CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

# ILAN: The Interference- and Locality-Aware NUMA Scheduler

## Incorporating interference and data locality awareness into the LLVM OpenMP runtime for the execution of taskloop constructs

Master's thesis in Computer science and engineering

Axel Carlsson
Edvin Mellberg

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

Master's thesis 2025

# ILAN: The Interference- and Locality-Aware NUMA Scheduler

Incorporating interference and data locality awareness into the LLVM OpenMP runtime for the execution of taskloop constructs

Axel Carlsson
Edvin Mellberg

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

**ILAN**: The **I**nterference- and **L**ocality-**A**ware **N**UMA Scheduler.
Incorporating interference and data locality awareness into the LLVM OpenMP runtime for the execution of taskloop constructs.
Axel Carlsson
Edvin Mellberg



Supervisor: Jing Chen, Department of Computer Science and Engineering
Examiner: Miquel Pericas, Department of Computer Science and Engineering

Master's Thesis 2025
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 2025

**ILAN**: The **I**nterference- and **L**ocality-**A**ware **N**UMA Scheduler.
Incorporating interference and data locality awareness into the LLVM OpenMP runtime for the execution of taskloop constructs

Axel Carlsson
Edvin Mellberg
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

# Abstract

Non-Uniform Memory Access (NUMA) systems are increasingly common as the go-to processor architecture for parallel computing within the field of High-Performance Computing (HPC). Similarly, OpenMP is the de-facto standard runtime for enabling parallelism. However, the default OpenMP runtime does not account for interference or data locality aspects, leading to performance degradations on NUMA systems where these effects become magnified. To address these challenges, this thesis proposes *ILAN*, an interference- and data locality-aware NUMA scheduler integrated into the LLVM OpenMP runtime, specifically targeting the `taskloop` construct. *ILAN* utilizes hardware topology information to enable a more structured task distribution strategy compared to the default OpenMP tasking scheduler, the work stealing scheduler, yielding improved data locality. Furthermore, the *ILAN* scheduler utilizes moldability to incorporate interference awareness, dynamically reducing the number of OpenMP threads to mitigate the effects of interference while further improving data locality. Performance evaluation using the NAS Parallel Benchmarks, Matrix Multiplication, and LULESH on a multi-socket NUMA platform demonstrates an average speedup of 10%, with a maximum speedup of 46%, compared to the default OpenMP work stealing scheduler.

Keywords: HPC, parallel computing, Scheduling, OpenMP, NUMA, interference, data locality.

# Acknowledgements

First and foremost, we would like to thank our supervisor Jing Chen for the support and guidance throughout the project. Our insightful discussions have been instrumental in achieving many of the project's successes.

Furthermore, we would like to thank our examiner Miquel Pericas for all the help and valuable knowledge he has provided during the work.

We also would like to express a big thanks to our respective girlfriends for taking on additional household chores while we spent late nights working to finish this thesis project.

Finally, we thank the hard-working people at C3SE for providing us with a cluster and a platform for evaluating our product.

Axel Carlsson & Edvin Mellberg, Gothenburg, June 2025

# Contents

# List of Figures

# List of Tables

# Listings

# 1

# Introduction

In high-performance computing (HPC), parallelism is leveraged to achieve high throughput on multi-core platforms. Task-based programming models have emerged as a powerful approach for expressing parallelism. These models enable different scheduling approaches, such as dynamic scheduling and load balancing across the system. Therefore, efficient task scheduling plays a vital role in maximizing performance by ensuring that resources are optimally utilized. A key challenge in parallel computing is reducing interference between tasks, which often arises from resource contention on shared resources, such as memory buses or inter-task communication. Most consumer systems are modeled as SMP (Symmetric Multiprocessing) architectures, where all processors share one memory bus. This type of system provides uniform memory access times for all processors. However, SMP architectures typically do not scale well above 8-12 processors as the memory bus may suffer heavy contention [1]. Therefore, HPC systems often leverage Non-Uniform Memory Access (NUMA) architectures to improve performance and scalability. In NUMA architectures, certain processor cores are physically closer to certain memory units, resulting in non-uniform memory access times. This leads to fewer processors competing for the same memory bus, reducing bus congestion and allowing memory access through high-speed interconnects [2].

NUMA architectures do, however, introduce additional complexity, as applications frequently accessing remote memory will suffer from long memory access times and consequently poor performance. Data locality refers to the principle of keeping frequently accessed data close to the processing unit that requires it, thereby reducing the need for remote memory accesses and minimizing access latency. To fully reap the benefits of the NUMA architecture, ensuring good data locality becomes important to maximize the performance of the system [3]. Poor data locality leads to longer memory access times and could also lead to memory resource congestion. However, optimizing too much for data locality can cause memory contention problems and result in uneven task distribution among processors, yielding degraded performance [4].

One of the major APIs and runtime libraries used for parallel programming is OpenMP [5]. The parallelism in OpenMP comes from the ability to split loops into multiple parallel tasks. This can be done by either using work sharing, such

as the `omp for` construct, or through tasking, such as the `omp taskloop` construct. In the former, the loop iterations are directly assigned to worker threads while the latter creates tasks containing the iterations. Distributing the workload over multiple processors can significantly improve the program performance. However, if too many processors are employed to execute the same loop, the interference between processors might become severe due to resource congestion or data dependencies. In some cases, the interference can even result in performance degradation, in which case fewer processors would yield faster execution [6].

While the current OpenMP runtime provides efficient tools for shared-memory parallelism and many options for task scheduling, it currently lacks mechanisms for dynamically detecting interference between processors executing a `taskloop` construct. Furthermore, it is not aware of the data locality within a program and, by default, ignores hardware topology information. This leaves significant room for performance improvements, particularly on NUMA platforms. An OpenMP scheduler that incorporates interference and data locality awareness could further increase the performance of OpenMP applications. This project aims to develop such a scheduler for the `taskloop` construct in the LLVM OpenMP runtime library [7].

## 1.1 Problem Description

The performance of parallel taskloops depends on several factors. One of the main factors is interference within the taskloop, which can arise due to data dependencies between tasks or as a result of resource contention. On NUMA architectures, the effect of data dependencies between tasks depends on the data locality of the program, as this dictates memory access times. In this case, poor data locality occurs when tasks operating on the same data execute on processors that do not share the same memory unit, resulting in performance degradation. The resource contention can also be impacted by the degree of data locality, but perhaps even more so by the number of processors employed to execute the taskloop. The performance will eventually degrade if too many processors try to operate on the same data.

Currently, the OpenMP runtime lacks mechanisms to dynamically account for interference and data locality when making scheduling decisions. Addressing this deficiency in the OpenMP runtime scheduler could significantly improve performance on NUMA platforms, especially in the parallel execution of taskloops, by reducing interference and improving data locality.

## 1.2 Related Work

Various previous studies have explored task scheduling in parallel computing with a focus on interference- or data locality-aware strategies. Some of this research will be presented in this section.

Interference-aware scheduling techniques have previously been investigated, such as the online scheduler proposed by Chen et al. [6]. This scheduler analyzes the execution of tasks to find performance asymmetries that indicate interference between tasks. This information is used to employ "moldability", a strategy that reduces the number of cores used to execute parallel tasks with the goal of minimizing interference. An earlier research paper proposes a similar approach for multiprocessor systems, called Feedback-driven threading [8]. This approach uses a profiling stage to analyze whether an application might be limited by data synchronization or memory bandwidth to select the number of threads used for execution. Since this research dates back to 2008, the reported results and conclusions may not directly apply to today's hardware platforms. However, the underlying ideas and findings remain relevant, perhaps even more so, given the growing presence of NUMA architectures within HPC.

As for data locality-aware scheduling, most previous proposed techniques either focus on limiting remote task stealing to maintain good data locality, or employee data re-binding through memory re-allocation or utilizing the first-touch policy. The Opera scheduler incorporates offline analysis of memory access patterns to find tasks that mostly access the same cache blocks [9]. A task-to-data affinity table is created which allows the online scheduler to schedule tasks with high memory access similarities on the same cores. Similarly, hierarchical schedulers have been proposed, where task stealing within sockets is prioritized to improve data locality while minimizing costly remote memory accesses [10]. A different approach to improve data locality on NUMA platforms is to re-bind data among NUMA nodes based on data-footprint hints given by the programmer [11].

To the best of our knowledge, there is no previous work on schedulers combining interference and data locality awareness.

## 1.3 Thesis Contributions

This thesis contributes to the field of parallel computing through the development of a novel OpenMP scheduler incorporating both interference and data locality awareness. The scheduler specifically targets symmetrical NUMA platforms and operates on the OpenMP `taskloop` construct.

The primary contributions of this thesis work are the following:

- A framework that detects and adapts to interference and data locality using moldability to guide scheduling decisions.
- A task distribution strategy tailored for NUMA platforms yielding improved data locality and reduced intra-taskloop interference.
- A performance evaluation demonstrating improved scheduling behavior through the integration of the two above methods focusing on execution time as the main performance metric while also reflecting on scheduling overhead and performance variability.

  The resulting scheduler is shown to achieve an average speedup of 9.7% and a maximum speedup of 45.8% compared to the default OpenMP work stealing scheduler.

## 1.4 Report Outline

The remainder of this thesis report is organized as follows: Chapter 2 provides the necessary theoretical knowledge to follow the concepts discussed in the thesis. Chapter 3 breaks down the methodology and tools used throughout the thesis work. In Chapter 4, the design of the proposed scheduler is presented in detail. Thereafter, Chapter 5 specifies the experimental setup and benchmarks used for performance evaluation. The results from the performance evaluation are presented in Chapter 6, and further discussed in Chapter 7. Finally, concluding remarks from the projects as a whole are summarized in Chapter 8.

# 2

# Background

This chapter provides the background knowledge needed to follow the rest of the report. This chapter presents the following topics: Scheduling Theory within HPC, the LLVM OpenMP runtime library, NUMA architectures, and performance monitoring.

## 2.1 Scheduling Theory

Within the field of HPC, scheduling is a critical factor for achieving optimal performance in computing clusters, by efficiently utilizing the platform resources. As HPC systems have evolved from early supercomputers to modern multi-core architectures, scheduling strategies have adapted to the increasing complexity of hardware configurations and workload characteristics.

Scheduling strategies can generally be classified as static or dynamic. Static scheduling assigns tasks to processors at compile time and relies on offline analysis to determine optimal task placement before execution. This approach is efficient when workload characteristics are well-known and predictable, minimizing runtime scheduling overhead. However, static scheduling lacks adaptability to unexpected workload variations and system state changes. Dynamic scheduling, on the other hand, determines task assignments at runtime, allowing for adjustments based on real-time conditions, such as processor availability, memory congestion, and task execution times. Dynamic scheduling can incorporate both offline and online analysis. Offline analysis is used to derive heuristics and performance models, while online analysis continuously monitors execution and adapts scheduling decisions accordingly. This adaptability makes dynamic scheduling more suitable for modern HPC systems, where workloads are often irregular and unpredictable.

Several dynamic scheduling techniques are commonly used in HPC environments. Work stealing is a widely adopted method in parallel programming models, such as OpenMP, where idle processors dynamically steal tasks from overloaded ones, thereby balancing the computational load. This technique is particularly effective in irregular workloads where task execution times vary unpredictably [12]. Another approach, guided self-scheduling, divides the workload into chunks of decreasing

size, initially assigning large tasks to minimize scheduling overhead and gradually shifting towards smaller tasks to maintain load balance towards the end of execution [13]. Hierarchical scheduling, often employed in large-scale HPC clusters, organizes scheduling decisions in a multi-level structure where high-level policies allocate workloads to different processing groups, while lower-level schedulers handle finer task distribution within each group. Energy-aware scheduling is another emerging area, aiming to optimize performance while minimizing power consumption through techniques such as intelligent task distribution and Dynamic Voltage Frequency Scaling (DVFS) [14]. The energy aspect becomes more and more important as the demand for HPC systems grows every year. Generally, several scheduling techniques are combined to form the task scheduler, where both online and offline analysis might be included.

### 2.1.1 Interference

In HPC systems, interference can occur within applications, called intra-application interference, or between different applications, called inter-application interference. This project will mainly focus on intra-application interference.

