← Maximum time before canceling a hard worker's request
while Running do
    easy_request = receive_work_request(easy_comm)
    hard_request = receive_work_request(hard_comm)
    (header, data) = take(queue)
    Test for easy_request until received
    Test for hard_request until received, or time_slice has passed
    if hard_request takes longer than time_slice then
        cancel(hard_request)
    end
    if (hard_worker is available) then
        send_task(hard_worker, header, data, HARD)
    end
    send_task(easy_worker, header, data, EASY)
end
```

The most relevant result of this scheduler was that it meets the tasks' deadlines, which is an improvement compared to the WI Scheduler. The DD Scheduler achieved this due to the possibility of choosing a result that actually met the deadlines, preferring a hard one if managed in time.

An alternative approach to the DD Scheduler would involve, for a fixed worker, performing a task with an easy difficulty first, followed immediately by the same task with hard difficulty. This variation would use fewer resources while still succeeding in meeting the deadlines. Nevertheless, there are certain trade-offs with this variation. The quota of tasks completed with difficulty hard would be lower, as the overhead of performing an easy task before a hard task on the same worker would delay the hard task by a certain amount of time.

Still, the major drawback of the DD Scheduler is its excessive usage of SMT, high resource demand, and increased worker execution times due to memory contention. Each task is run once for both easy and hard difficulty, which is due to it being infeasible to detect outlier tasks at the scheduler process. A scheduler that uses the available resources of the cluster system more efficiently was thus desired.

3.8.2 Emergency Workers Scheduler

The Dual-Difficulty Scheduler proved to be a robust scheduling implementation, but the constant usage of backup workers led to the following:

1. Increased execution times due to memory contention, which was discovered when evaluating the Subgroups Scheduler.
2. Increased execution times due to excessive usage of logical cores.

There was a need for a scheduler which, if possible, only sent an easy task as a backup when it was actually needed.

The first idea was to send easy tasks to a specific group of workers if the task queue contained more than one element. Due to the nature of the deadlines (see section 3.3.3), a queue size greater than four would imply that deadlines are likely to fail before the tasks are even sent to the workers. Moreover, if there is a queue size greater than one, some or all workers are most likely busy with unusually time-consuming tasks which will not meet their deadlines. Consequently, there was a need always to keep track of all the tasks that the workers were currently working on to be able to send an easy version of the same task as a backup if needed.

The desired scheduling behavior was implemented and given the name *Emergency Workers Scheduler* (**EW Scheduler**). The EW Scheduler has two groups of workers of equal size: The ordinary workers and the emergency workers. The ordinary workers are always used and are only sent hard tasks. On the other hand, the emergency workers are only sent easy tasks and only under two circumstances:

1. When there is a queue size greater than one.
2. A hard task currently performed by an ordinary worker has reached a certain time limit and needs to be backed up.

The second condition is particularly difficult to determine. Finding optimal time thresholds for when to send backup tasks for different **ST** dimensions proved to be difficult. The introduction of the emergency workers implied that **SMT** had to be utilized, which implies an increased average execution time of the tasks (see Table 3.10 and **??**). Followingly, an equilibrium between robustness and performance was desired. More concretely, sending easy backup tasks too early could lead to increased latencies on the hard tasks, while sending them too late could lead to missing deadlines. Lastly, a worker waiting for a task sleeps for short time intervals to give up its resources to any other worker which might need them.
The scheduling algorithm of the EW Scheduler can be further studied in Algorithm 4.

Algorithm 4: Emergency Workers Scheduler

$o\_workers \leftarrow$ Collection containing the ordinary workers
$e\_workers \leftarrow$ Collection containing the emergency workers
$waiting\_o \leftarrow$ Collection containing the waiting ordinary workers
$waiting\_e \leftarrow$ Collection containing the waiting emergency workers
$queue \leftarrow$ Task queue populated by the provider
$tasks \leftarrow$ Dictionary mapping ordinary workers to their current tasks. Maintains a sorted order according to the start times of the tasks

```
while Running do
    send_backups(tasks, waiting_e, waiting_o, e_workers)
    if length(waiting_o) > 0 then
        worker = pop(waiting_o)
        (header, data) = take(queue)
        send_task(worker, header, data, HARD)
        tasks[worker] = (header, data)
        continue
    end
    if length(waiting_e) > 0 and length(queue) > 1 then
        worker = pop(waiting_e)
        (header, data) = take(queue)
        send_task(worker, header, data, EASY)
        continue
    end

    worker = receive_work_request()
    if worker ∈ o_workers then
        (header, data) = take(queue)
        send_task(worker, header, data, HARD)
        tasks[worker] = (header, data)
        continue
    end

    /* Request came from an emergency worker.  Only send it work if
       there is a queue size, otherwise save the worker for later.
       */
    if length(queue) > 1 then
        (header, data) = take(queue)
        send_task(worker, header, data, EASY)
    end
    else
        waiting_e.append(worker)
    end
end
```

The *send_backups* function iterates over the *tasks* dictionary and sends an easy version of the task to an emergency worker if necessary. If a backup is sent, the task is deleted from the dictionary so that multiple backups cannot be sent for the same task. The workers (if any) from the waiting emergency workers collection are used first. After that, work requests are gathered. If a work request from an ordinary worker is received in the *send_backups* function, that worker is saved for later in the *waiting_o* collection since ordinary workers must not be used for sending backups.

3.8.3 Further Optimizations

After evaluating the scheduling implementations, it was discovered that the EW Scheduler slightly outperformed the Dual-Difficulty Scheduler. That said, they both sacrificed a significant quota of successful hard tasks to run the easy backup tasks. It was established that the reason for this was the excessive usage of SMT.
Furthermore, the DRA Scheduler proved to be an efficient scheduling implementation when there was a need to use the resources efficiently. Followingly, an optimized version of the EW Scheduler that uses the dynamic thread allocation (from the DRA Scheduler) for its emergency workers was implemented. By doing so, resource contention could be minimized further, and the quota of hard tasks could potentially be increased.
The dynamic thread allocation for the emergency workers of the EW Scheduler utilized the resources more efficiently. As a result, the negative impacts of SMT were further minimized, and the execution time of the hard tasks was reduced.

3.9 MPI Implementation Details

The scheduling implementations utilize several MPI communication primitives, and these are all explained in further detail in this section.
To begin with, the common MPI communicators (logical groups) of all scheduling implementations can be seen in Figure 3.15.

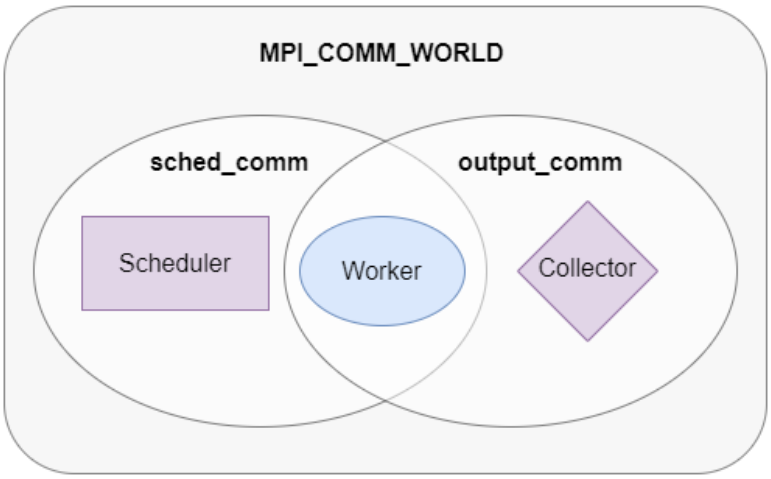


Figure 3.15: The common MPI communicators used in all scheduling implementations.

These communicators were merely used for convenience and the facilitation of debugging. The same functionality can be achieved without these. Furthermore, all communication in all scheduling implementations is non-blocking point-to-point communication, utilizing `MPI_Isend` and `MPI_Irecv`.

Up until the second iteration, the schedulers perform one send for each task it pops from the queue. The Baseline Scheduler always sends the current task to the next worker in line, while all the other schedulers rely on a work-request from a worker. The scheduler acknowledges the request from an idle worker by receiving from `MPI_ANY_SOURCE` in `sched_comm`, which permits receiving from any worker. The workers in these worker-initiated implementations always start by sending their MPI rank to the scheduler in `sched_comm`, which serves as a request. After this, which is common for all the workers, they receive a task, execute it, and send the result to the collector. Furthermore, the workers follow a master-only design where only the master thread can perform MPI calls, illustrated in Figure 3.16. The appropriate MPI thread level support for this is `MPI_THREAD_FUNNELED`, which is also used in all components of the implemented methods.

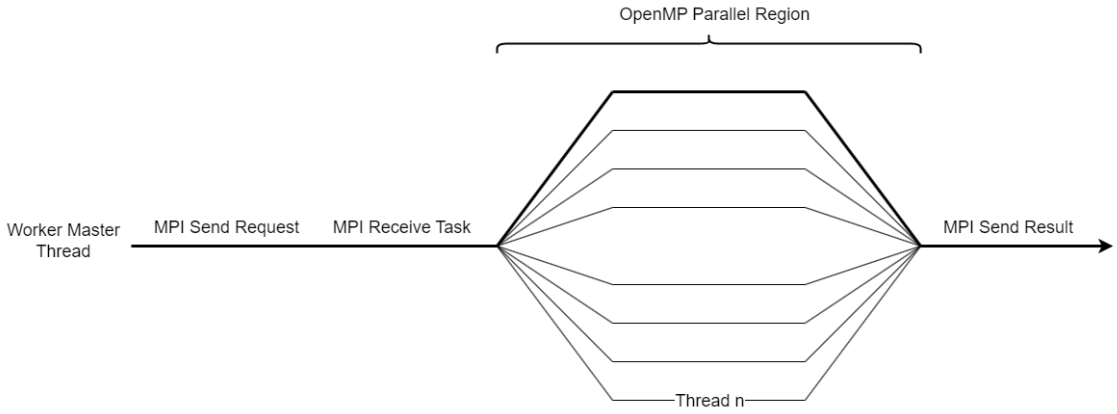


Figure 3.16: A simplified visualization of the MPI and OpenMP functionality of the worker.