Intra-application interference occurs when threads collaboratively execute a parallel application and compete for resources, typically by accessing the same memory regions. Some potential sources of intra-application interference in HPC systems are memory congestion and memory contention. Memory congestion occurs when the memory resources in the system are overloaded, such as memory controllers or memory buses. This mostly has to do with the number of processors requesting memory data at the same time rather than what specific data is requested. Memory congestion also includes the case when interference occurs due to two or more processors competing for the same cache memory unit, evicting each other's data from the cache [15]. Memory data contention, on the other hand, refers to the case when two processors compete for data in the same memory region, resulting in cache invalidations of the cache line. When the processors compete for the same data, this is called true sharing. The other case is false sharing, which occurs when multiple threads access independent data stored in the same cache line, causing invalidations despite no true dependency. In the context of parallel applications, memory data contention is also referred to as task data dependencies. Both true and false sharing degrade the program performance, however, none of these issues are caused by limited resources in the system overall. In theory, the interference caused by memory data contention can be reduced by taking the task data dependencies into account.

The types of interferences considered in this project are listed below:

- Intra-taskloop interference due to memory congestion. Performance is reduced due to memory bandwidth or resource oversubscription.
- Intra-taskloop interference due to memory data contention. Tasks may interfere with each other as a result of true or false sharing.

### 2.1.2 Data Locality

Data locality refers to the tendency of processors to access the same set of memory locations over a short period of time. Usually, two versions of data locality are considered: temporal locality and spatial locality [16]. Temporal locality refers to the tendency that memory accessed is usually reused within a short time frame. Spatial locality refers to the tendency of the next data being read from memory to be closely located to the last data that was read.

However, within HPC and specifically NUMA platforms, data locality is a term used in a slightly different manner. In this context, data locality refers to the proximity of data relative to a processor, both spatial and temporal. The spatial data locality tells how far away from the processor the memory unit is that stores the required data, which translates into memory access time that gets longer the farther away the memory unit is. Temporal data locality, on the other hand, explains how recently the required data was used by the processor. For example, suppose two tasks operating on the same data run consecutively on the same processor. In that case, this results in good temporal data locality since the second task can reuse the data already loaded in the cache memory of the processor. Both types of data locality affect the degree of memory congestion and contention in the system, therefore significantly impacting interference. Thus, by tracing and optimizing for good spatial and temporal data locality, interference in HPC systems can be reduced [4].

The types of data locality considered in this project are listed below:

- Intra-taskloop data locality. Placing tasks with similar data access patterns closely together might reduce memory access latencies and improve performance. This type of data locality strongly impacts the degree of interference between tasks with data dependencies.
- Inter-taskloop data locality. Performance is impacted by the reuse of data from one taskloop to another. One taskloop's data access patterns might impact the performance of the subsequent taskloops, especially if the same data is being reused by many taskloops throughout the program.

## 2.2 LLVM OpenMP Runtime Library

OpenMP is a widely used API for shared-memory parallel programming, providing directives for work sharing, synchronization, and task-based parallelism [5]. It enables efficient multi-core execution in C, C++, and Fortran applications. The LLVM project is an open-source compiler infrastructure used in various compilers. The LLVM OpenMP runtime library is hence the open-source implementation of OpenMP within the LLVM ecosystem. The runtime library manages parallel execution, thread coordination, and resource allocation to optimize the performance of parallel applications across different architectures.

There are two main constructs to enable parallelism within the OpenMP runtime: the work sharing constructs, such as `omp for`, and the tasking constructs, such as `omp taskloop`. Work sharing constructs like `omp for` distribute loop iterations across available threads based on predefined scheduling strategies, including static, dynamic, or guided scheduling. The static work sharing scheduler divides loop iterations evenly among threads at compile-time, minimizing runtime overhead and enhancing data locality for contiguous data access, but potentially leading to load imbalance if iteration workloads vary, causing some threads to become idle. Dynamic and guided scheduling assign loop iterations to threads at runtime, allowing improved load balancing for irregular workloads, though at the cost of increased scheduling overhead and potentially reduced data locality due to thread synchronization and iteration reassignment.

In contrast, tasking constructs like `taskloop` dynamically create tasks consisting of loop iterations, which are then scheduled by the OpenMP runtime. The default scheduling technique used is work stealing, where all newly created tasks are distributed to one or more task queues, and the threads steal tasks from each other to achieve load balancing. Notice that only a few threads will be given all of the tasks, and the other threads need to rely on work stealing to fetch tasks from other threads for execution. This approach is particularly effective for handling irregular or nested workloads, as the work stealing ensures improved load balancing compared to the work sharing schedulers. However, this increased flexibility introduces additional overhead related to task creation, management, and scheduling, which can negatively impact performance if tasks are too fine-grained. Additionally, dynamic reassignment of tasks may reduce data locality, potentially causing more cache misses. Therefore, tasking constructs are generally favored for irregular, dynamic, or nested parallel workloads, while work sharing constructs are preferred when iteration workloads are uniform and predictable [17].

Aside from the constructs enabling parallel execution in the OpenMP runtime, there are also mechanisms for configuring processor affinity. This allows threads to be bound to specific processor cores, thus influencing cache efficiency and memory latency on some processor architectures. Available affinity strategies include compact, scatter, and explicit binding, each influencing thread distribution uniquely. Compact affinity groups place threads on adjacent cores, maximizing data locality,

whereas scatter affinity spreads threads across cores to minimize resource contention. Explicit binding grants the programmer detailed manual control over thread-core placement. While properly configured processor affinity can significantly enhance performance, the default setting in OpenMP is to use no affinity. Therefore, the potential performance gain is only obtained if the programmer appropriately configures the affinity settings.

### 2.2.1 The `taskloop` Construct

The OpenMP `taskloop` construct is a common tasking construct used to divide loop iterations between several tasks. The construct creates the tasks required to execute the loop concurrently, with an even distribution of iterations among the tasks. Below, the syntax for the `taskloop` construct is shown:

**#pragma omp taskloop** *[clause[, ] clause] ... new-line*
*for-loops*

Two important clauses available for this construct are *grainsize*, meaning the number of iterations assigned to each task, and the *num_tasks*, specifying the number of tasks to be created. These clauses enable the programmer to help the runtime create a suitable number of tasks depending on the work to be done inside the loop. When none of these clauses are specified, the number of tasks created is always *threads*10*, where *threads* are the number of OpenMP threads assigned to the application. The only exception to this is if the total number of iterations is less than *threads*10*, in which case the number of tasks created will be equal to the number of iterations.

An example usage of the `taskloop` construct within OpenMP is shown in Listing 2.1. In this example, 20 tasks will be created as specified by the *num_tasks* construct. If we remove that construct, 40 tasks would be created, as the number of OpenMP threads is set to 4 by calling the function *omp_set_num_threads()*. The `parallel` construct is used to initiate a parallel region and create the OpenMP threads. The `single` construct ensures that only one of the threads executes the `taskloop` construct, which leads to the for loop being executed exactly once as intended. All created threads still participate in executing the loop.

**Listing 2.1:** Example usage of the `taskloop` construct.

```
void main(){
    omp_set_num_threads(4);

    #pragma omp parallel
    {

        #pragma omp single
        {

            #pragma omp taskloop num_tasks(20)
            for (int i=0;i<N;i++){
                // for-loop iterations...
            }
        }
    }
}
```

When using tasking constructs in OpenMP, each thread has its own queue of tasks. The main thread that executes the `taskloop` construct will create all tasks and place them in one or more task queues. Then, it is up to the dynamic work stealing scheduler to distribute the tasks among all threads. The task execution works in the following way: Whenever a thread is idle, it will look in its queue and select a task to execute. If the thread is idle and its queue is empty, it will try to steal a task from another thread's queue. This is the most common case, as all tasks will be placed in only a few of the threads' queues. In the current LLVM implementation of OpenMP, the task-stealing among threads is random, meaning that an idle thread will randomly select another thread, here called the victim, and try to steal a task from it. If the victim thread has at least one task in its queue, this task will be successfully stolen. Only one task is stolen at a time, even if more tasks are available in the victim thread's queue. After the successful steal, the stealing thread will keep trying to steal from that same victim thread until it no longer is successful, due to the queue being empty, in which case the thread will go back to randomly trying to steal from other threads. This is how dynamic load-balancing is achieved in the case of taskloops using the default OpenMP work stealing scheduler.

It is easy to see how this way of scheduling tasks might lead to suboptimal data locality properties, as the threads are stealing tasks at random. If contiguous iterations of the loop work on the same or closely located data, this locality will not be taken advantage of if contiguous tasks are randomly spread out across threads. However, improving the random work stealing scheduler in a way that benefits all types of platforms and applications is not trivial. Nevertheless, it generally delivers adequate performance in most scenarios.

## 2.3 NUMA Architecture

The NUMA architecture is a memory design for multiprocessor systems in which memory latencies vary based on the physical location of the memory relative to the processor [1]. NUMA architectures aim to solve the scalability issue of SMP systems. In SMP architectures, all processors share the same memory and memory bus [18]. While this design is suitable for desktop computers, it struggles to scale efficiently in systems with many CPU cores [1]. Many processors accessing the same memory controller leads to memory congestion and increased latency.

To resolve this issue, NUMA architectures are divided into NUMA nodes. Each NUMA node consists of processor cores and associated physical memory (*local memory*), grouped both logically and physically. The access time to local memory is fast, while access to the memory of other NUMA nodes, called *remote memory*, depends on the distance to the other NUMA node. Since memory is distributed on NUMA nodes, congestion is mitigated as only a subset of all processors share local memory, and all remote memory is instead accessed through high-speed interconnects. However, this improvement in scalability comes at a cost in complexity for the scheduler and programmer. Since memory access time to remote memory is highly dependent on the physical distance between the requesting node and the remote memory, scheduling decisions greatly impact the overall performance of a parallel application. Thus, to produce an efficient schedule for parallel applications on NUMA platforms, the scheduler needs to be aware of the hardware topology, as well as dynamic factors during runtime. Additionally, algorithmic decisions by the programmer significantly impact performance on NUMA platforms, as frequently accessing remote memory will quickly degrade performance.

## 2.4 Online Performance Monitoring

Online scheduling requires performance monitoring to make informed scheduling decisions. A common approach is measuring execution time. Another approach is to collect hardware performance counters, which are usually transformed into higher level performance metrics.

### 2.4.1 Hardware Performance Counters

Most modern processors provide a set of hardware performance counters that store the count of different hardware events. These events typically include cache accesses, cache misses, instruction counts, clock cycles, and stall cycles, to name a few. They are stored in special-purpose registers, and thus, accessing them comes with very low overhead. The most common tool to access these counters is called *perf*, which in 2009 was integrated into the Linux kernel. The *perf* utility has a simple interface with the kernel, consisting of a single system call that provides a file descriptor. The file descriptor can then be used to enable and disable the counter, as well as resetting and reading the current value of the hardware performance counter. *Perf* provides an API to efficiently open generic counters that most processors support, such as

clock cycles, but it also provides the possibility to open specific counters only supported by specific processors. Specific counters, also called *raw* counters, are usually documented in programmer references provided by the processor manufacturer.

Hardware performance counters can be used by advanced users and programmers for performance analysis, i.e. identifying performance bottlenecks and gaining insight into a program's execution. One performance counter by itself is usually not very useful as the absolute value can depend on many different factors. Instead, performance counters are usually combined and analyzed in relation to other counters. The combinations of performance counters are called performance metrics, which are explained in more detail in Section 2.4.2. Some academics refer to performance counters and performance metrics interchangeably, but this paper refers to a performance metric as a combination of multiple performance counters. This is sometimes also called derived metrics.

### 2.4.2 Performance Metrics

Most existing performance models rely on performance metrics to make predictions, such as *instructions per cycle* (IPC), *cache miss ratio*, and *stall cycles in the backend*. A high IPC indicates efficient usage of the processor, while a low IPC could mean long stalls due to memory latency or pipeline flushes due to e.g. branch mispredictions. High cache miss ratio could indicate poor data locality, leading to long memory accesses. However, this does not necessarily need to be the case as very compute intensive programs could exhibit a high cache miss ratio, while still being efficient because of very infrequent memory accesses. Stall cycles in the backend are cycles wasted by the processor because it is waiting for resources, usually memory. The number of stall cycles compared to the total amount of cycles can be a metric for CPU utilization, as an example of how counters can be combined into metrics to provide deeper insight.