The DD Scheduler always performs two sends for each task it pops from the queue, one with difficulty easy and one with difficulty hard. This leads to the double amount of sends and also the double amount of receiving actions. Moreover, the DD scheduler further divides the `sched_comm`. As a result, two `MPI_Groups` are created, one for the ranks of easy workers and one for the ranks of hard workers. These groups are then used to create two new communicators. Figure 3.17 shows an overview of this for two compute nodes.

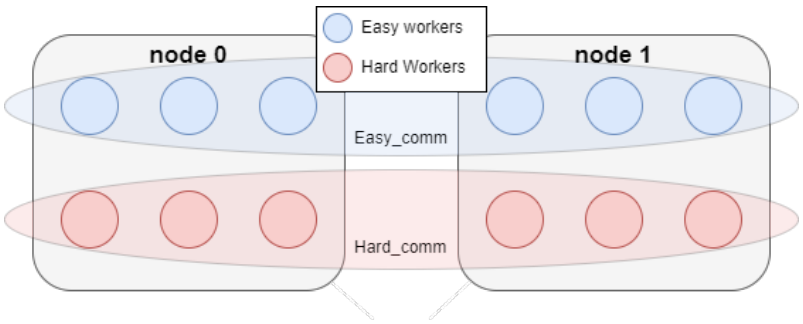


Figure 3.17: A simplified overview of communicators.

The EW Scheduler mitigates many send actions by only sending easy tasks when necessary. A comparison between the number of tasks sent and received for the DD Scheduler and the EW Scheduler can be seen in Table 3.3.

DD Scheduler			**EW Scheduler**		
#Hard tasks	**#Easy tasks**	**#Tasks**	**#Hard tasks**	**#Easy tasks**	**#Tasks**
13 351	13 351	26 702	13 351	4089	17 440

Table 3.3: Comparison of the number of tasks produced between the DD Scheduler and the EW Scheduler in a test run.

Resultingly, the number of MPI communication calls is drastically reduced in the EW Scheduler compared to the DD Scheduler. Lastly, the MPI functionality of

the collector is the same in all implementations; it listens for task results from `MPI_ANY_SOURCE` in `output_comm`.

3.10 Summary

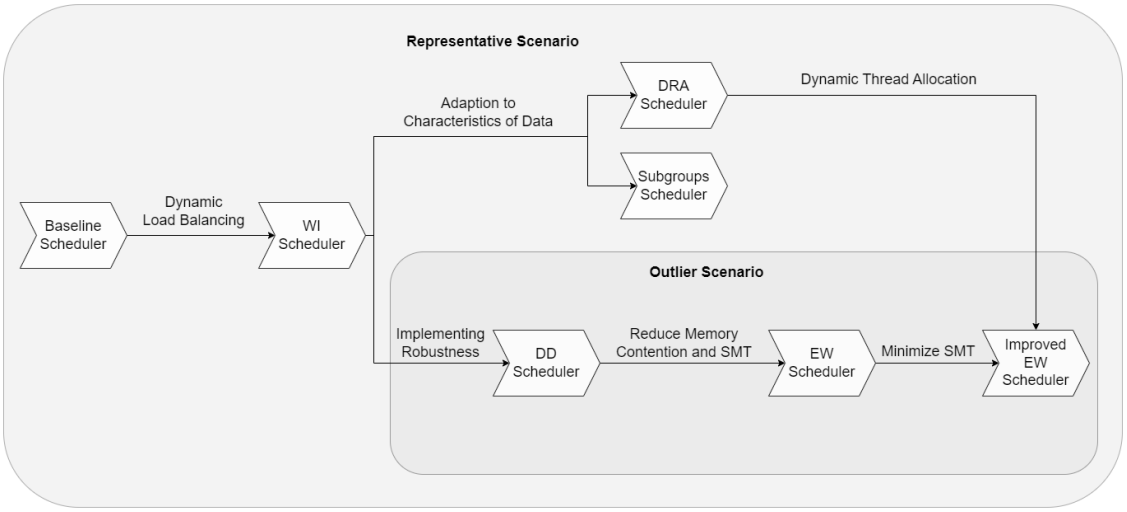


Figure 3.18: A simplified overview of the different scheduling implementations in this thesis.

Six different scheduling implementations were realized in addition to one optimized version of the EW Scheduler. These can be seen in Figure 3.18, which provides a simplified illustration of the methodology process. As should be apparent by now, there are more crucial details than the figure shows; it merely serves as a comprehensible visual aid.

A compact summary of the implemented schedulers in this thesis can be seen in Table 3.4.

Scheduler	Motivation	Properties
Baseline	Evaluation purposes	• Simple implementation • Round-robin • Blocking while currently watched worker is executing a task
WI	To put the process of finding an available worker onto already idle workers	• Worker-initiated communication • Less idle-time for workers • Indirect load balancing (with global task queue) • Static behavior
Subgroups	Adapt #workers to the workload to minimize resource contention	• Winnings by being dynamic with #workers • Magnitude rather low for small differences in #workers between subgroups • Many application-dependent parameters
DRA	Adapt resources/worker to a varying workload, allowing for more workers during high acquisition rates, and fewer during lower acquisition rates	• Benefits when resources are restricted • No significant improvements for current scenario
DD	Robustness in the ratio of met deadlines, as some tasks are infeasible even if given optimal prerequisites	• Sends easy backup tasks of every task • Meets all deadlines, even in an outlier-scenario • A significantly lower quota of finished hard tasks • Resource demanding
EW	More efficient resource management compared to the DD Scheduler	• Sends easy backup tasks when needed • Meets all deadlines, even in an outlier-scenario • Less resource demanding than DD Scheduler
Improved EW	Reduce excessive use of SMT by introducing dynamic thread allocation	• More efficiently utilize resources • Lower execution time of hard tasks

Table 3.4: Summary of implemented schedulers, with their motivation and properties

4

Evaluation

This chapter provides results attained from the different evaluation steps. The performance measurements are presented in plots, tables, and diagrams. The measurements are based on quantitative goals that are formulated based on the goals in section 1.4.1.

4.1 Prerequisites

To enable fair comparisons between different scheduling implementations, they had certain set prerequisites, which are listed below:

- **Allocation of workers:** The worker processes were allocated to nodes in the target environment such that each node had an equal number of workers.
- **Runtime:** The provider ran for 160 seconds.
- **Intensity:** All the implementations sampled the intensity from the same distribution with the same seed.
- **Difficulties:** If no explicit difficulty is mentioned, the tasks are run at a hard difficulty by default. The later scheduling implementations may run some tasks at an easier difficulty, and then a hard quota is introduced to measure the ratio of tasks completed with a hard difficulty.
- **Deadlines:** These were computed based on the largest data block dimensions as described in 3.3.3
- **Data:** The data block dimensions and inflow rate varied similarly for all implementations. This can be studied in more detail in section 3.3.1 and 3.3.2, respectively.

4.2 Worker Initiated Scheduler Compared to the Baseline Scheduler

In this section, the WI scheduler is compared to the Baseline Scheduler. The intention is to clearly show the advantages of a worker-initiated point-to-point communication of the hybrid programming model in this application scenario.
The latency is the total time it takes for a task to be sent by the provider until

the collector receives it. The average latency for the different ST dimensions and a different number of workers for both the WI Scheduler and the Baseline Scheduler can be seen in Figure 4.1.

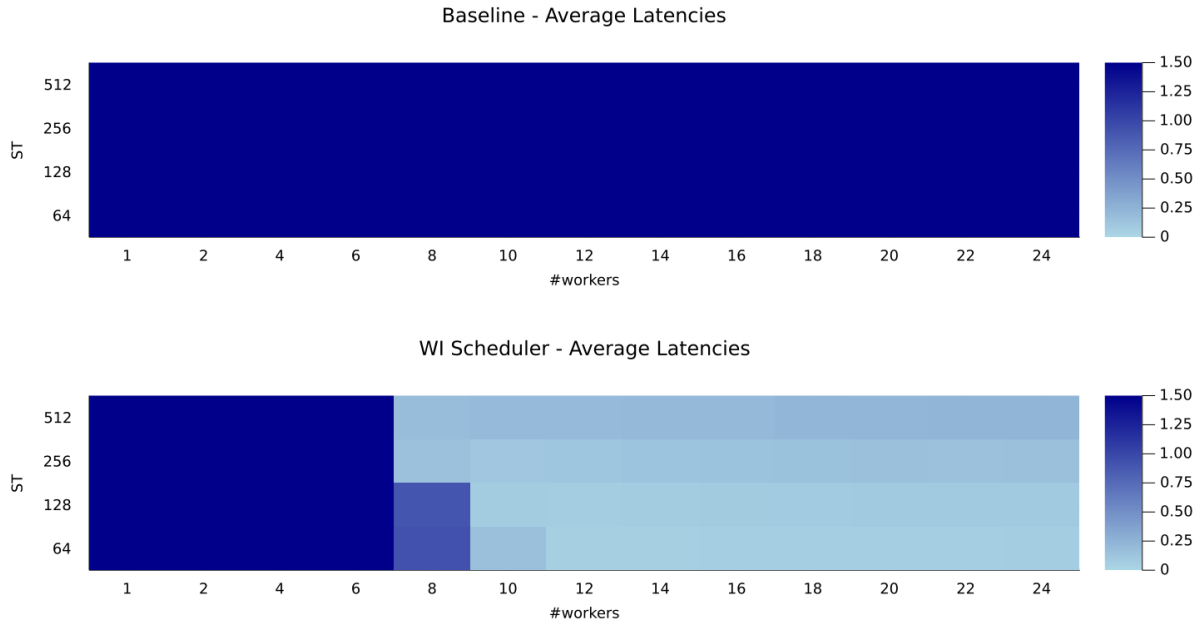


Figure 4.1: Average task latency from 0 to 1.5 seconds for the Baseline Scheduler and the WI Scheduler.

Figure 4.2 instead shows the quota of met deadlines for the different ST dimensions and a different number of workers for both the WI Scheduler and the Baseline Scheduler.

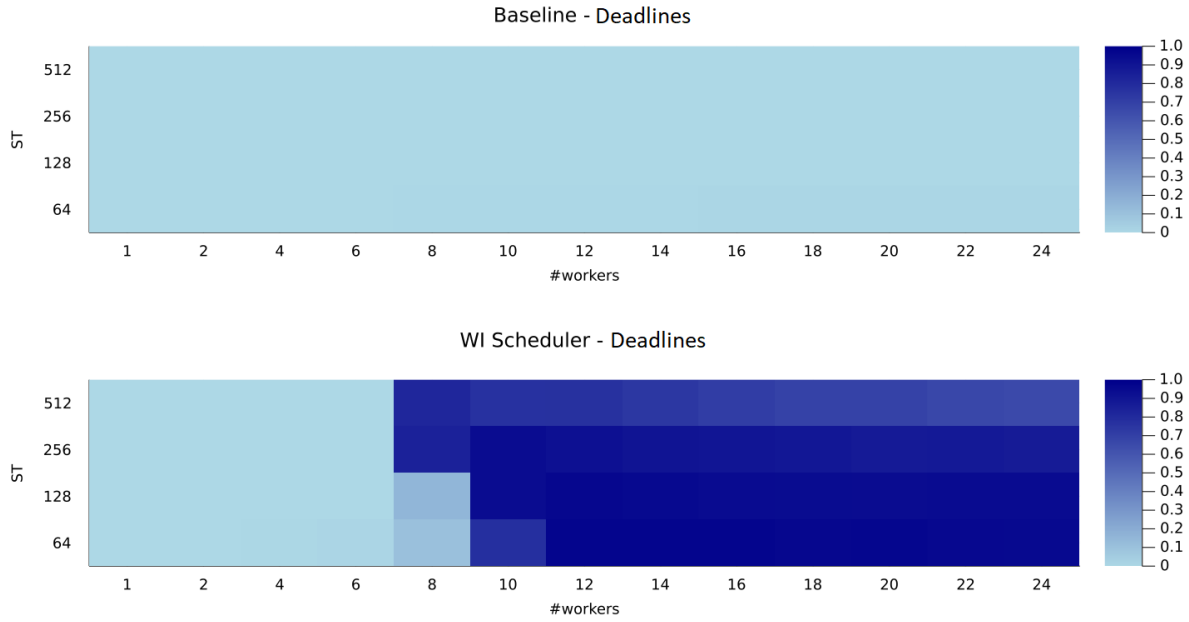


Figure 4.2: Quota of met deadlines for the Baseline Scheduler and the WI Scheduler. For instance, a value of 0.7 indicates that 70% of the deadlines were successful.

As seen in both Figure 4.1 and 4.2, there are major limits to the blocking point-to-point MPI communication in this application scenario. A more effective scheduling approach is achieved by considering worker-initiated, point-to-point communication where the scheduler directly knows when a worker is ready for a new task. The main reason for this is that the scheduler is no longer required to wait for workers to finish executing, as the burden of finding which worker should be assigned the next task is put onto the already idle workers. This is described in Sections 3.4 and 3.5 and illustrated in more detail in Figures 3.6 and 3.9.

The reasons behind the results of the above heatmaps can be more concretely illustrated by observing the relationship between the throughput of tasks and the data acquisition rate (the rate at which the provider populates the task queue). How the Baseline Scheduler adapted to the changes in inflow data over time for 4, 8, and 12 workers can be seen in Figure 4.3, Figure 4.4, and Figure 4.5 respectively.

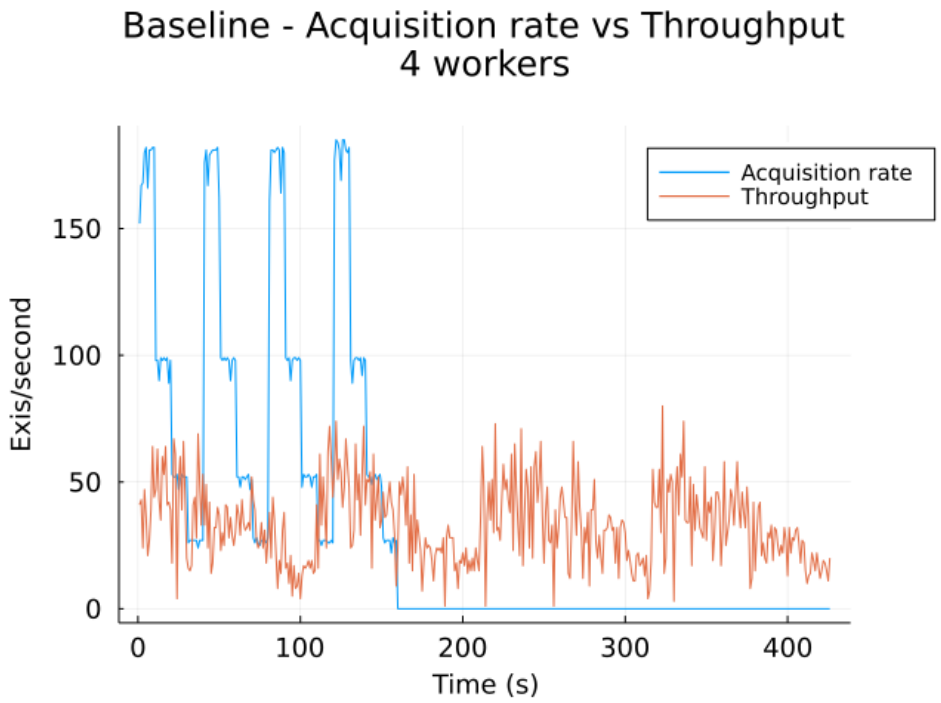


Figure 4.3: Correlation between acquisition rate and throughput for the Baseline Scheduler, 4 workers

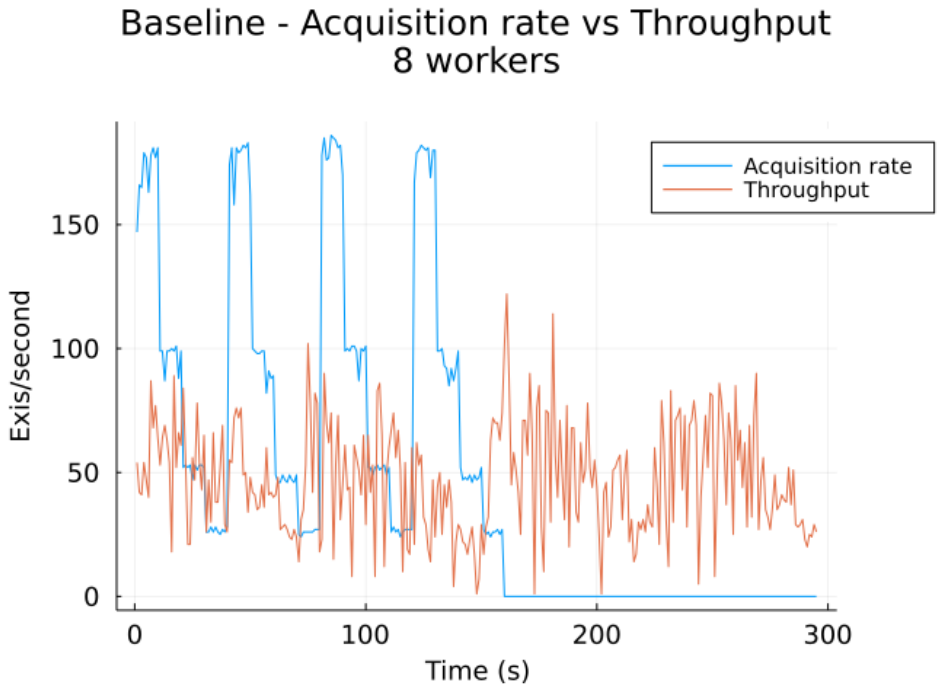


Figure 4.4: Correlation between acquisition rate and throughput for the Baseline Scheduler, 8 workers

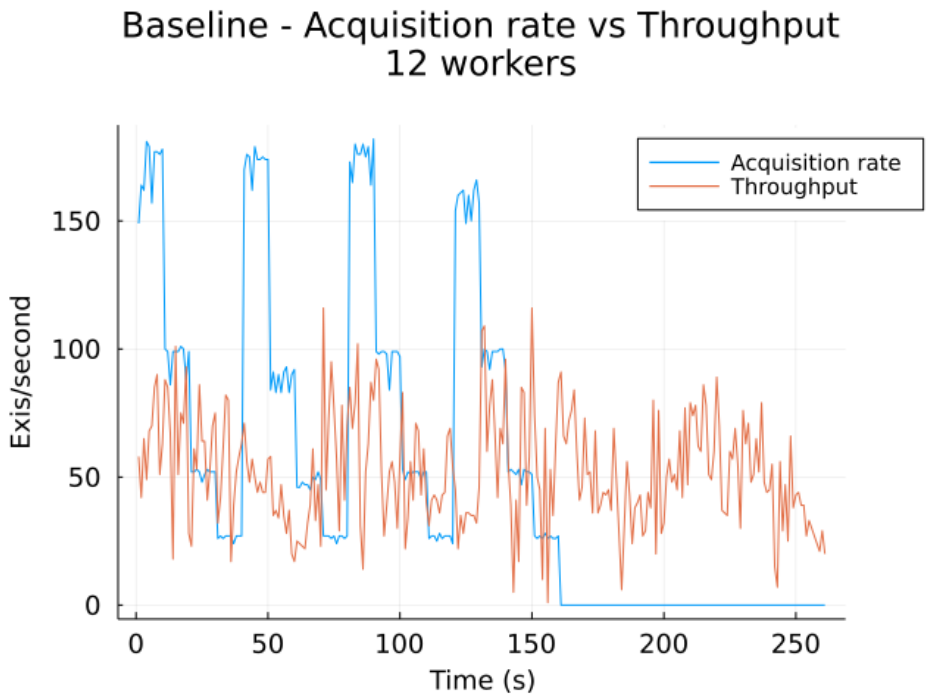


Figure 4.5: Correlation between acquisition rate and throughput for the Baseline Scheduler, 12 workers

How the WI Scheduler adapted to the changes in inflow data over time for 4, 8, and 12 workers can be seen in Figure 4.6, Figure 4.7, and Figure 4.8 respectively.