Combining many metrics could help provide a more holistic view of the performance of a program, and thus indicate what potential measures could be taken to improve the performance. For example, excessive memory stalls could be mitigated by reducing pointer indirections and improving memory layout of the program, or by implementing prefetching and adding memory awareness to the scheduler. The latter could potentially help a smart scheduling algorithm improve the performance of programs without the intervention of the programmer.

# 3

# Method

This chapter introduces the methodology and tools used during the development of this thesis work. The tools used in the project are introduced in Section 3.1. The different scheduler features developed leading up to the proposed scheduler are presented in Section 3.2. Finally, the performance counters and metrics used for analysis during the work are presented in Section 3.3.

## 3.1 Tools

The implementation and evaluation of the proposed scheduler relied on low-level system tools to help understand and give insight into runtime behavior. Specifically, Linux *perf*, *hwloc*, and *numactl* were used to validate scheduler behavior and measure performance characteristics.

- **Linux perf** is an advanced profiling utility within the Linux kernel, specifically designed to provide a generalized interface of low-level performance data from hardware performance counters available on processors or software counters available in the Linux kernel [19]. In the scope of this thesis, the *perf* API was used to integrate performance monitoring directly into the OpenMP runtime. As mentioned in Section 2.4.1, some performance counters are general and available on most platforms, while other *raw* counters are hardware-specific and only available on certain processors. Using *perf* within the OpenMP runtime enabled deeper and more detailed analysis of low-level details of the OpenMP runtime, which has been essential to understanding the runtime behavior as well as for performance analysis.
- **Hwloc** stands for Portable Hardware Locality [20]. This tool provides hardware topology information, such as the hierarchical design of memory units or processor cores and how these components are interlinked. On platforms where *hwloc* is available, the *hwloc* API is already integrated into the LLVM OpenMP runtime, and possibly used to determine thread affinity. In this project, the information gained from *hwloc* was essential for many online scheduling decisions, as the tool provides detailed information about the structuring of NUMA nodes and the memory latencies between them.

- **Numactl** is a Linux command-line tool for NUMA data placement [21]. It can report topology information, generate a memory access latency matrix, and has the functionality to bind specific data or memory pages to specific NUMA nodes. *Numactl* was primarily used during benchmarking and analysis to gain insight into static properties of the platform and to investigate the effects of remote memory accesses by manually pinning processes and memory data to NUMA nodes.

## 3.2 Scheduler Development

The first feature implemented in the runtime was NUMA awareness for two primary reasons. Firstly, the non-uniform nature of NUMA platforms adds complexity to scheduling decisions. Disregarding the underlying platform topology could significantly degrade the performance of an application. Secondly, one goal of the scheduler was to collect performance counters to be used as a basis for guiding the scheduling decisions. For consistency, this requires a connection between OpenMP threads and physical cores, i.e. thread-to-core pinning, due to the non-uniformity of the platform. Therefore, the first feature set a strict thread-to-core affinity for all logical threads and used hardware topology information to make scheduling decisions. With the aforementioned knowledge, the scheduler could make informed decisions in the task scheduling to distribute likely dependent tasks on the same NUMA node, thus reducing the effects of intra-taskloop interference. One key insight gained from this feature was the importance of flexibility in the scheduler. Although optimizing for data locality is often beneficial, it is seldom worth pursuing data locality if doing so significantly reduces the load balance.

The second developed feature was interference awareness through moldability. Intra-taskloop interference is interference between tasks or threads that execute the same taskloop, as explained in Section 2.1.1. Typically, intra-taskloop interference can be mitigated in a couple of ways. One way is improving the application code to reduce the dependencies between loop iterations or reducing the number of memory accesses, both of which are not in the scope of this work, nor desirable as in our case the application works as intended. Alternatively, intra-taskloop interference can be mitigated through improved scheduling, by avoiding the simultaneous execution of dependent tasks. However, this is difficult to accomplish as such fine-grained control is challenging without introducing too much overhead from the scheduler. Moldability is a simple yet effective way of reducing the amount of intra-taskloop interference, simply by reducing the number of interfering actors, an approach that has previously proven effective [6][22]. In this project, moldability was implemented and tailored for NUMA architectures.

The NUMA awareness implemented as the first feature showed promising improvements, although it also showed some drawbacks. The main shortcoming proved to be the lack of load balancing, as tasks had only been allowed to be stolen from within NUMA nodes. In benchmarks with a strong NUMA effect, the difference in data locality between nodes caused a large performance difference and load im-

balance. Therefore, the third feature implemented load balancing, allowing faster nodes to steal tasks that originally belonged to other nodes. Note that there is no difference in hardware capabilities between the nodes, the difference in performance stems mainly from distance to physical memory and the effects thereof.

The *Moldability Framework* showed promising performance, but had an important limitation: it was oblivious to the non-uniformity of NUMA platforms, as the nodes used for taskloop execution were selected randomly. As explained earlier, although the cores are identical with the same amount of cache and memory, they still exhibit different performance executing the same taskloop. This is due to the aforementioned NUMA effects. Therefore, the moldability was modified to no longer select nodes at random. Instead, performance statistics were collected during the execution to enable a more informed selection of nodes. For example, the fastest nodes for a particular taskloop could be identified and used for execution.

All of the above features were implemented, integrated, and refined into a cohesive scheduler featuring both interference and data locality awareness. A detailed description of the proposed scheduler, in its entirety, is presented in Chapter 4.

## 3.3 Performance Metrics

As described above, the *perf* tool was used for online performance monitoring through the use of performance counters. The performance counters were either analyzed directly or combined into performance metrics, a common practice described in Section 2.4.2. Analyzing performance metrics proved to be highly valuable in identifying bottlenecks and other issues in the scheduler. Primarily, data locality effects were analyzed from cache- and memory-related performance counters. Interference and overall performance aspects were analyzed mainly on the basis of IPC and the efficiency metric. Table 3.1 shows some of the most used performance counters and metrics throughout the project.

**Table 3.1:** Performance counters and metrics used throughout the project.

| Counter/Metric | What the metric indicates |
|---|---|
| `instructions` | Number of retired instructions. Used to calculate IPC. |
| `IPC` | Instructions per cycle. Used to determine which nodes execute more efficiently and identify effects of interference. |
| `Cache references` | All accesses to the last-level cache (LLC), indicating memory intensity. |
| `Cache misses` | LLC misses that reached DRAM. A high number of LLC misses could intuitively indicate poor data locality, but no such correlation has been found during this project. |
| `Efficiency` | Calculated as $IPC \times n$, where $n$ is the number of NUMA nodes used. This metric indicates how efficiently a given configuration executed a taskloop. |
| `L1-fills-all` | Data loaded into L1 cache, typically due to cache misses or prefetches. Used to compute the two L1-fill ratios below. |
| `Ratio of L1-fills from same NUMA node` | Hints at good data locality. Local memory accesses reduce stalls caused by remote-memory latency. |
| `Ratio of L1-fills from different NUMA node` | Opposite of the above, hints at poor data locality. |
| `Backend bound - Memory` | Fraction of cycles stalled due to memory operations. Used to show how memory-bound a taskloop is and to compare nodes by memory-access stalls. |

# 4

# Scheduler Design

The default OpenMP work stealing scheduler for tasking constructs distributes the tasks to a few OpenMP threads at random, and relies on work stealing to achieve load balancing between threads. This results in a simple, effective, and versatile scheduling with low overhead. However, random task distribution could lead to poor data locality within a taskloop, especially if the data is operated on contiguously. Furthermore, this scheduler uses all available computing cores (if not instructed otherwise by the programmer), a strategy that proves to be sub-optimal in scenarios with high levels of interference and data dependencies between tasks [6].

In this chapter, we propose the **I**nterference and **L**ocality-**A**ware **N**UMA scheduler, or shortly *ILAN*, a dynamic scheduler for NUMA platforms incorporating interference and data locality awareness through the *Moldability Framework*. The data locality is further improved by the *NUMA-aware Task Distribution* strategy, which is designed to reduce memory data contention and offers a more predictable task distribution over the random strategy. Work stealing is done within NUMA nodes to maintain data locality, however, if the data dependencies between tasks are low, load balancing across NUMA nodes might increase performance even if leading to worse data locality. In this case, full-processor work stealing is enabled.

## 4.1 Overview

The overview of the scheduler is presented in Figure 4.1. Steps 1 through 8 in the figure are explained briefly in chronological order in this section. The *Moldability Framework* controls the selection of the taskloop configuration for the next execution of a taskloop. The taskloop configuration contains the attributes {`num_threads, node_mask, steal_policy`}. The configuration is used by the *NUMA-aware Task Distribution* to determine task placement and task steal policy. To enable NUMA awareness, all OpenMP threads are pinned to specific cores, a process further explained in Section 4.4. The `num_threads` and `node_mask` determine how many threads should be part of the taskloop execution and which nodes should receive tasks. The `steal_policy` determines if tasks can only be stolen from threads of the same NUMA node, called NUMA-strict tasks, or from any thread. More detailed information about how the configuration is selected can be found in Section 4.2. Both

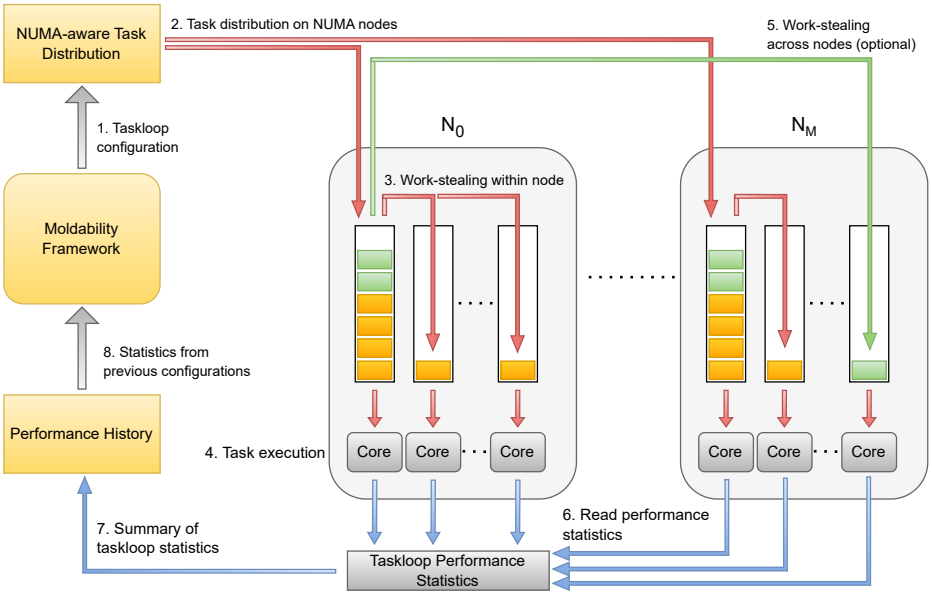



**Figure 4.1:** Overview of the *ILAN* scheduler. The number of NUMA nodes ranges from 0 to M, with the same number of threads as cores on each NUMA node. In the task queues, the orange tasks are NUMA-strict, while green tasks are available for stealing across the whole processor.

the task placement and the steal policy are assigned as attributes of the task itself, which decentralizes decision-making for the task distribution and task stealing. The tasks are created and assigned their attributes according to the taskloop configuration, and thereafter gets pushed to the respective node. All tasks are placed on the first core of the NUMA node. Work stealing is used to distribute tasks within the NUMA node. As soon as tasks are placed on the first thread of the NUMA node, other threads on the node will begin stealing tasks for execution. Same as for the default OpenMP work stealing scheduler, threads only steal one task at a time. After a thread has stolen a task, task execution begins, and statistics for the execution are collected on a thread basis. If the steal policy of the taskloop configuration allows work stealing across the whole processor, load balancing between nodes will begin when nodes start to run out of tasks. When all tasks have been completed, the performance statistics are summarized for the whole taskloop on a NUMA node basis. The performance statistics are stored in the performance history to be used the next time the taskloop is encountered, repeating the process explained in this section.

## 4.2 Moldability Framework

The *Moldability Framework* handles the main scheduling decisions and performance tracing in the proposed scheduler. As described in Section 2.1.1, intra-taskloop interference can arise for various reasons, for example, from memory congestion or memory data contention. A simple yet effective way of handling this interference is through moldability, as proposed in the previous research discussed in Section 1.2. In our proposed *Moldability Framework*, the number of threads and NUMA nodes are moldable to adapt for intra-taskloop interference and data locality. Furthermore, load balancing is optional and can be enabled or disabled by the framework.

The *Moldability Framework* samples performance statistics from taskloop executions and uses the performance history to determine the optimal taskloop configuration. The optimal configuration for each taskloop is found during an exploration stage. Once the optimal taskloop configuration is found, this configuration is used for the remainder of executions of the taskloop. The output of the *Moldability Framework* is the taskloop configuration itself, which tells the runtime how the taskloop should be executed. The taskloop configuration consists of three attributes:

- `num_threads` : *int {0..C}* - the number of threads allocated for the execution, where $C$ is the total CPU cores available to the OpenMP runtime.
- `node_mask` : *bitmask 16 bits* - the number of NUMA nodes, as well as the IDs of the NUMA nodes, to allocate for execution.
- `steal_policy` : *Enum {strict, full}* - disables/enables the last portion of tasks to be stolen across NUMA nodes for full-processor load balancing.

Currently, `num_threads` is selected with NUMA node granularity. This means that all the cores on the nodes selected by the `node_mask` will be active. In other words, if a certain NUMA node is selected for execution, all cores on that node will be used for execution. Recall that OpenMP threads are pinned to specific cores, which means that the number of active cores and active threads will always be equal. The reason for always enabling all cores on the selected NUMA nodes is to keep scheduling decisions on a NUMA node level. This simplifies the online analysis by prioritizing the mitigation of inter-node interference, although this can cause intra-node data congestion on caches. This granularity also speeds up the exploration stage by making sure it is not too fine-grained. However, we recognize that this granularity might not be suitable for all platforms, depending on the size of NUMA nodes in the system. It should also be noted that the 16-bit *node_mask* limits the current implementation to a maximum number of 16 NUMA nodes.

After each taskloop execution, the taskloop performance statistics are stored for that specific configuration and taskloop. Since threads wait at an implicit barrier when finishing their tasks, the slowest core determines the execution time of its NUMA node; consequently, the slowest NUMA node determines the execution time of the whole taskloop. Many different statistics have been used throughout the project for

taskloop analysis, but the execution time is the only statistic used to make online scheduling decisions. How reported execution times are used to find the optimal taskloop configuration is explained in the following section.

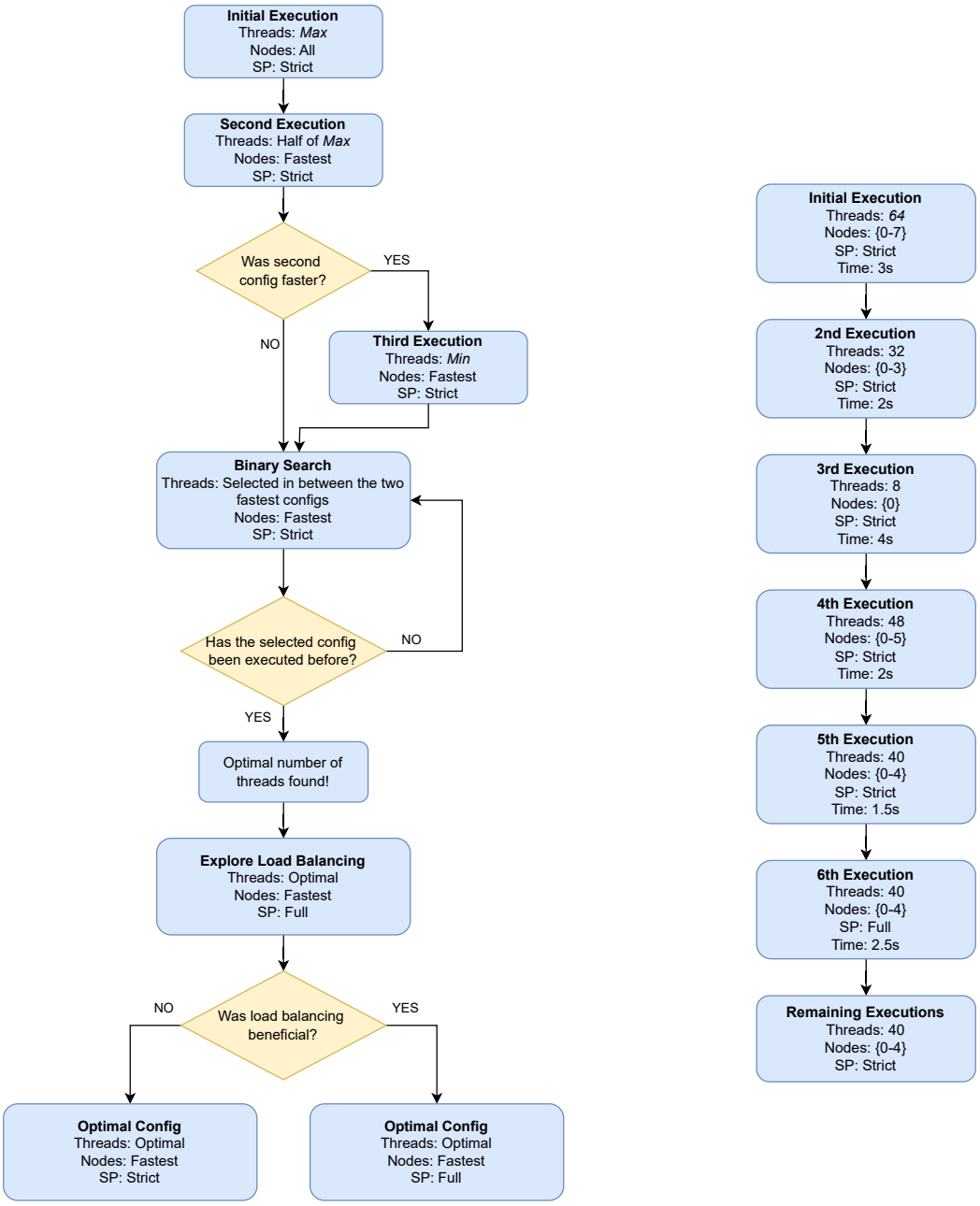
### 4.2.1 Exploration Stage

The goal of the exploration stage is to find the optimal taskloop configuration for each individual taskloop. This is done through an exploratory approach, where several configurations are executed and compared to find the optimal one. The exploratory approach is greedy, meaning that a local optimum is accepted as the optimal configuration, rather than continuing for an exhaustive search.

The taskloop configuration selection of a specific taskloop during the exploration stage is illustrated in Figure 4.2a. Initially, the performance history will be empty. For the initial execution, the largest configuration will be selected, i.e. the maximum number of threads and nodes available. NUMA-strict stealing policy is chosen by default during exploration to favor data locality over load balancing. In the second iteration of the taskloop, half of the available threads are used. In the case where this gives a performance increase, the next iteration of the taskloop will be assigned the smallest possible configuration, i.e. one NUMA node. A specific sample execution of the exploration stage is shown in Figure 4.2b.

After the initial executions, the exploratory approach begins, where a form of binary search is employed to find the best taskloop configuration, i.e. the local optimum. The binary search algorithm retrieves the configurations used for the fastest and second-fastest taskloop executions from the execution history. The algorithm will select the configuration in the middle between these two configurations. For example, if the fastest configuration ran on 32 threads and the second fastest ran on 64 threads, 48 threads will be selected for the next execution. When selecting a configuration that does not allocate all available NUMA nodes, the fastest NUMA nodes will be selected. This is based on performance statistics from the previous execution in which all NUMA nodes participated. Eventually, the binary search will compare two configurations that are next to each other, i.e. there is no selectable configuration in between these two configurations. When this happens, the binary search is finished and the fastest of the two configurations will be selected.

After finding the optimal number of threads and NUMA nodes, the steal policy is evaluated. This is also done through exploration, by employing full-processor work stealing for one taskloop execution. With full-processor work stealing, threads will still prioritize stealing within their NUMA node since only the last portion of tasks are stealable across nodes, as inter-node load balancing should only be required towards the end of the taskloop execution. After full-processor work stealing has been tried, a decision is made whether it was beneficial or if the NUMA-strict stealing should be kept based on which execution was faster. The faster of the two configurations is selected as the optimal configuration. This concludes the exploration stage of the scheduler.




**(a)** General case. **(b)** Specific example.

**Figure 4.2:** Taskloop configuration selection during the exploration stage, including the binary search used to select the optimal configuration {`num_threads, node_mask, steal_policy`}. *Threads* indicate the number of threads selected for execution out of all the available OpenMP threads. *Max* is the total number of available threads, while *Min* represents the number of cores on a NUMA node as this is the smallest possible amount of threads to select. *SP* stands for steal-policy. Figure 4.2a shows the general case, while Figure 4.2b shows a specific example on a processor with 64 cores spread across 8 NUMA nodes.

## 4.3 NUMA-Aware Task Distribution

The *ILAN* scheduler features *NUMA-aware Task Distribution*, with the aim of improving task placement on NUMA architectures. In the case of a loop that operates contiguously on the data, spatial data locality will improve if contiguous iterations are placed on adjacent cores, to lower the memory latencies during true and false sharing between loop iterations. In this case, the proposed task distribution should prove superior to the default OpenMP random task assignment given that the assumption on contiguous data holds. Furthermore, stealing within NUMA nodes is prioritized by marking the first portion of tasks as NUMA-strict, limiting the number of remote tasks steals, an approach previously shown to be effective [10].

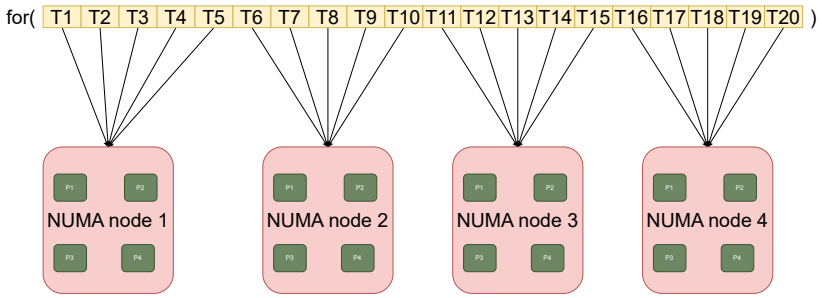



**Figure 4.3:** The *NUMA-aware Task Distribution* on an example processor with 4 NUMA nodes and 4 CPU cores per node.

Figure 4.3 illustrates the task distribution in the scheduler. A newly created task is assigned to a specific NUMA node based on its loop iteration count. The NUMA node placement is assigned as a task attribute and is later used when the task is pushed onto the thread's task queue. All tasks are placed on the first thread of the respective NUMA node. This is done to simplify task placement and reuse the already present OpenMP work stealing mechanism for the fine-grained distribution of tasks within the node. Cores within the same NUMA node are considered equal in terms of data locality aspects, which is why work stealing within a NUMA node is always allowed.

Besides the task placement, the stealing policy is also assigned as a task attribute. The goal of the *NUMA-aware Task Distribution* is that all threads on a NUMA node start by executing tasks within the node. To ensure this, the first portion of tasks distributed to each node is NUMA-strict. The rest of the tasks may be created as NUMA-strict or available for full-processor stealing, depending on the taskloop configuration.