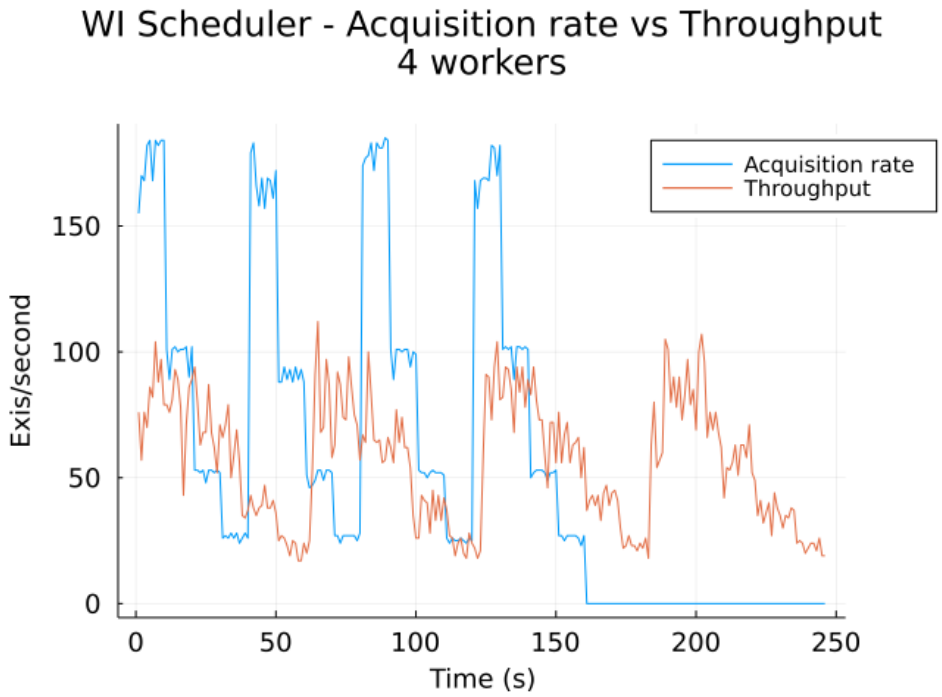


Figure 4.6: Correlation between acquisition rate and throughput for the WI Scheduler, 4 workers

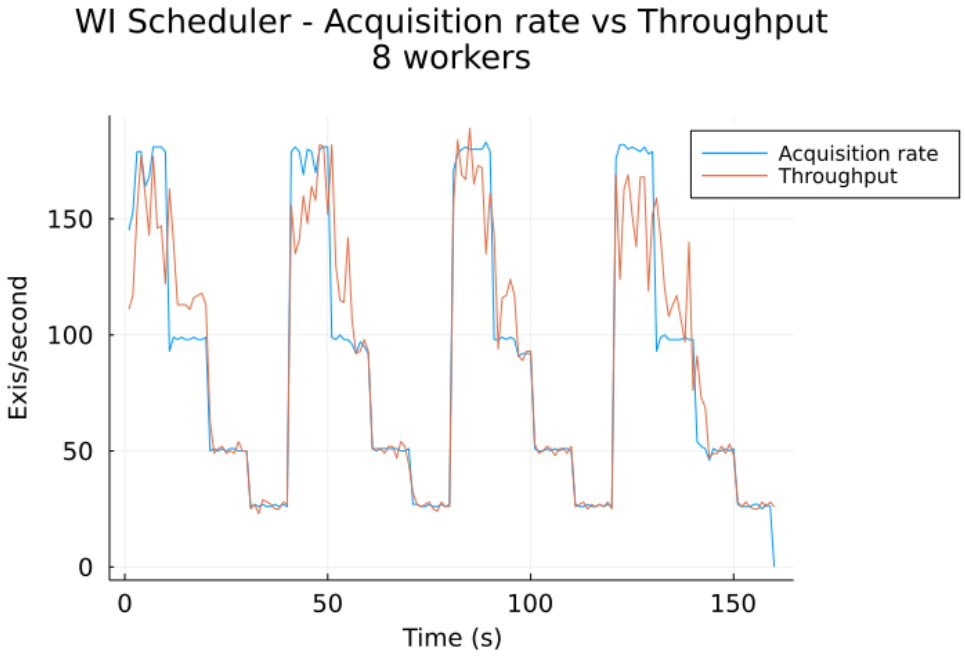


Figure 4.7: Correlation between acquisition rate and throughput for the WI Scheduler, 8 workers

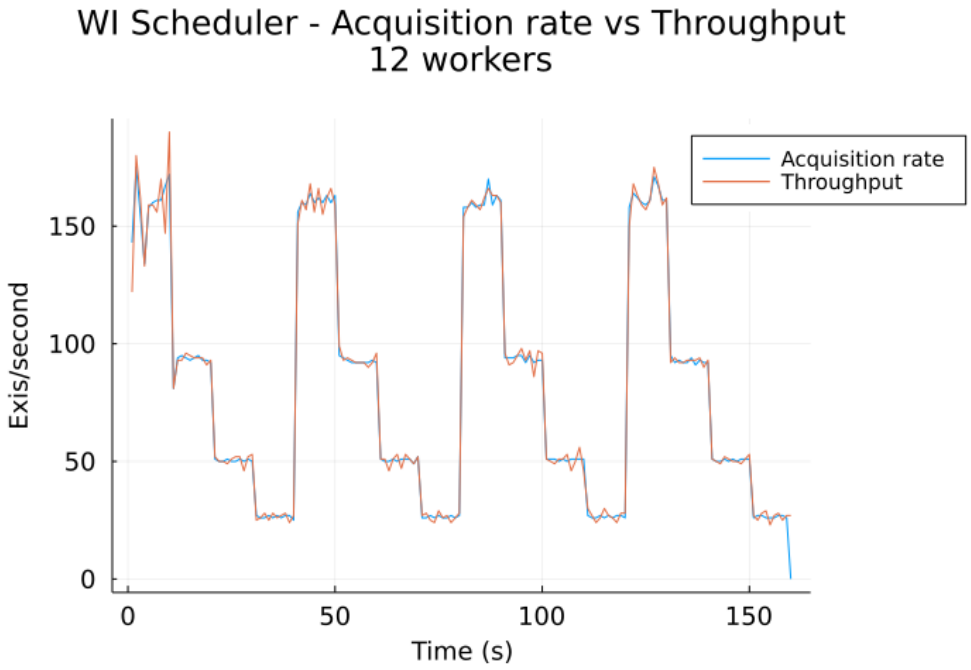


Figure 4.8: Correlation between acquisition rate and throughput for the WI Scheduler, 12 workers

As can be seen, a throughput rate that roughly follows the influx rate of data is a clear sign that the system is more capable of managing the inflow of data. In the case of the WI Scheduler, this becomes clear when observing Figure 4.8. Comparing this with the heatmaps in Figures 4.1 and 4.2, it can be observed that this setting is achieving comparably fair results in terms of latency and met deadlines as well. The reasons for this are the same as for the above heatmaps. The system has enough workers to manage the data inflow, it is dynamically load-balanced due to worker-initiated communication, and the scheduler does not need to block for a worker which is already executing a task.

4.3 Subgroups Scheduler

The number of workers per node directly affects individual workers' execution times, as shown in Table **??**. As the majority of each task's latency consists of actual worker execution time, the system could benefit from running as few processes per node as possible while maintaining its ratio of met deadlines.
The Subgroups Scheduler was compared to the WI Scheduler at its peak performance at 12 workers to allow for fair comparisons. Based on the results shown in Table 3.2, a suitable group division would be to run all 12 workers for the smaller ST dimensions (i.e., 64 and 128) while running a subgroup of eight workers for the larger ST dimensions (i.e., 256 and 512). The results from the Subgroups Scheduler are compared to the WI Scheduler for the same prerequisites in Table 4.1.

	WI Scheduler		**Subgroups (256,512)**	
ST	**Latency (ms)**	**Deadlines (%)**	**Latency (ms)**	**Deadlines (%)**
64	72.3	96.6	72.1	97.2
128	77.1	96.6	79.1	96.5
256	125.0	92.8	115.0	94.5
512	202.8	77.8	195.0	79.3

Table 4.1: WI Scheduler compared to Subgroups Scheduler. Per node, two workers belong to the subgroup, and one is a helper worker. The parenthesis (e.g. (256,512)) indicates the ST dimensions where only the subgroup is utilized.

The smaller ST dimensions (64,128) were run with the same number of workers for both scheduling methods, and the larger ST dimensions (256,512) were run with a subgroup of 8 workers in the case of the Subgroups Scheduler. It can be observed that there is not much of a change in latency and ratio of met deadlines for the smaller ST dimensions. However, the interesting result is that there is some increase in performance for the larger ST dimensions. This correlates to the observations made in Section 3.7.1 connected to memory contention and that running as few workers as possible can improve overall system performance.

Although there are benefits of being dynamic with the number of workers, the drawback of the results in this representative scenario is that the actual magnitude of the improvements may be small. In another scenario, where the difference is larger between the number of workers in a subgroup compared to all workers, the results

may be more prominent. The impact of memory contention may not be that large for this number of workers, which may not be a surprising result. However, these results point towards a tendency that as few workers as possible may yield improved performance results regarding latency and ratio of met deadlines. In summary, there are some winnings for a scheduler in adapting the number of workers to cope with a varying workload.
Group sizes, the condition for selecting a group of workers, and how many subgroups should be used are parameters that could be further optimized to provide more unambiguous results. These parameters are heavily application-dependent, and thus these specific values do not generalize well to any scenario, which is another drawback of this approach.

4.4 Dynamic Resource Allocating Scheduler

Due to the DRA Scheduler's ability to adapt to varying workloads, it could possibly perform well in the target domain. To evaluate the performance of the DRA Scheduler, it was compared to the WI Scheduler at its peak performance when using 12 workers. An important parameter that influenced the performance of the DRA Scheduler is the number of threads assigned for each task depending on its slow time dimension. A tested specific thread assignment can be seen in Table 4.2.

ST	**#threads**
64	4
128	4
256	8
512	8

Table 4.2: A thread assignment that was tested.

How the DRA Scheduler performed in this setting in comparison to the WI Scheduler can be seen in Table 4.3.