Since the first tasks distributed are NUMA-strict, this means that threads will be blocked from stealing tasks from other NUMA nodes at the beginning of the taskloop execution. Same as in the default OpenMP runtime, as soon as a thread manages to successfully steal a task, it will keep trying to steal from the same thread. The initial NUMA-strict tasks ensure that all threads on a NUMA node will successfully steal from within their node, and continue to steal from within the node until all

tasks on the node have been executed. Once the NUMA node runs out of tasks, the threads will start trying to steal from other threads across the processor. This will only be successful if tasks have been created that are available for full-processor load balancing, which depends on the taskloop configuration. Therefore, the exact portion of NUMA-strict tasks to be created is not relevant, as long as this behavior is ensured. Currently, the portion of NUMA-strict tasks created for each node is approximately a third of the total tasks assigned to the node, a number that has been shown to work as intended.

## 4.4 Runtime Initialization

When executing a program using the OpenMP runtime, the runtime first requires initialization upon entering a parallel region, before moving on to executing the taskloops. In the initialization stage, the runtime makes the necessary setup and configuration that is required before the taskloops can be executed. For the proposed scheduler, additional initialization steps are required, such as thread-to-core pinning, performance counter initialization, and abstracting hardware topology information.

Since the scheduler assumes that OpenMP threads execute on the same core throughout the program execution, thread-to-core pinning is required. This feature is what enables the NUMA awareness and use of performance counters. Since OpenMP supports processor affinity, the functionality required already exists in the runtime. Therefore, thread-to-core pinning is achieved using the already built-in OpenMP runtime macros. Disabling the thread migration between CPU cores is essential to ensure that correct decisions are made in the *Moldability Framework*. If threads migrate between cores, informed decisions on which NUMA nodes to use for execution would not be possible. Furthermore, performance counters are collected per CPU core but accumulated and stored on a thread basis as attributes of the OpenMP thread. This design would not work if threads were to migrate between cores.

Once threads have been pinned to CPU cores, the required performance counters can be initialized. This includes enabling the *perf* event through API calls to the Linux *perf* tool and opening file descriptors for each counter. This results in one counter per event, per thread. Performance counter initialization is performed when threads are launched upon entering the parallel region. As of now, performance counters are not used to make scheduling decisions, but are available for use in future developments of the scheduler. In this project, performance counters were exclusively used for performance analysis.

In addition, hardware topology information is gathered in the initialization stage, through the use of *hwloc* [20]. Topology information, such as the number of cores, NUMA nodes, and sockets available on the platform is extracted. The information is stored in the runtime to be used by the *Moldability Framework* as described in the above sections.

# 5

# Experimental setup

In this chapter, we introduce the experimental setup and benchmarks used to evaluate the proposed scheduler. We outline the software and hardware configurations, followed by an overview of the benchmarks used to evaluate the performance of the scheduler. In Section 5.1, we describe the platform used for evaluation and testing. Section 5.2 introduces the benchmarks used to evaluate the performance of the scheduler, while Section 5.3 describes the metrics of evaluation.

## 5.1 Hardware platform

This work was conducted on an HPC cluster called Vera, provided by NAISS, which consists of AMD Zen4 and Intel Icelake processors. Both of these processors have NUMA architecture topology. In this project, the experiments were performed on the AMD Zen4 EPYC 9354 processors [23], code-named Genoa. Vera has a couple of AMD Zen4 login nodes, which are dual socket processors with a total of 64 cores that are divided into 8 NUMA nodes, i.e. 8 cores per NUMA node and 4 NUMA nodes per socket. Each core has its own L1 and L2 cache, but shares a 32 MB L3 cache with three other cores on the same NUMA node. In total, each processor has 1536 GB of RAM connected. The login nodes are shared between researchers, to use for development and light compilation work. Benchmarks and simulations are instead scheduled on compute nodes that are identical to the login nodes except that they have slightly less connected RAM, 768 GB. The hardware topology of the compute nodes are shown in figure 5.1. This platform is an excellent testbed for evaluating new OpenMP runtime features in various parallel applications. Since it is a NUMA platform, the effects of local compared to remote memory access will greatly accentuate how thread affinity policies can improve performance. Memory intensive kernels and benchmarks run on NUMA platforms will greatly benefit from a NUMA optimized OpenMP runtime, which will help discern whether new features improve or degrade performance.

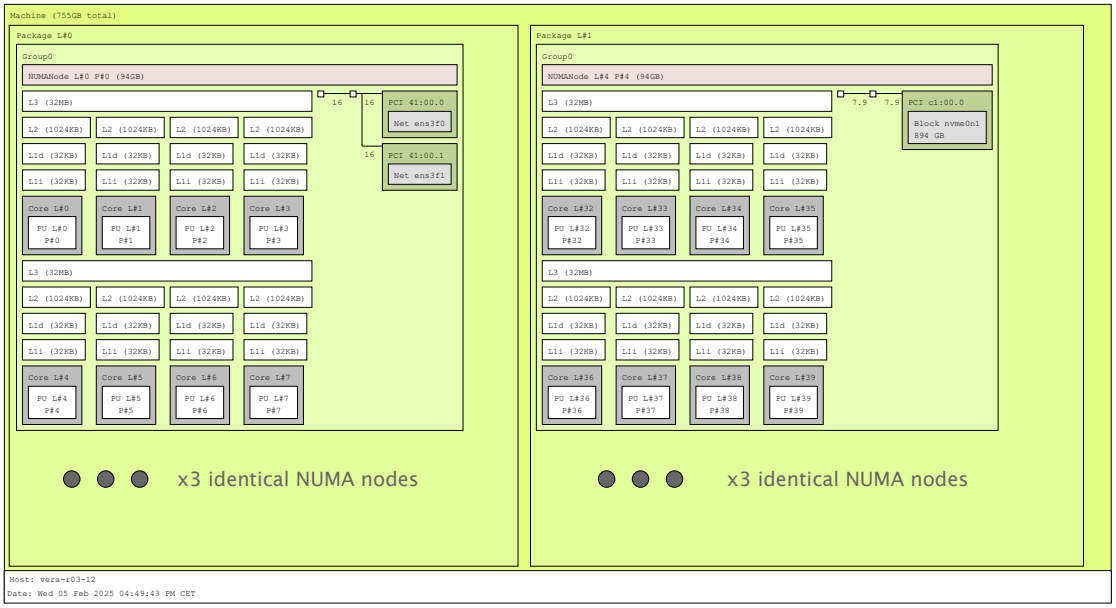



**Figure 5.1:** The topology of the AMD Zen4 EPYC 9354 processors used by the Vera cluster

## 5.2 Benchmarks

Several benchmarks have been used to validate the proposed scheduler's performance and compare its effectiveness for different applications. The selected benchmarks have different characteristics to capture a wider span of application types and to give nuance to the analysis of the scheduler performance. The benchmarks used for evaluation were *LULESH* [24], *NAS Parallel Benchmarks (NPB)* test suite [25], and *Matrix Multiplication (Matmul)*.

Most of these benchmarks originally use work sharing loops, such as `omp for`, for the parallel portion of the applications. Our proposed scheduler operates on loops using tasking, such as `omp taskloop`. Therefore, the benchmarks were rewritten to allow the use of our scheduler. Furthermore, the exploratory approach used in the proposed scheduler requires that taskloops execute a large number of times. Many of the benchmarks are already designed to iterate over the main parallel loops. The size and iteration count of the benchmarks were therefore modified to better suit the analysis of the proposed scheduler.

### 5.2.1 NPB

A widely used test suite within parallel computing is the *NAS Parallel Benchmarks*, *NPB* [25]. This benchmark test suite is designed to evaluate the performance of parallel supercomputers and has implementations for both MPI and OpenMP. *NPB* consists of five benchmark kernels. *Embarrassingly Parallel (EP)* performs floating point operations to stress the processor's computing capacity. *Multi Grid (MG)* computes a 3D scalar Poisson equation which causes non-local memory accesses and

both short- and long-distance communication. *Conjugate Gradient (CG)* tests the caches and memory locality through computing matrix eigenvalues using the conjugate gradient method. *Fourier Transform (FT)* calculates a Fast Fourier Transform of a 3D partial differential equation, causing long-distance communications. *Integer Sort (IS)* sorts an array of integers, which measures the integer computation capacity.

Only three of the kernels were used, as *EP* and *IS* were deemed not interesting for this project. This is because these benchmarks are very simple and do not offer much insight into task interference or data dependencies. Besides these kernels, NPB also contains three pseudo-applications that solve 3D Navier-Stokes systems. These pseudo-applications are *Block Tri-diagonal solver (BT)*, *Scalar Penta-diagonal solver (SP)* and *Lower-Upper Gauss-Seidel solver (LU)*. The specific implementation of NPB for OpenMP used in this thesis is the C++ implementation developed by Löff et al. [26].

All *NPB* benchmarks were executed on size D. Besides modifying all benchmarks to utilize the `taskloop` construct, another change was made for *FT* to increase its iteration count from 25 to 200. This is required to suit the exploratory approach of the proposed scheduler.

### 5.2.2 LULESH

*LULESH* [24] is a balanced and versatile benchmark. The main workload of *LULESH* comes from simulating hydrodynamic models. The benchmark aims to replicate the behavior of typical HPC applications, where this type of simulation is common. The *LULESH* benchmark is very diverse as it has many loops with different characteristics. This gives a broad analysis of the scheduler's performance under different scenarios. *LULESH* was executed with size: 400, iteration count: 200.

### 5.2.3 Matmul

Matrix multiplication is a widely used computational kernel in scientific computing and machine learning. Even though this application has been more commonly offloaded to GPUs in recent years, it is still a valuable benchmark to evaluate runtime schedulers. The benchmark has a predictable memory access pattern with high arithmetic intensity. This type of structured workload provides a good evaluation of how well the runtime scheduler can utilize computing resources. Our implementation of *Matmul* uses a loop size of 3500, and iterates over the same loop 200 times.

## 5.3 Performance Evaluation

For each benchmark, the proposed scheduler was compared to the default OpenMP work stealing scheduler. The default OpenMP scheduler will henceforth also be referred to as the *baseline*. Below is a list of all performance evaluations presented in the upcoming result chapter:

- Performance comparison of the *ILAN* scheduler compared to the baseline.
- Specific performance impact from NUMA awareness.
- Specific performance impact from moldability.
- Performance variability of the *ILAN* scheduler compared to baseline.
- Scheduling overhead analysis of the *ILAN* scheduler.
- Performance comparison of the default OpenMP static work sharing scheduler compared to *ILAN* and the baseline work stealing scheduler.

Besides comparing the *ILAN* scheduler with the baseline, specific functionalities of the proposed scheduler were examined in greater detail, such as the NUMA awareness and moldability. This was done to identify where performance improvements or losses originated from. Furthermore, both the *ILAN* and baseline schedulers were compared to the default OpenMP static work sharing scheduler. This scheduler operates on the work sharing construct `omp for`, which is the default construct used to achieve parallelism in the original version of all benchmarks presented in the above section. By also comparing *ILAN* with the work sharing scheduler, deeper analysis of the strengths and weaknesses of the proposed scheduler could be demonstrated. As discussed in Section 2.2, the tasking schedulers are beneficial for irregular, dynamic, or nested parallel workloads, while the static work sharing scheduler is preferred when iteration workloads are uniform and predictable. Comparing the different scheduler types provides broader insight into the conditions under which *ILAN* achieves the greatest performance gains.

The benchmark results were primarily evaluated using the execution time metric, as improving performance is the main goal of the proposed scheduler. However, performance variability across multiple executions was also analyzed, since lower variance is preferable to achieve more predictable execution times. Therefore, each benchmark was executed 30 times for each of the schedulers. This might not be enough to draw conclusions on performance variability, but it gives an indication of the variance. Also, this reduces the chances of outliers impacting the performance evaluation. Finally, scheduler overhead was measured and compared between *ILAN* and baseline.

# 6

# Results

This chapter presents the results from executing the benchmarks outlined in Chapter 5 using the implemented scheduler described in Chapter 4. In Section 6.1, the results of the final scheduler are compared with the default OpenMP work stealing scheduler. To get a more detailed idea of which features of the scheduler gave rise to which performance gains, the results of NUMA awareness and moldability are shown in Section 6.2 and Section 6.3 respectively. Analysis of the scheduler overhead is provided in Section 6.4. Additionally, the performance variability of the schedulers will be reviewed in Section 6.5. Finally, *ILAN* and the baseline are compared to the OpenMP static work sharing scheduler in Section 6.6.