	WI Scheduler		**DRA Scheduler**	
ST	**Latency (ms)**	**Deadlines (%)**	**Latency (ms)**	**Deadlines (%)**
64	69	97	84	95
128	81	96	124	93
256	125	93	126	93
512	202	78	205	76

Table 4.3: A comparison of the average latencies and quota of met deadlines between the WI Scheduler and the DRA Scheduler using the thread assignment shown in Table 4.2

From Table 4.3, it can be seen that the DRA Scheduler performs roughly the same on the larger data blocks but worse on the smaller data blocks. This is because the WI Scheduler uses eight threads for the smaller data blocks, while the DRA Scheduler

only uses four threads. Perceptibly, the double number of workers that the DRA Scheduler can utilize when only using four threads per worker does not give it any advantage in this scenario. Using 12 workers with eight threads each is enough to handle the acquisition rate for all ST sizes, which means that the execution times of the tasks are the bottleneck. This execution time is minimized by giving the workers as many threads as possible. Evidently, it is not sensible to assign the workers fewer threads when they benefit from more, and the target system's resources allow to give them more. Many more thread assignment combinations were tested, but similar results were seen.

However, what if the resources of the system were restricted? Giving the required number of workers their maximum number of threads might not be feasible. An example could be a system with fewer nodes or nodes with fewer CPUs.
Two nodes were used instead of four to evaluate a scenario with restricted resources. Furthermore, only the physical CPUs of each node were used. The result from this run can be seen in Table 4.4.

	WI Scheduler		**DRA Scheduler**	
ST	**Latency (ms)**	**Deadlines (%)**	**Latency (ms)**	**Deadlines (%)**
64	16 145	1	118	94
128	19 824	0	134	92
256	21 000	0	218	72
512	21 376	0	399	25

Table 4.4: A comparison of the average latencies and quota of met deadlines between the WI Scheduler and the DRA Scheduler when using two nodes with 28 physical CPUs each.

The DRA Scheduler outperforms the WI Scheduler in this scenario. Under these circumstances, the WI Scheduler can only use six workers with eight threads each, which is insufficient (as reflected in the latency). However, how do the schedulers compare if the WI Scheduler instead uses roughly twice the amount of workers with four threads each? This was tested, and the results can be seen in Table 4.5.

	WI Scheduler		**DRA Scheduler**	
ST	**Latency (ms)**	**Deadlines (%)**	**Latency (ms)**	**Deadlines (%)**
64	358	27	118	94
128	235	68	134	92
256	222	72	218	72
512	393	28	399	25

Table 4.5: A comparison of the average latencies and quota of met deadlines between the WI Scheduler and the DRA Scheduler when using two nodes with 28 physical CPUs each. The WI Scheduler uses roughly twice as many workers with four threads each in comparison to the results from Table 4.4

As seen, the DRA Scheduler still outperforms the WI Scheduler in this scenario.
The thread assignment for the emergency workers, which worked the best in the

scenarios with the restricted resources, can be seen in Table 4.6. This was also the thread assignment used for the results in Table 4.4 and 4.5.

ST	#threads
64	2
128	4
256	4
512	4

Table 4.6: The thread assignment of the DRA Scheduler that gave the best results in the scenario with restricted resources.

4.5 Dual-Difficulty Scheduler

The motivation for the Dual-Difficulty Scheduler was to create a robust scheduling implementation that meets all deadlines at all times. To evaluate the DD Scheduler, the standard scenario was used in addition to a modified scenario where a series of abnormally heavy tasks occur sporadically.
The performance of the DD Scheduler in comparison to the WI Scheduler in the standard scenario and the outlier scenario can be seen in Table 4.7 and 4.8, respectively.

	WI Scheduler	DD Scheduler	
ST	Deadlines (%)	Deadlines (%)	Hard quota (%)
64	97	100	95
128	96	100	94
256	93	100	82
512	78	100	57

Table 4.7: A comparison of the quota of met deadlines between the WI Scheduler and the DD Scheduler in the standard scenario.

	WI Scheduler	DD Scheduler	
ST	Deadlines (%)	Deadlines (%)	Hard quota (%)
64	72	100	95
128	72	100	94
256	80	100	84
512	68	100	54

Table 4.8: A comparison of the quota of met deadlines between the WI Scheduler and the DD Scheduler in the outlier scenario.

In these tables, it is not reasonable to measure the latency anymore. The easy tasks are sent as a backup, which means their result can first be shown when the deadline passes if the hard version of the task is not finished. Consequently, the latency of easy tasks would always be the time of the deadline. In addition, the hard tasks' latency would also be misleading since a lower average latency of them could imply

that more easy tasks were used.
The performance of these scheduling implementations is instead reflected in the quota of hard tasks, while the overall quota of successfully met deadlines indicates their robustness.

From Table 4.7 and 4.8, it can be confirmed that the DD Scheduler is a robust scheduling implementation since it meets all the deadlines in both the standard scenario and the outlier scenario. This result was especially prominent in the outlier scenario where the WI Scheduler struggled while the DD Scheduler performed close to equally well as in the standard scenario.
That being said, the DD Scheduler uses a lot more resources that might not always be available. Another drawback of the DD Scheduler, which is notable in the standard scenario in Table 4.7, is that it sacrifices a big chunk of the hard tasks for the larger data blocks. For instance, the WI Scheduler meets 78% of the task deadlines for the hard tasks of the data blocks of ST size 512, while the DD Scheduler only manages 57%. This is because more resources have to be utilized in order to run the easy workers, which increases the execution time of the tasks (as can be studied in Table 3.10). As such, it should be used sparingly. Another factor influencing this decrease is a higher degree of memory contention.

4.6 Emergency Workers Scheduler

The EW Scheduler was an attempt to improve the DD Scheduler by not sending easy backup tasks of every task but only when necessary. This could lead to reduced average latencies since it reduces the execution time when logical cores are used, consequently minimizing overall resource contention. Table 4.9 and 4.10 show how the EW Scheduler performs in comparison to the DD Scheduler in the standard scenario and the outlier scenario, respectively.

	DD Scheduler		**EW Scheduler**	
ST	**Deadlines (%)**	**Hard quota (%)**	**Deadlines (%)**	**Hard quota (%)**
64	100	95	100	97
128	100	94	100	95
256	100	82	100	84
512	100	57	100	57

Table 4.9: A comparison of the quota of met deadlines between the DD Scheduler and the EW Scheduler in the standard scenario.

	DD Scheduler		EW Scheduler	
ST	**Deadlines (%)**	**Hard quota (%)**	**Deadlines (%)**	**Hard quota (%)**
64	100	95	100	97
128	100	94	100	95
256	100	84	100	83
512	100	54	100	56

Table 4.10: A comparison of the quota of met deadlines between the DD Scheduler and the EW Scheduler in the outlier scenario.

The EW Scheduler reaches a slightly higher quota of hard tasks than the DD Scheduler in both scenarios. However, a significant percentage of hard tasks still fail compared to the WI Scheduler in the standard scenario.

4.7 Further Optimizations

The main concern with both the DD Scheduler and the EW Scheduler was that the generous usage of SMT led to a decreased quota of successful hard tasks, though a node was not oversubscribed. SMT was advantageous for the implementations, but it should be used sparingly. To manage the logical cores efficiently, the optimized EW Scheduler utilized dynamic thread allocation for its emergency workers.

A comparison of results between the WI Scheduler, the DD Scheduler, and the improved version of the EW Scheduler in the standard scenario can be seen in table 4.11.

	WI Scheduler	DD Scheduler		EW Scheduler	
ST	**Deadlines (%)**	**Deadlines (%)**	**Hard quota (%)**	**Deadlines (%)**	**Hard quota (%)**
64	97	100	95	100	98
128	96	100	94	100	97
256	93	100	82	100	92
512	78	100	57	100	72

Table 4.11: A comparison of the quota of met deadlines between the WI Scheduler, DD Scheduler, and the improved EW Scheduler in the standard scenario.

To begin with, the improved version of the EW Scheduler reaches a significantly higher quota of hard tasks than the DD Scheduler. Secondly, the quota of successful hard tasks of the WI Scheduler and the EW Scheduler is roughly the same for every ST except for the largest data blocks. This signifies that the EW Scheduler sends easy backup tasks without exceedingly stalling the execution time of the hard tasks. The performance of the same three scheduling implementations in the outlier scenario can be seen in Table 4.12.

	WI Scheduler	DD Scheduler		EW Scheduler	
ST	**Deadlines (%)**	**Deadlines (%)**	**Hard quota (%)**	**Deadlines (%)**	**Hard quota (%)**
64	72	100	95	100	98
128	72	100	94	100	96
256	80	100	84	100	91
512	68	100	54	100	73

Table 4.12: A comparison of the quota of met deadlines between the WI Scheduler, DD Scheduler, and the improved EW Scheduler in the outlier scenario.

The results from Table 4.12 is yet another verification that the improved version of the EW Scheduler delivers robustness under unpredictable workloads while still maintaining good performance. Lastly, the thread assignment of the emergency workers of the EW Scheduler can be seen in Table 4.13.

ST	**#threads**
64	1
128	1
256	2
512	2

Table 4.13: The thread assignment of the emergency workers of the EW Scheduler that gave the best results.

4.8 Summary

This section contains two summarized tables for all scheduling implementations, one for the standard scenario and one for the outlier scenario. How the different scheduling implementations perform in the standard scenario can be studied in Table 4.14, and how they perform in the outlier scenario can be studied in Table 4.15.

	ST			
	64	**128**	**256**	**512**
Schedulers	**DL | HQ (%)**	**DL | HQ (%)**	**DL | HQ (%)**	**DL | HQ (%)**
Baseline	0 | 0	0 | 0	0 | 0	0 | 0
WI	97 | 97	96 | 96	93 | 93	78 | 78
Subgroups	97 | 97	97 | 97	95 | 95	79 | 79
DRA	95 | 95	93 | 93	93 | 93	76 | 76
DD	100 | 95	100 | 94	100 | 82	100 | 57
EW	100 | 97	100 | 95	100 | 84	100 | 57
Improved EW	100 | 98	100 | 97	100 | 92	100 | 72

Table 4.14: A summarization of how all the schedulers perform in terms of successfully met deadlines (DL) and quota of successful hard tasks (HQ) in the standard scenario.

	ST			
	64	**128**	**256**	**512**
Schedulers	**DL | HQ (%)**	**DL | HQ (%)**	**DL | HQ (%)**	**DL | HQ (%)**
Baseline	0 | 0	0 | 0	0 | 0	0 | 0
WI	72 | 72	72 | 72	80 | 80	68 | 68
Subgroups	72 | 72	75 | 75	72 | 72	56 | 56
DRA	90 | 90	89 | 89	79 | 79	64 | 64
DD	100 | 95	100 | 94	100 | 84	100 | 54
EW	100 | 97	100 | 95	100 | 83	100 | 56
Improved EW	100 | 98	100 | 96	100 | 91	100 | 73

Table 4.15: A summarization of how all the schedulers perform in terms of successfully met deadlines (DL) and quota of successful hard tasks (HQ) in the outlier scenario.

5

Discussion

This chapter brings up general project discussions, e.g., concerning methodology and concepts related to the field. Reflections on key results, their connection to the research questions, and the applications and ethical impacts of the thesis are also brought up here.

5.1 Connecting Outcomes with Research Questions

The stated research questions in section 1.3 and the results' connection to these will be concretely summarized and discussed in this section.

> 1. How does the choice of scheduling technique impact the overall system latency and load balancing of a real-time signal processing application?

The most drastic change in terms of load balancing occurred when switching from the static load balancing in the Baseline Scheduler to the dynamic load balancing in the WI Scheduler. This change proved critical in a real-time signal processing application which became apparent when the results of the runs were presented. The Baseline Scheduler failed to produce satisfactory results independently of how many workers it could use. Followingly, worker-initiated communication was used in all the other scheduling implementations. Any further optimizations to the load balancing were deemed difficult due to the unforeseeable execution times of the signal processing chains and, thus, not attempted.
In a hypothetical scheduling implementation with partitioned task queues, a more direct approach to load balancing would be required. However, since almost any queue size implied failure, a partitioned task queue was never considered. Thus, the global task queue and the worker-initiated communication implied that fair load balancing was indirectly applied in this project.

The DRA Scheduler and the Subgroups Scheduler attempted to improve the system latency further by adapting to the workload characteristics. There were some slight winnings to gain, but neither of these scheduling techniques drastically impacted the overall system latency when running on the target system in the representative scenario. Despite that, there existed scenarios where both of them had or could significantly impact the system latency. This shows that the scheduling techniques

affect the overall system latency in different ways under different circumstances. These circumstances could, for instance, be a different target system, a more demanding or varying signal processing chain, or greater ranges of data block sizes. A universal and portable scheduling technique that positively affects the system latency under all circumstances remains a daunting task.
However, it was proven possible to keep the system latency under a certain threshold (the deadline) by utilizing scheduling techniques that required having fall-back workers that could perform an easier version of a task. In this thesis, the Dual-Difficulty Scheduler and the EW Scheduler implemented such techniques. Nonetheless, this technique requires the target system to have enough resources to allow for it, which might not always be the case.

> 2. How can a scheduling approach adapt to varying and unpredictable workload conditions in real-time signal processing applications?

The varying and unpredictable workload conditions were first attempted to be handled by trying to optimize the overall latency. This worked to a certain degree, but there exist cases where the task was too time-consuming to finish before the deadline, even under optimal circumstances. To be more precise, a worker's execution time in itself would be greater than its deadline even when the worker had optimal prerequisites. Due to the nature of real-time signal processing applications where all deadlines should be met, the scheduler was responsible for ensuring this did not bottleneck the system. A way of preventing this was to perform a less computationally heavy signal processing chain when needed. This possibility was shown to be essential in this application scenario and evaluated as the most robust strategy. It was further demonstrated that performance did not have to be excessively sacrificed if the target system had enough resources.
There are various ways to implement such strategies, two of which were tested and evaluated. The best combination of performance and robustness in this thesis was reached when the number of easy backup tasks was minimized as much as possible. The purpose of this is to minimize memory contention and the use of simultaneous multithreading, which were shown to be two significant factors that impacted the worker execution time.

> 3. What are the possibilities and challenges associated with developing a scheduler for signal processing applications, utilizing cluster-oriented programming models?

The most obvious possibility the results indicate is that a scheduler implemented with a cluster-oriented programming model could be utilized in signal processing applications to meet the ever-increasing demands for throughput and task latency. As a consequence, there is a possibility to run more computationally demanding and varying SP-chains, as the tasks can be scheduled across multiple nodes. Then, the resource- and computationally-intensive tasks will cause less of an issue for overall system performance. However, implications such as contention for memory resources must be considered, as that overhead may result in degraded system performance.

A scheduler developed with a cluster-oriented programming model will imply challenges concerning portability. In order to reach an ideal and acceptable performance,

some custom tailoring is needed to get the most out of the available system. This further implies that a scheduler performing well in one type of environment does not necessarily perform well in another. Things that limit this portability and thus need to be considered in each scheduling scenario are, for example, the usage of logical cores, the **NUMA** architecture, and the interconnection topology. Another challenge that may occur when developing a scheduler with a cluster-oriented programming model might be that different methods for distributed- and shared-memory communication are applied, as with the hybrid programming model. This will cause increased complexity as the system becomes more challenging to manage and extend.

5.2 Hybrid Programming Model

After consideration of the problem statement, purpose, and research questions in this thesis, the hybrid programming model (which utilizes both MPI (Message Passing Interface) and OpenMP) was chosen as the central software architecture. A primary goal of the implementations, beyond adequate performance, is utilizing the available resources of the target environment. This was best attained by considering the two-level parallelism approach. This thesis did not consider a model based on pure MPI, as the representative scenarios would take significantly longer to implement if no reusable components from the stakeholder could be utilized. However, a model based on pure MPI may perform slightly better than the hybrid programming model for lower core counts, as it is likely that the hybrid model introduces overheads in that regime, resulting in decreased performance [27]. However, this trend only continues for a while, as the hybrid programming model can outperform pure MPI at larger core counts [27]. Therefore one should not, in general, expect a hybrid programming model to outperform a pure MPI programming model. Nevertheless, this thesis has not considered the possible benefits of pure MPI.

The hybrid programming model introduces a few drawbacks connected to this application scenario. Firstly, there is an additional development overhead as two separate libraries are utilized to achieve parallelism. Secondly, as an intended goal was that a scheduler should be able to adapt to unpredictable workloads, a dynamic architecture would likely be needed for best performance. The hybrid programming model, especially MPI, generally considers a static allocation of processes to the nodes in a computer cluster. Therefore, some implemented methods spawn a large number of processes per node while dynamically choosing which processes to utilize based on some criteria. The initial number of spawned processes had to be set statically, which does not scale well when additional hardware is added or when the representative scenarios change regarding data acquisition rate, block dimensions, or intensity. Anyhow, this may be the preferable way in a real-time application as there is an inevitable overhead with starting and stopping processes. This may cause performance issues regarding latency and the ratio of met deadlines, which are crucial parameters in a real-time system.

5.3 Task Queue Design

Generally, a task queue that grows too large during run-time is undesirable in real-time applications. For this application scenario, this is undesired for both a global task queue and potential partitioned queues. This is motivated by the fact that the tasks have their respective tight deadlines, and if a task gets stuck in a queue, the task in question will most certainly miss its deadline. Therefore, a task queue that stores too many tasks is undesirable. As such, as long as there was not a bottleneck in emptying the global queue, partitioned queues would serve little additional benefits.

However, scalability issues are connected to global queue design, as it often is with global data structures. If the distribution of tasks from the scheduler becomes the bottleneck in the system, the overall system will benefit more from having partitioned queues. Then tasks can be immediately distributed to the workers instead of stored at the scheduler. The workers would require more resources as additional threads would be needed to constantly listen for incoming data. This should not impact the execution time of the worker, as the blocking call will solely affect the calling thread. If scalability becomes an issue due to a bottlenecking scheduler, the question regarding how to best manage load balancing between the partitioned queues would be crucial to investigate further. Most likely, an adaptation of work-stealing would be suitable in this scenario.
A global task queue was employed since the scheduler did not become a significant bottleneck in this signal processing application. If the prerequisites for, e.g., data acquisition changes, a design including partitioned queues may very well be the most performant design.

5.4 Scheduling Schemes

In this application scenario, the priority scheme by which tasks are scheduled directly correlates to the tasks' deadlines. Since the intention is that there should be a minimal queue at all times, there is little to no choice in choosing the next task to execute. Rather it is the task with the earliest deadline that gets scheduled first. As the deadlines for all tasks are based on the largest block dimensions, the first-in-first-out scheduling that is adopted is correlated to **EDF** in this application scenario.
The varying workload in this application scenario suggests that some form of dynamic scheduling scheme is required. However, it would be infeasible to employ a fixed scheduling scheme like the rate-monotonic scheduling algorithm presented by Liu and Layland [23], as the behavior of the tasks is not known beforehand.

5.5 Experimental Methodology

The experimental methodology of this thesis followed an iterative process, where results and outcomes from different evaluations served as the foundation for the subsequent investigated methods. The selection of which outcomes to further investigate

in a new iteration was primarily influenced by the findings in the literature. A global task queue for scheduling in a computer cluster, the impact of **SMT** and memory contention, receiver-initiated communication, and scheduling schemes such as **EDF** were some of the outcomes that were considered more significant. By leveraging the knowledge in the field, this methodology aimed to address the research questions by allowing for iterative refinement of the scheduling methods.

This iterative methodology may have caused some limitations on the scope of the thesis. The iterative process focuses on iterative improvements, implying that there may be alternative methods that were not explored in this thesis. For instance, while considering how to concretely address the third research question and the overall scenario, a hybrid programming model was deemed well-suited. Therefore, the focus of the thesis mainly revolved around this programming model. Therefore, alternative approaches, such as pure MPI or systems based entirely on different models like *Remote Procedural Calls* (RPC)[1], were not significantly investigated in how they might have suited this scenario.
While the iterative approach might have narrowed the scope of the thesis, it also allowed the research to concentrate on more promising outcomes. In other words, by narrowing down the research, more focus could be put on the details and quality of the contributions from this thesis.