## 6.1 Performance Overview

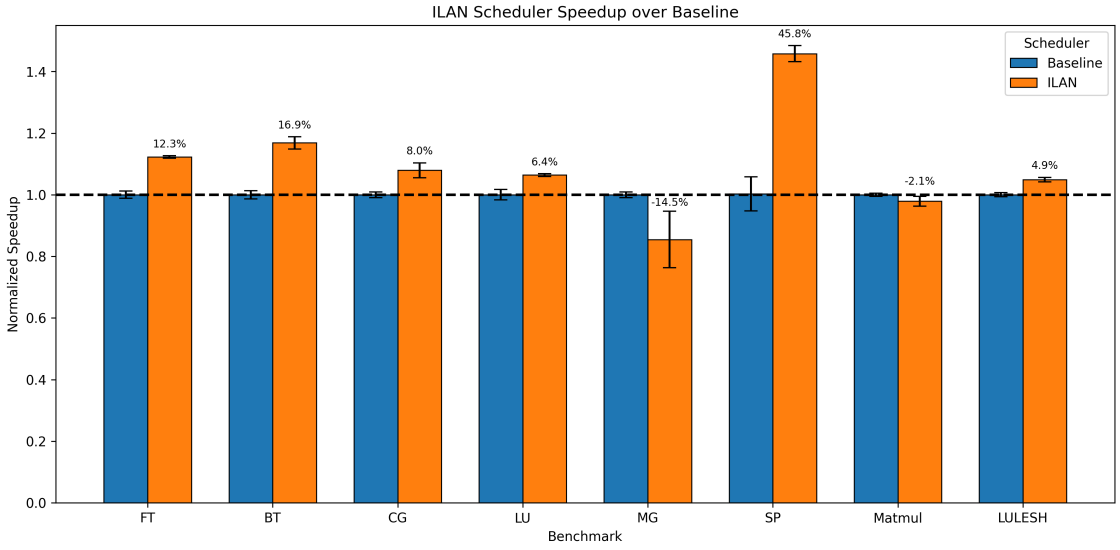



**Figure 6.1:** Normalized speedup of *ILAN* scheduler compared to the default OpenMP work stealing scheduler (Baseline). Higher is better. The execution variance for both schedulers is shown for each of the benchmarks. All benchmarks were run 10 times to ensure reliable results.

Figure 6.1 shows the normalized speedup achieved by the proposed scheduler, *ILAN*, relative to the baseline scheduler. Across all benchmarks used for evaluation, the proposed scheduler consistently improves over the baseline, with improvements ranging from modest gains of a few percentage points to a maximum speedup of 45.8% observed in the *SP* benchmark. On average, we observed a speedup of 9.7%. While the majority of the benchmarks benefited from the *ILAN* scheduler, performance degradations were observed in *MG* and *Matmul*. *Matmul* is a compute-bound benchmark and should reflect a scenario where interference and NUMA awareness do not pay off. Therefore, the lower performance in *Matmul* was expected. In the case of *MG*, the structure of the benchmark does not work well with the *Moldability Framework* causing large performance degradation. The taskloops in *MG* change size throughout the execution, something the proposed scheduler is not equipped to handle. When *MG* is excluded from the analysis, the proposed scheduler achieves an average speedup of 13.2%.

## 6.2 Speedup from NUMA awareness

One of the main features of the proposed scheduler is its NUMA awareness. This includes the *NUMA-aware Task Distribution* and the NUMA-strict stealing policy as outlined in Chapter 4. To investigate the independent benefits of this NUMA awareness compared to the *Moldability Framework*, an intermediate scheduler iteration has also been benchmarked. We call this the *NUMA-aware Scheduler*, consisting of the NUMA awareness while excluding the *Moldability Framework*. Load balancing is also excluded from this scheduler iteration, making it truly NUMA-strict. In this section, the performance and performance variability of the *NUMA-aware Scheduler* will be evaluated compared to the baseline. In Section 6.3, this scheduler iteration will instead be compared to the final scheduler as a whole, allowing us to investigate the specific performance improvements arising from the *Moldability Framework* on top of the NUMA awareness.

Figure 6.2 shows the results from the execution of all benchmarks using the *NUMA-aware Scheduler*. Overall, the *NUMA-aware Scheduler* achieves a performance gain compared to baseline, with an average speedup of 8.7% and maximal speedup of 24.4%. The benchmarks *FT*, *BT*, *LU*, *MG*, *SP* and *LULESH* all see a speedup of between 5-24%, while *Matmul* shows no significant change in performance. *CG* is the only benchmark suffering a performance reduction, by losing 10% in performance. Furthermore, the performance variability is reduced for all benchmarks when using the *NUMA-aware Scheduler*, except again for *CG*.

The performance gain achieved in five of the benchmarks is due to the improved data locality, possibly both at intra- and inter-taskloop levels. The intra-taskloop data locality improves as a result of the contiguous placement of loop iterations on NUMA nodes. While most benchmarks exhibit minimal true sharing between loop iterations within taskloops, false sharing between tasks is likely to occur. The negative effects of false sharing are reduced when adjacent loop iterations are placed to share L3 cache, due to the reduced memory latencies. If two cores compete over a

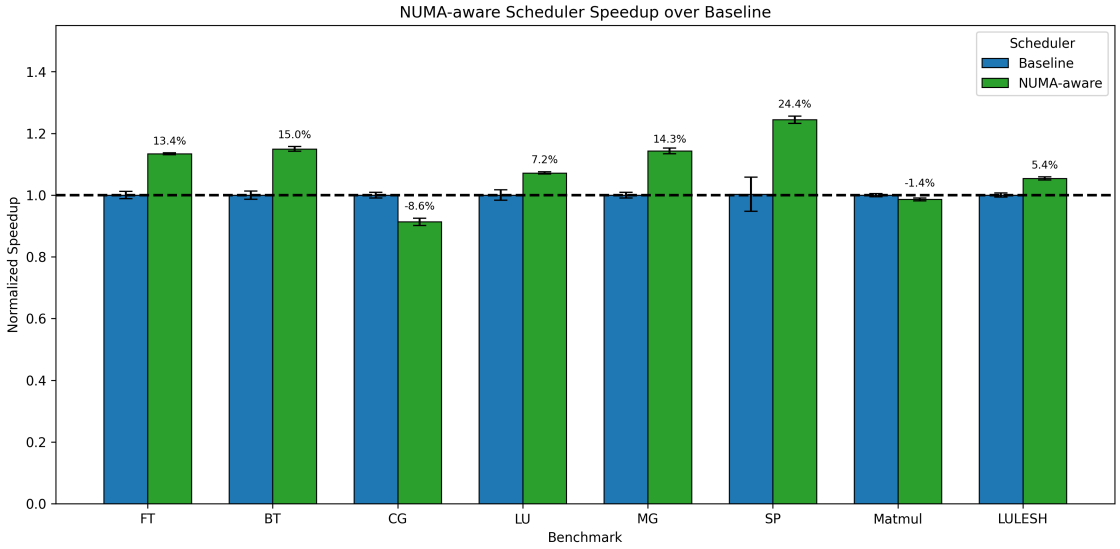



**Figure 6.2:** Normalized speedup of the *NUMA-aware Scheduler* compared to the default OpenMP work stealing scheduler (Baseline). Higher is better. The execution time variance for both schedulers is shown for each benchmark.

cache line, this performance degradation effect is minimized if the cores are sharing L3 cache, i.e. placed within the same NUMA node. There is also a performance increase coming from improved inter-taskloop data locality. Suppose the same data is handled in sequential taskloops, and the taskloops have similar loop iteration sizes. In that case, the tasks will likely be placed on NUMA nodes so that the same node will handle the same data across multiple taskloops. The degree of this behavior depends on both the design of taskloops and the overall application.

The effects of inter-taskloop data locality were tested by changing the *NUMA-aware Scheduler* to mix the NUMA node placement between every taskloop. Originally, loop iteration 0 would always be placed on NUMA node 0. In the changed scheduler version, for every other taskloop encountered, loop iteration 0 would be shifted and placed "half a processor" away, i.e. on node 4 if there are 8 NUMA nodes in total. By doing this, the inter-taskloop data locality is removed, while the intra-taskloop data locality should be preserved. The results from the test show an average 5.2% performance degradation when removing the inter-taskloop data locality compared to the *NUMA-aware Scheduler*.

Two of the benchmarks did not see an increase in performance from the improved data locality. The likely reason for *Matmul* not improving is because it is compute-bound and does not suffer from poor data placement. The slight performance loss when using the *NUMA-aware Scheduler* should come from increased scheduling overhead.

The only benchmark that sees a significant performance loss with the *NUMA-aware Scheduler* is *CG*. The likely explanation for this performance loss is the small size of

taskloops in *CG*. The taskloops in *CG* are very small and often execute in under 1-5 milliseconds (ms). Compared to other *NPB* benchmarks, where the main taskloop often executes for more than 100 ms, this reveals vulnerabilities in the *NUMA-aware Task Distribution* that are not seen in other benchmarks. The issue stems from load imbalance between NUMA nodes caused by the task distribution and the NUMA-strict stealing policy. While tasks are being distributed to the first nodes, other nodes that are yet to receive tasks remain idle. When taskloops are very short, the time taken to distribute tasks becomes significant, causing cores to remain idle for a large portion of the taskloop and thus resulting in this performance degradation.

## 6.3 Speedup from Moldability

To identify the speedup achieved specifically by the *Moldability Framework*, the results presented in Figure 6.1 and Figure 6.2 can be compared. The main difference between the two results is the large speedup in benchmark *SP* and *CG* when using moldability. However, *MG* sees a performance loss when moldability is added, indicating that the *Moldability Framework* comes at a cost and can cause performance degradation in some cases. The other benchmarks have a similar performance with and without moldability.

Both benchmarks *SP* and *CG* saw significant performance improvements due to moldability. Figure 6.3 shows how efficiently a representative taskloop in *SP* executes for configurations tested during the exploration stage. Recall that configurations are selected based only on execution times, and thus the scheduler has no awareness of "efficiency". The efficiency metric used in the figure is calculated as the geometric average $IPC \times n$, where $n$ is the number of nodes used in the taskloop. Importantly, this efficiency metric is only used in this section to illustrate why the scheduling decisions in the representative taskloop provide a performance gain. Although primitive, this metric correlates surprisingly well with the configuration ultimately selected. The depicted taskloop shows the best efficiency at 5-6 NUMA nodes, depending on performance variability during the profiling stage between runs of the benchmark. From figure 6.3, it can be seen that this taskloop runs more efficiently and thus faster on fewer nodes, a performance gain that can be attributed mostly to the *Moldability Framework*. The performance counters and metrics shown in Table 6.1 indicate why the proposed scheduler performs better compared to the baseline. Primarily, it executes the same workload in fewer instructions with a much higher efficiency. The reason why the total number of instructions varies this much depending on the configuration is unclear, but it could be due to changes in task synchronization overhead, threading overhead, or other features of the OpenMP runtime.