Lastly, to construct representative scenarios, this experimental methodology implies that some time had to be spent on understanding the characteristics and features of radar and signal processing systems. This allowed the research to be consistent and for the results to be more readily comparable in order to address the research questions more concretely. However, the time spent on this did not directly contribute to the results of the thesis, and as such, the quality of the results might suffer from additional time constraints.

5.6 Applications and Ethical Considerations

The importance of the thesis work is derived from implementing and evaluating different scheduling methods tailored for real-time signal processing applications. As the evaluations are based on characteristics typical to signal processing applications, such as deadlines and varying intensive workloads, the results presented in this thesis contribute to the knowledge of computer clusters' suitability for these applications. Cluster real-time scheduling is not exclusively interesting in signal processing applications, and the general results presented in this thesis may apply to other domains beyond signal processing. However, as some investigated scheduling methods utilize the option for running an easier rather than a hard task, these results are tailored to domains where this choice can be applicable.

This thesis focuses on signal processing applications in radar systems, specifically how the data acquired by an **AESA** radar can be effectively scheduled and processed in a cluster environment. A central ethical element concerning radar systems is the dual use of the technology. The outcomes from and the focus of the thesis do not

[1] https://grpc.io/

directly impact how a radar system should be used, as it is solely concerned with the signal processing of the application. Nevertheless, the nature of radar systems allows the technology to be applicable in multiple contexts.

The use of cluster environments is a trend that continues into the foreseeable future, as the evergrowing demand for processing power is prominent in several domains, including radar systems. Adapting tasks such that parallel execution is possible in distributed and shared memory architectures can enable more efficient real-time data processing. This implies that these systems become increasingly resource- and energy-demanding. As such, there could be some level of environmental impact caused by the demand for increased processing power. The outcomes from the thesis may further motivate the use of computer clusters.

6

Conclusion

In this thesis, dynamic load balancing resulting from worker-initiated communication was proven to make an advantageous difference in task latency and quota of successful deadlines. Moreover, simultaneous multithreading was beneficial in scheduling implementations when more workers were needed. However, it should be used sparingly as it, together with memory contention, cause increased worker execution times. Thus, there were small winnings in adjusting the number of workers in order to adapt to varying and unpredictable workloads. Theoretically, there could be even larger winnings if there is a more considerable difference in the number of workers per node needed for different periods of slow time dimensions. Furthermore, dynamically adjusting the number of workers and their number of threads as the workload varied proved promising in scenarios where hardware resources are restricted.

To achieve a robust scheduling implementation, backup workers performing easy versions of the hard tasks were essential. Running an easy version of every task was a reliable solution, but overusing it resulted in degraded performance. The best combination of performance and robustness was achieved by a scheduling algorithm that sent as few easy backup tasks as possible in order to minimize the negative impact caused by SMT and memory contention. In addition, the negative impacts caused by SMT could be further minimized by using as few threads as possible for the workers performing easy tasks, which involved dynamic thread allocation.

A scheduler for real-time signal processing tasks implemented using a cluster-oriented programming model has the possibility of processing more data compared to a single multiprocessor environment. The programming model used should be capable of effectively using available resources, and a mixed hybrid model could be preferable to a fully hybrid model in this thesis' application scenario and target system. This is to make effective use of large core counts per processor and other resources such as memory and network bandwidth. The challenges in the development of such a scheduler can involve the usage of multiple parallel programming libraries. It may also be challenging to cope with a varying workload when a static allocation of processes to nodes is required. Moreover, there will likely be the need for custom tailoring to the hardware architecture to achieve optimal performance. Consequently, this results in a scheduler developed using cluster-oriented programming models that might not be perfectly scalable or portable.

To summarize the outcomes of developing a domain-specific scheduler, considering the hardware architecture in collaboration with the software programming models

is of utmost importance. The interplay between these should be the guiding light when designing and implementing a system-level scheduling implementation.

6.1 Future Work

Several identified topics would have been interesting to explore further. However, due to the time constraints of this project, these topics are presented here as suggestions for future research.

6.1.1 Heterogeneous Clusters

This project's scope was limited to homogeneous computing systems. An interesting aspect to further look into would be *heterogeneous* multi-processing systems, i.e., a system where some processing units are tailored to execute certain operations more efficiently. This could, for instance, imply hardware such as FPGA, GPU, or ASIC. This would add another dimension of complexity to the scheduling methods. The results could be of great interest to domains such as embedded cluster systems, which are often tailored systems designed to handle specific tasks.

6.1.2 Alternative Programming Models

This thesis has considered certain programming models deemed well-suited to address the research questions. Nevertheless, it would be of great interest to this research to investigate other programming models and their ability to schedule real-time signal processing tasks in a cluster environment. The methods investigated in this thesis could then suit as inspiration for other methods in other approaches or as a comparison where task latency, throughput, and rate of met deadlines are important measurements.
A programming model could be composed of pure MPI, where MPI collective calls inside each task could be used instead of OpenMP parallel regions. Another interesting approach would be to see the impact of utilizing the one-sided communication of MPI, where *remote memory access* (RMA) is employed to move data without requiring process synchronization. At last, a final suggestion for another approach would be to use a system based on other types of message-passing, e.g., RPC.

6.1.3 Scheduling Schemes

There are several possibilities for future investigation in the area of scheduling schemes. A particularly interesting aspect to further research is the utilization of preemption in task scheduling. A concrete example would be if preemption could be used in the Dual-Difficulty Scheduler presented in this thesis. If a worker performing a hard task recognizes that the deadline would be missed, it could allow another task to preempt it and thus free up resources that other tasks could use. This would require the research of a preemption procedure that can effectively judge when a task will miss its deadline, but it may improve the scheduling performance.

Bibliography

[1] B. Edde, *Radar : principles, technology and applications.* Prentice Hall, 1993, ISBN: 0137523467. [Online]. Available: https://search.ebscohost.com/login.aspx?direct=true&db=cat09075a&AN=clpc.oai.edge.chalmers.folio.ebsco.com.fs00001000.96e446e2.9762.4191.acec.a834c4265af5&site=eds-live&scope=site&authtype=guest&custid=s3911979&groupid=main&profile=eds.

[2] J. Budge Mervin C. and R. German Shawn, "14. aesa basics and related topics.," in *Basic Radar Analysis (2nd Edition)*. Artech House, 2020, ISBN: 978-1-5231-4594-2. [Online]. Available: https://search.ebscohost.com/login.aspx?direct=true&db=edsknv&AN=edsknv.kt012ZRVR7&site=eds-live&scope=site&authtype=guest&custid=s3911979&groupid=main&profile=eds.

[3] 1. Richards M. A (Mark A.), *Fundamentals of Radar Signal Processing, Third Edition.* McGraw Hill LLC, 2022, ISBN: 1-260-46872-0. [Online]. Available: https://search.ebscohost.com/login.aspx?direct=true&db=edsace&AN=edsace.ACE2205310001&site=eds-live&scope=site&authtype=guest&custid=s3911979&groupid=main&profile=eds.

[4] M. Flynn, "Flynn's taxonomy," in *Encyclopedia of Parallel Computing*, D. Padua, Ed. Boston, MA: Springer US, 2011, pp. 689–697, ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_2. [Online]. Available: https://doi.org/10.1007/978-0-387-09766-4_2.

[5] A. S TANENBAUM and H. Bos, *Modern operating systems*. 2015.

[6] T. Rauber and G. Rünger, *Parallel programming*. Springer, 2013.

[7] F. Nielsen, "Topology of interconnection networks," in *Introduction to HPC with MPI for Data Science*. Cham: Springer International Publishing, 2016, pp. 63–97, ISBN: 978-3-319-21903-5. DOI: 10.1007/978-3-319-21903-5_3. [Online]. Available: https://doi.org/10.1007/978-3-319-21903-5_3.

[8] J. Subhlok, J. M. Stichnoth, D. R. O'hallaron, and T. Gross, "Exploiting task and data parallelism on a multicomputer," in *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, 1993, pp. 13–22.

[9] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *Proceedings of the 22nd annual international symposium on Computer architecture*, 1995, pp. 392–403.

[10] D. Culler, J. P. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998, ISBN: 9780080573076.

[11] F. C. H. Lin and R. M. Keller, "The gradient model load balancing method," *IEEE transactions on software engineering*, no. 1, pp. 32–38, 1987.

[12] P. Berenbrink, T. Friedetzky, and L. A. Goldberg, "The natural work-stealing algorithm is stable," *SIAM Journal on Computing*, vol. 32, no. 5, pp. 1260–1279, 2003.

[13] J. Dinan, S. Olivier, G. Sabin, J. Prins, P. Sadayappan, and C.-W. Tseng, "Dynamic load balancing of unbalanced computations using message passing," in *2007 IEEE International Parallel and Distributed Processing Symposium*, 2007, pp. 1–8. DOI: 10.1109/IPDPS.2007.370581.

[14] A. Karlsson, "The julia programming language for embedded high-performance signal processing in radar systems," M.S. thesis, Chalmers University of Technology, Chalmersplatsen 4, 2020.

[15] *Mpich*, Last accessed 10 April 2023. [Online]. Available: https://www.mpich.org/.

[16] *Open mpi: Open source high performance computing*, Last accessed 10 April 2023. [Online]. Available: https://www.open-mpi.org/.

[17] S. Byrne, L. C. Wilcox, and V. Churavy, "Mpi.jl: Julia bindings for the message passing interface," *Proceedings of the JuliaCon Conferences*, vol. 1, no. 1, p. 68, 2021. DOI: 10.21105/jcon.00068. [Online]. Available: https://doi.org/10.21105/jcon.00068.

[18] *Choosing how many mpi tasks and computational threads to use*, Last accessed 11 May 2023, 2021. [Online]. Available: https://www.ibm.com/docs/en/pessl/5.3.0?topic=ctopp-choosing-how-many-mpi-tasks-computational-threads-use.

[19] R. A. Tau Leng, J. Hsieh, V. Mashayekhi, and R. Rooholamini, "An empirical study of hyper-threading in high performance computing clusters," *Linux HPC Revolution*, vol. 45, 2002.

[20] B. Chapman, G. Jost, and R. Van Der Pas, *Using OpenMP: portable shared memory parallel programming*. MIT press, 2007.