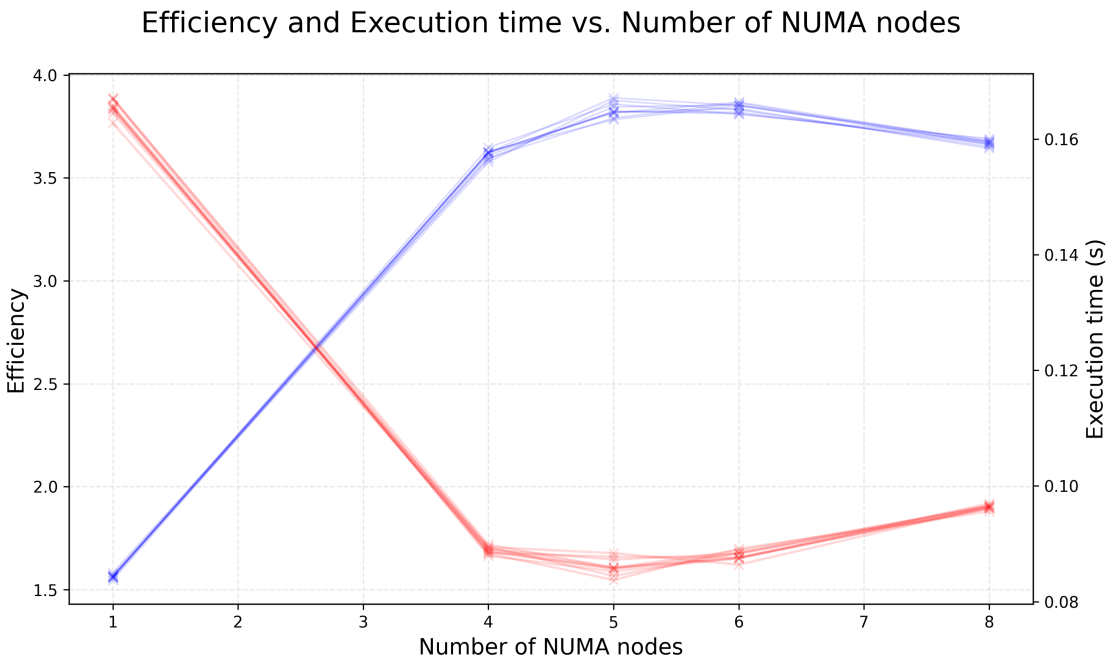



**Figure 6.3:** Efficiency curve for a representative taskloop in *SP* executed with different configurations. The red lines represent execution time and the blue lines represent a simple metric for efficiency. Higher is better for the efficiency metric. In total, the benchmark was run 10 times and a similar trend was seen in all runs.

**Table 6.1:** Performance counters and metrics for the same representative taskloop in *SP* shown in the figure above. *ILAN* utilized 5 NUMA nodes when these metrics were collected. The metrics have been averaged over 500 runs of the taskloop. The efficiency metric is the one described in Section 6.3. Refer to Section 3.3, for descriptions of the performance counters and metrics.

| Counter/Metric | Baseline | *ILAN* |
|---|---|---|
| Execution time (s) | 0.107 | 0.067 |
| Total instructions (millions) | 388 | 313 |
| IPC | 0.317 | 0.844 |
| Efficiency metric | 2.536 | 4.220 |
| Ratio of L1 fills from same NUMA | 0.766 | 0.814 |
| Ratio of L1 fills from different NUMA | 0.234 | 0.186 |
| Backend bound - Memory | 0.873 | 0.705 |

While *SP* and *CG* showed great performance increases due to moldability, *MG* exhibited a performance degradation. This is because *MG* has a set of taskloops that change properties during program execution, most notably, changes in the number of loop iterations. The exploratory approach proposed in this work is not designed to handle such changes. This results in an optimal configuration being selected based on the early executions of the taskloop. The selected configuration

gradually declines in performance due to changing properties of the taskloop, thus explaining performance loss compared to the baseline.

### 6.3.1 Taskloop Configuration Selection

Another aspect of the *Moldability Framework* is the degree to which the reduction in cores is employed. Some benchmarks, particularly those that are less memory-bound, often run taskloops on all cores. Other benchmarks use moldability to reduce interference and thus gain performance.

Figure 6.4 shows the weighted average number of cores employed by each benchmark. Recall that the AMD Zen4 used for benchmarking has a total of 64 cores. The selected number of cores for a taskloop is weighted against the overall execution time of the taskloop, so that larger taskloops have a greater impact on the weighted average. Moldability seems to be used the most in *CG*, as well as in *SP* and *LULESH*. The high reduction of cores in *CG* and *SP* is no surprise, as these are also the benchmarks that gain the most in performance from moldability.

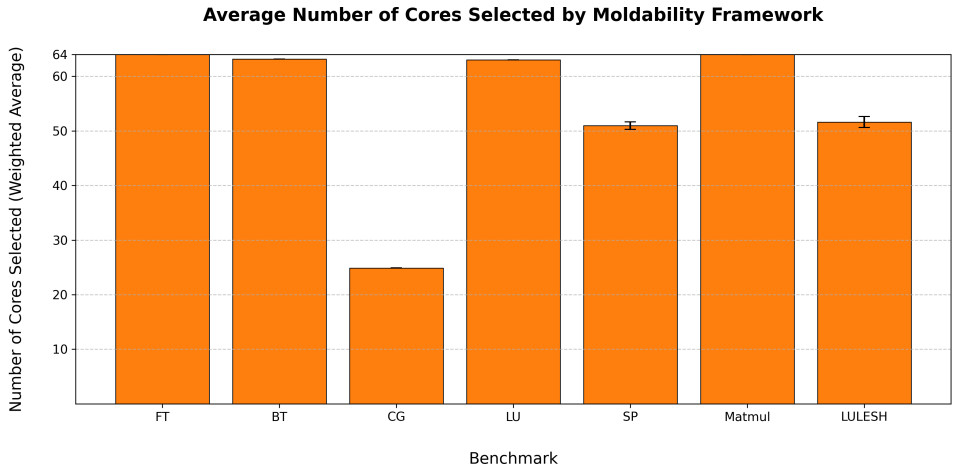



**Figure 6.4:** The weighted average number of cores selected by the *Moldability Framework* in each benchmark. MG has been excluded due to its variable taskloop size.

Why *LULESH* does not see the same performance increase as *SP* is uncertain, as it seems to reduce the number of cores in a similar manner. From analysis of performance metrics when executing *LULESH*, we have seen that this benchmark has relatively high *IPC* and low memory stall cycles, irrespective of the number of cores used. This indicates low interference within taskloops, which could explain why moldability has little impact on *LULESH*. Therefore, even if a lower number of cores is selected, it does not provide a performance increase. The small performance loss compared to the *NUMA-aware Scheduler* could arise due to overhead in the *Moldability Framework*, or possible inter-taskloop effects as a result of some taskloops using fewer nodes for execution.

As seen in Figure 6.4, the rest of the benchmarks utilize close to the maximum num-

ber of cores. This supports our earlier findings that most of the performance increase in these benchmarks comes from the NUMA awareness and not the moldability.

## 6.4 Scheduling Overhead

Since the *ILAN* scheduler supports more advanced scheduling decisions compared to baseline, it should intuitively introduce more overhead. However, due to the complexity of the OpenMP runtime, accurately isolating scheduler overhead is difficult. Therefore, this analysis approximates the overhead by accumulating time spent in the core scheduling components of the runtime for both schedulers and comparing the totals. The results are presented in Figure 6.5.

Interestingly, the accumulated overhead for *ILAN* is lower than that of the baseline for most benchmarks. The reduction is particularly evident in benchmarks where the moldability algorithm selects fewer threads for most taskloops. The effect is most pronounced in *CG*, which is also the benchmark that on average selects the smallest number of threads. Conversely, in benchmarks where all threads are used, such as *Matmul*, *ILAN* yields more scheduling overhead.

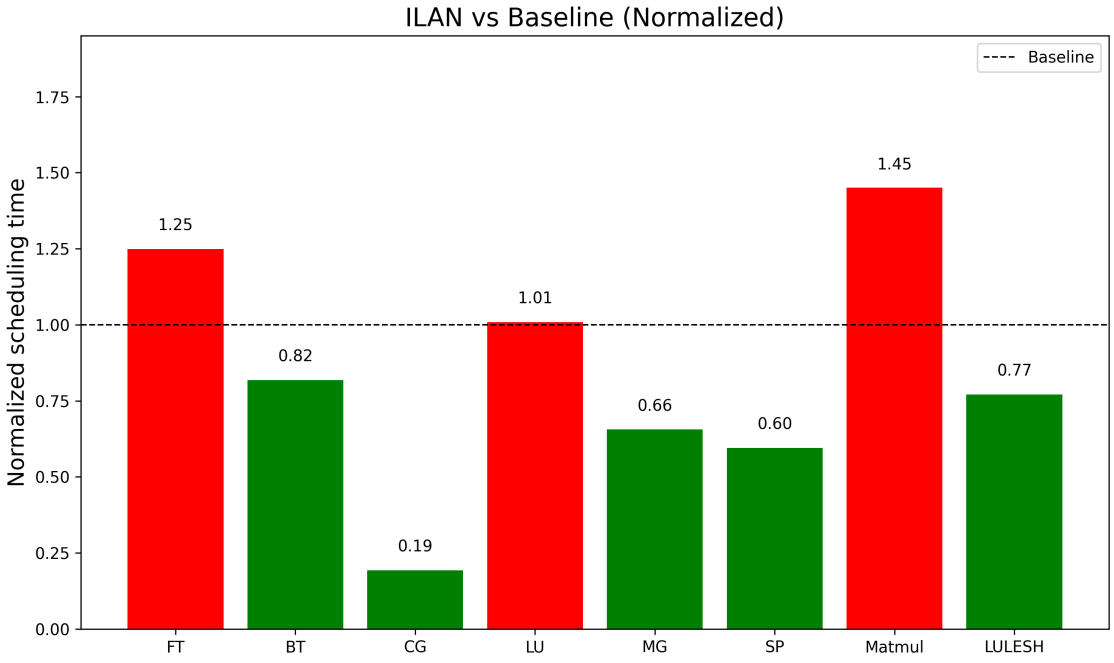



**Figure 6.5:** Total accumulated scheduling overhead for *ILAN* compared to the baseline (normalized), lower is better.

## 6.5 Performance Variability

In terms of performance variability, the proposed scheduler showed a smaller variance compared to the baseline 3 out of 8 benchmarks, see table 6.2 for details. Since each benchmark was only executed 30 times, the values in table 6.2 give an indication of performance variability, but they are insufficient to draw statistically robust conclusions. Particularly, on *FT* and *LU* the variability seems to be negligible, with a standard deviation below 0.005. Compared to the baseline, which for the same benchmarks had standard deviations of 0.0117 and 0.0169, respectively, the proposed scheduler showed a noticeable reduction in variability. The performance variability of *LULESH* seems small for both the proposed scheduler and the baseline, 0.0074 and 0.0065 respectively. For *CG* the situation is reversed, with the *ILAN* scheduler showing higher variability, 0.0239 compared to 0.0094, probably due to the non-deterministic nature of the exploratory approach. As shown in Figure 6.3, there is a slight variability in the performance of identical configurations between different runs of the same taskloop. Consequently, the same taskloop is occasionally scheduled with different configurations, leading to differences in performance between runs of the benchmark. This effect is especially noticeable in *CG* as its taskloops are short and execute frequently, thus reinforcing the effects of sub-optimal configuration selection.

**Table 6.2:** Shows the standard deviation in execution time for each benchmark for the baseline and *ILAN*. Each benchmark was run 30 times

| Benchmark | Baseline | *ILAN* |
|---|---|---|
| *FT* | 0.0117 | 0.0037 |
| *BT* | 0.0133 | 0.0197 |
| *CG* | 0.0094 | 0.0239 |
| *LU* | 0.0169 | 0.0045 |
| *MG* | 0.0091 | 0.0916 |
| *SP* | 0.0554 | 0.0258 |
| *Matmul* | 0.0050 | 0.0158 |
| *LULESH* | 0.0065 | 0.0074 |

For benchmarks where a reduction in performance variability was observed, it appeared to come from the *NUMA-aware Task Distribution*. The predictable and deterministic task distribution of the proposed scheduler should intuitively yield a smaller variability compared to the random task distribution used in the baseline. In contrast, *BT* showed a larger standard deviation for the proposed scheduler than for the baseline, 0.0197 and 0.0133. However, this might not actually be the case as the increase in variability observed for this benchmark is the result of an outlier. An outlier that plausibly could have been the result of frequency scaling or some other event outside the control of the scheduler. Without the outlier, the performance variability of *BT* would be 0.0033 for the proposed scheduler. In general, the results presented in this section are sensitive to outliers, as the benchmarks were only run 30 times each.

For the baseline, *SP* showed the largest standard deviation of all benchmarks by a large margin. One significant factor of the high performance variability is thread migration, as with one-to-one thread-core affinity, the execution time variability of the benchmark is reduced by 50 %, as shown in Figure 6.6. However, disallowing thread migration also reduces performance for this particular benchmark.