[21] R. Rabenseifner, G. Hager, and G. Jost, "Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes," in *2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, 2009, pp. 427–436. DOI: 10.1109/PDP.2009.43.

[22] E. Saillard, "Static/dynamic analyses for validation and improvements of multi-model hpc applications.," Ph.D. dissertation, Université de Bordeaux, 2015.

[23] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, 1973.

[24] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM computing surveys (CSUR)*, vol. 43, no. 4, pp. 1–44, 2011.

[25] J. Y.-T. Leung and J. Whitehead, "On the complexity of fixed-priority scheduling of periodic, real-time tasks," *Performance evaluation*, vol. 2, no. 4, pp. 237–250, 1982.

[26] T. P. Baker, "A comparison of global and partitioned edf schedulability tests for multiprocessors tr-051101," 2005.

[27] I. Consortium *et al.*, *Best practice guide to hybrid mpi+ openmp programming*, 2017.

[28] *Omp_display_affinity*, Last accessed 10 May 2023. [Online]. Available: `https://www.openmp.org/spec-html/5.0/openmpse61.html`.

[29] *Hydra_topo_debug*, Last accessed 10 May 2023. [Online]. Available: `https://github.com/pmodels/mpich/blob/main/doc/wiki/how_to/Using_the_Hydra_Process_Manager.md`.

A

Compute Resources

This appendix states the computing resources of a node in the cluster target environment. The target environment is homogenous, so the architecture is the same across all nodes.

A.1 lscpu

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                56
On-line CPU(s) list:   0-55
Thread(s) per core:    2
Core(s) per socket:    28
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 85
Model name:            Intel(R) Xeon(R) Gold 6238R CPU @ 2.20GHz
Stepping:              7
CPU MHz:               4000.000
CPU max MHz:           4000.0000
CPU min MHz:           1000.0000
BogoMIPS:              4400.00
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              1024K
L3 cache:              39424K
NUMA node0 CPU(s):     0-55
```

B

Evaluation Details

This appendix contains a more exact description of certain evaluation prerequisites and details used in the project.

B.1 CPU-pinning Strategies

These CPU-pinning strategies were used in order to determine the increases in worker execution times with respect to the use of simultaneous multithreading. Information is acquired from *OMP_DISPLAY_AFFINITY* [28], and *HYDRA_TOPO_DEBUG* [29].

B.1.1 1 node, 1 worker (MPI process)

```
----Using logical cores----
process 0 binding:
11110000000000000000000000001111000000000000000000000000 //Scheduler process
process 1 binding:
00001111000000000000000000000000111100000000000000000000 //Collector process
process 2 binding:
00000000111100000000000000000000000011110000000000000000 //Worker process

OMP: pid 450288 tid 450288 thread 0 bound to OS proc set {8-11,36-39}
OMP: pid 450288 tid 450315 thread 1 bound to OS proc set {8-11,36-39}
OMP: pid 450288 tid 450316 thread 2 bound to OS proc set {8-11,36-39}
OMP: pid 450288 tid 450317 thread 3 bound to OS proc set {8-11,36-39}
OMP: pid 450288 tid 450318 thread 4 bound to OS proc set {8-11,36-39}
OMP: pid 450288 tid 450319 thread 5 bound to OS proc set {8-11,36-39}
OMP: pid 450288 tid 450320 thread 6 bound to OS proc set {8-11,36-39}
OMP: pid 450288 tid 450321 thread 7 bound to OS proc set {8-11,36-39}

----Pinning to physical cores----
process 0 binding:
11000000000000000000000000000000000000000000000000000000 //Scheduler process
process 1 binding:
00100000000000000000000000000000000000000000000000000000 //Collector process
process 2 binding:
```

```
00011111111000000000000000000000000000000000000000000000 //Worker process
OMP: pid 450288 tid 450288 thread 0 bound to OS proc set {3-10}
OMP: pid 450288 tid 450315 thread 1 bound to OS proc set {3-10}
OMP: pid 450288 tid 450316 thread 2 bound to OS proc set {3-10}
OMP: pid 450288 tid 450317 thread 3 bound to OS proc set {3-10}
OMP: pid 450288 tid 450318 thread 4 bound to OS proc set {3-10}
OMP: pid 450288 tid 450319 thread 5 bound to OS proc set {3-10}
OMP: pid 450288 tid 450320 thread 6 bound to OS proc set {3-10}
OMP: pid 450288 tid 450321 thread 7 bound to OS proc set {3-10}

----mpirun default----
OMP: pid 450288 tid 450288 thread 0 bound to OS proc set {0-55}
OMP: pid 450288 tid 450315 thread 1 bound to OS proc set {0-55}
OMP: pid 450288 tid 450316 thread 2 bound to OS proc set {0-55}
OMP: pid 450288 tid 450317 thread 3 bound to OS proc set {0-55}
OMP: pid 450288 tid 450318 thread 4 bound to OS proc set {0-55}
OMP: pid 450288 tid 450319 thread 5 bound to OS proc set {0-55}
OMP: pid 450288 tid 450320 thread 6 bound to OS proc set {0-55}
OMP: pid 450288 tid 450321 thread 7 bound to OS proc set {0-55}
```

B.1.2 1 node, 2 workers (MPI processes)

```
----Using logical cores----
process 0 binding:
11110000000000000000000000011110000000000000000000000000 //Scheduler process
process 1 binding:
00001111000000000000000000000001111000000000000000000000 //Collector process
process 2 binding:
00000000111100000000000000000000000111100000000000000000 //Worker process
process 3 binding:
00000000000011110000000000000000000000001111000000000000 //Worker process

OMP: pid 450288 tid 450288 thread 0 bound to OS proc set {8-11,36-39}
OMP: pid 450288 tid 450315 thread 1 bound to OS proc set {8-11,36-39}
OMP: pid 450288 tid 450316 thread 2 bound to OS proc set {8-11,36-39}
OMP: pid 450288 tid 450317 thread 3 bound to OS proc set {8-11,36-39}
OMP: pid 450288 tid 450318 thread 4 bound to OS proc set {8-11,36-39}
OMP: pid 450288 tid 450319 thread 5 bound to OS proc set {8-11,36-39}
OMP: pid 450288 tid 450320 thread 6 bound to OS proc set {8-11,36-39}
OMP: pid 450288 tid 450321 thread 7 bound to OS proc set {8-11,36-39}

OMP: pid 450289 tid 450289 thread 0 bound to OS proc set {12-15,40-43}
OMP: pid 450289 tid 450322 thread 1 bound to OS proc set {12-15,40-43}
OMP: pid 450289 tid 450323 thread 2 bound to OS proc set {12-15,40-43}
OMP: pid 450289 tid 450324 thread 3 bound to OS proc set {12-15,40-43}
OMP: pid 450289 tid 450325 thread 4 bound to OS proc set {12-15,40-43}
```

```
OMP: pid 450289 tid 450326 thread 5 bound to OS proc set {12-15,40-43}
OMP: pid 450289 tid 450327 thread 6 bound to OS proc set {12-15,40-43}
OMP: pid 450289 tid 450328 thread 7 bound to OS proc set {12-15,40-43}

----Pinning to physical cores----
process 0 binding:
11000000000000000000000000000000000000000000000000000000 //Scheduler process
process 1 binding:
00100000000000000000000000000000000000000000000000000000 //Collector process
process 2 binding:
00011111111000000000000000000000000000000000000000000000 //Worker process
process 3 binding:
00000000000111111110000000000000000000000000000000000000 //Worker process

OMP: pid 450288 tid 450288 thread 0 bound to OS proc set {3-10}
OMP: pid 450288 tid 450315 thread 1 bound to OS proc set {3-10}
OMP: pid 450288 tid 450316 thread 2 bound to OS proc set {3-10}
OMP: pid 450288 tid 450317 thread 3 bound to OS proc set {3-10}
OMP: pid 450288 tid 450318 thread 4 bound to OS proc set {3-10}
OMP: pid 450288 tid 450319 thread 5 bound to OS proc set {3-10}
OMP: pid 450288 tid 450320 thread 6 bound to OS proc set {3-10}
OMP: pid 450288 tid 450321 thread 7 bound to OS proc set {3-10}

OMP: pid 450289 tid 450289 thread 0 bound to OS proc set {11-18}
OMP: pid 450289 tid 450322 thread 1 bound to OS proc set {11-18}
OMP: pid 450289 tid 450323 thread 2 bound to OS proc set {11-18}
OMP: pid 450289 tid 450324 thread 3 bound to OS proc set {11-18}
OMP: pid 450289 tid 450325 thread 4 bound to OS proc set {11-18}
OMP: pid 450289 tid 450326 thread 5 bound to OS proc set {11-18}
OMP: pid 450289 tid 450327 thread 6 bound to OS proc set {11-18}
OMP: pid 450289 tid 450328 thread 7 bound to OS proc set {11-18}

----mpirun default----
OMP: pid 450288 tid 450288 thread 0 bound to OS proc set {0-55}
OMP: pid 450288 tid 450315 thread 1 bound to OS proc set {0-55}
OMP: pid 450288 tid 450316 thread 2 bound to OS proc set {0-55}
OMP: pid 450288 tid 450317 thread 3 bound to OS proc set {0-55}
OMP: pid 450288 tid 450318 thread 4 bound to OS proc set {0-55}
OMP: pid 450288 tid 450319 thread 5 bound to OS proc set {0-55}
OMP: pid 450288 tid 450320 thread 6 bound to OS proc set {0-55}
OMP: pid 450288 tid 450321 thread 7 bound to OS proc set {0-55}

OMP: pid 450289 tid 450289 thread 0 bound to OS proc set {0-55}
OMP: pid 450289 tid 450322 thread 1 bound to OS proc set {0-55}
OMP: pid 450289 tid 450323 thread 2 bound to OS proc set {0-55}
OMP: pid 450289 tid 450324 thread 3 bound to OS proc set {0-55}
```

```
OMP: pid 450289 tid 450325 thread 4 bound to OS proc set {0-55}
OMP: pid 450289 tid 450326 thread 5 bound to OS proc set {0-55}
OMP: pid 450289 tid 450327 thread 6 bound to OS proc set {0-55}
OMP: pid 450289 tid 450328 thread 7 bound to OS proc set {0-55}
```