*MG* exhibited a tenfold increase in performance variability under the proposed scheduler compared to the baseline. This substantial difference can reasonably be attributed to the structure of the benchmark being poorly aligned with the *Moldability Framework*, as previously discussed in Section 6.1.

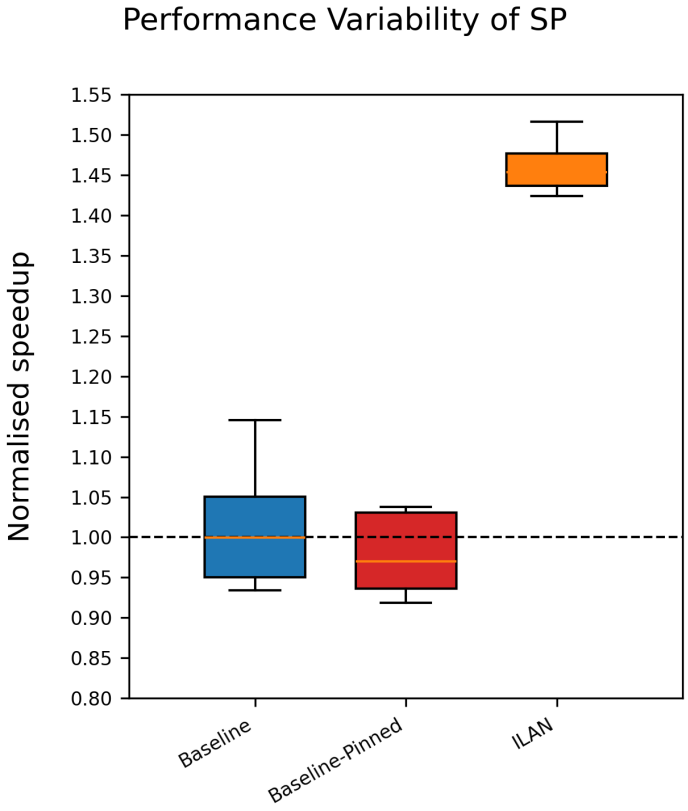



**Figure 6.6:** Shows the normalized speedup and standard deviation in the *SP* benchmark for the baseline with two strategies: No thread affinity and one-to-one thread-core affinity. The two baseline schedulers are compared to *ILAN*.

## 6.6 Comparing Work Stealing vs Work Sharing

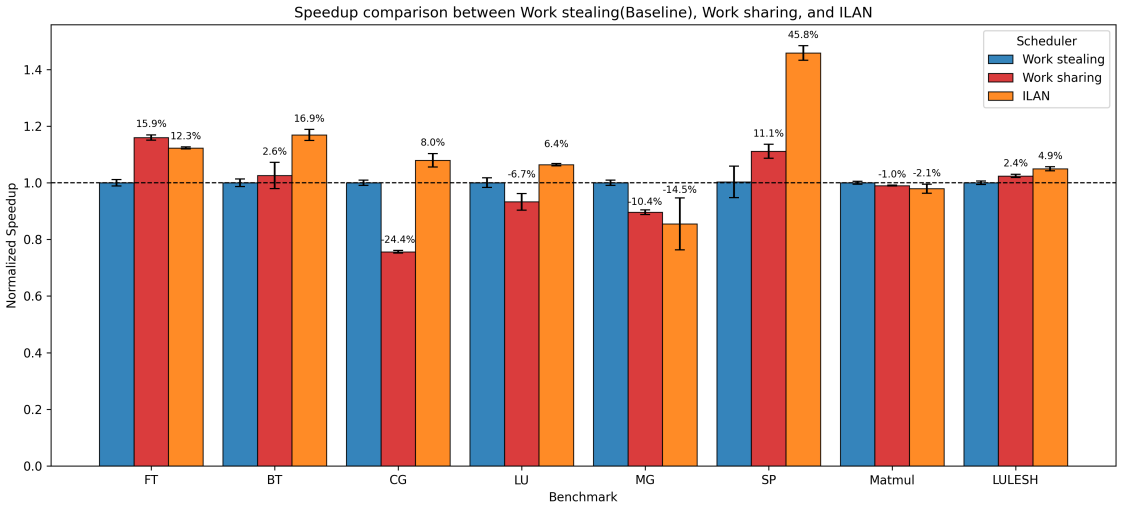



**Figure 6.7:** Normalized speedup comparison of *ILAN* compared to the baseline and the work sharing scheduler.

Figure 6.7 shows the normalized speedup of both the *ILAN* scheduler and the work sharing scheduler compared to the baseline. *ILAN* outperforms the work sharing scheduler on most benchmarks, with a few exceptions. The most notable case is *FT*, where work sharing outperforms not only the baseline but also *ILAN*. As seen in Section 6.2, all performance gain for the proposed scheduler in *FT* was from the NUMA awareness, i.e. by improving data locality and mitigating NUMA effects. Naturally, the work sharing scheduler provided a similar performance gain over the baseline in *FT*, as the absence of load imbalance makes more advanced scheduling unnecessary. Furthermore, the static work sharing scheduler distributes loop iterations across cores very similarly to how *ILAN* distributes tasks. In contrast, the clear benefit of the task-based scheduling approach becomes apparent in most other benchmarks, most notably in *CG*. Unlike *FT*, *CG* has an inherently imbalanced workload, making effective load balancing a key factor for performance. The *MG* benchmark is, as discussed before, a special case due to the varying sizes of taskloops. This seems to be handled best by the randomness of the baseline, while the static work sharing sees a performance loss, most probably again due to workload imbalance.

# 7

# Discussion

In this chapter, the results and analyzes presented in the previous chapter are discussed. Furthermore, we will offer our insights into the strengths and weaknesses of the *ILAN* scheduler and provide suggestions for future implementations.

Overall, the *ILAN* scheduler achieved a significant performance increase over the baseline. From the results, we saw that both NUMA awareness and the *Moldability Framework* contributed to performance gains. The additional comparison with the OpenMP static work sharing scheduler provided deeper insights into the relative strengths and weaknesses of work stealing versus work sharing approaches.

## 7.1 NUMA awareness

From the results presented in Chapter 6, we saw that the NUMA awareness provided a fairly stable performance increase. This was the result of improved data locality, both within and between taskloops. The proposed scheduler mostly focused on optimizing the intra-taskloop data locality. However, inter-taskloop data locality effects were shown to be of importance as well. A different scheduler approach could have been to select taskloop configurations globally, considering many or all taskloops, instead of finding the local optimum for each individual taskloop. This direction was considered during the project, but was discarded since finding a global optimum was deemed too complex to finish with the time left in the project.

Another scheduler approach that could utilize the benefits of inter-taskloop data locality would be to re-bind the data as proposed in previous research [11]. However, doing this requires information about the data to enable re-binding that improves data locality. Gathering this information can be done in one of two ways as we see it: either input from the programmer is required to specify data dependencies within taskloops, or a profiling approach is required to extract the data dependencies. While both of these options are viable, they step away from the simple approach used by the *ILAN* scheduler, which doesn't require any offline profiling or programmer interaction.

## 7.2 Moldability Framework

As for the moldability, the performance impact was more widespread. Our results show that moldability can provide large performance gains in memory-bound applications with significant interference. The exploratory approach should not be very costly, even if all threads end up being used. However, if the taskloop executes very few times or has changing properties, the *Moldability Framework* might cause a large performance degradation as it is not designed to handle these scenarios. This is one of the major weaknesses of the *ILAN* scheduler.

Another drawback of the *Moldability Framework* is the rigidness of the exploration stage. In the current implementation, each explored configuration is only executed once, after which a decision is made to select the optimal configuration. As we saw from the result section, outliers and performance variability are still present in the proposed scheduler even though these effects have been somewhat reduced compared to baseline. The short exploration stage could thus lead to a sub-optimal configuration being selected, for example, if the truly optimal configurations just happen to execute worse than usual the one time it is used. To address this flaw of the exploratory approach, the search could be extended by executing each taskloop configuration multiple times. However, the longer exploration stage incurs a steeper performance penalty and requires extra taskloop iterations to regain the lost performance. Another possible solution would be to keep the exploration stage as is, but later in the program execution explore other configurations again, perhaps based on significant changes in execution time or after a fixed number of iterations. This approach would be advantageous if the application and taskloops change behavior throughout program execution.

Another limitation of the current exploratory approach is that it neglects load balancing during the exploration stage. This is necessary to select which NUMA nodes have the best data locality, and is therefore required during exploration. However, taskloops that have high load imbalance will suffer as load balancing will not be turned on until the configuration exploration is finished. Furthermore, load balancing is only evaluated using the number of cores selected by the *Moldability Framework*. As a result, the truly optimal configuration may never be tested, since only a single configuration is executed with load balancing enabled.

## 7.3 Work Stealing vs Work Sharing

From comparing the *ILAN* and the baseline scheduler with the OpenMP static work sharing scheduler, we saw that the preferable scheduling strategy depends heavily on workload characteristics. We have already discussed the benefits and drawbacks of different scheduling techniques in Section 2.2. However, this analysis is interesting because *ILAN* shares some characteristics with both types of OpenMP schedulers. For example, the *NUMA-aware Task Distribution* resembles the distribution of loop iterations employed by the static work sharing scheduler.

From the results of comparing these schedulers, we saw that *ILAN* achieves the highest performance for most benchmarks. The work sharing scheduler performs better in the *FT* benchmark where the workload is well balanced within taskloops, while the work stealing scheduler performs best in *MG* for the opposite reason. The high overall performance of the *ILAN* scheduler indicates that this scheduler successfully combines benefits from both the OpenMP schedulers. *ILAN* combines the *NUMA-aware Task Distribution* with the possibility of load balancing, which seems to reduce the drawbacks of both the OpenMP work stealing and work sharing schedulers. Furthermore, the addition of interference awareness through moldability gives *ILAN* an extra performance increase. These findings indicate that there exists a gap between the current OpenMP schedulers available, a gap which *ILAN* seems to successfully fill by providing a good balance between the two different scheduling strategies.

## 7.4 Assumptions

The *ILAN* scheduler operates under several assumptions. First, each taskloop needs to be executed a large number of times and should not change properties during the program execution. Otherwise, the proposed scheduler will not be able to adapt, and thus cause severe performance degradation as seen from the results of the *MG* benchmark.

Another assumption is that taskloops are executed without the `nogroup` clause. Currently, the scheduler uses the slowest reported execution time for each NUMA node to determine the overall taskloop execution time of a certain taskloop configuration. With `nogroup` specified, threads will not wait at the implicit barrier after finishing executing their tasks. This would most probably result in inaccurate performance statistics that could cause incorrect scheduling decisions.

Furthermore, assumptions were made regarding the data arrangement in operations within taskloops. For one, the proposed task distribution assumes that loop iterations operate on data contiguously and therefore benefit from the adjacent placements. Additionally, the scheduler assumes that the same data is re-used across multiple taskloops, and thus keeps the task placement intact throughout the program execution. If any of these assumptions do not hold, the scheduler could potentially cause performance degradation. An extreme example of this would be if the first portion of loop iterations has data dependencies with the the second portion of loop iterations, which would cause the interference to spread across the whole processor with the current task distribution policy.

## 7.5 Performance Counters

In theory, performance counters and metrics provide a good indication of the performance characteristics of a taskloop. However, using them to make scheduling decisions proved to be difficult for multiple reasons. The two primary issues are the unreliable nature of the performance counter and the complexity of their implications. Firstly, in short taskloops the sampling rate of many counters is too low, such that comparisons yield unreliable results. Secondly, even though the counters and metrics give good insight into data locality effects and potential bottlenecks, it is most difficult to actually make scheduling decisions that can meaningfully impact their root causes. The simple yet powerful concepts introduced by *ILAN*, such as NUMA awareness and interference mitigation through moldability, yield a good performance increase without employing more advanced online analysis that incorporates performance counters. Nevertheless, the use of performance counters in this project as a tool for detailed offline analysis has proven to be highly useful, as many insights were drawn based on the performance data collected during runtime.

## 7.6 Future Work

There are several interesting directions that one could take to further improve the *ILAN* scheduler. As mentioned earlier, inter-taskloop data locality effects seem to impact the performance more than we originally thought. Exploring a more global scheduling approach, that is aware of the effects between taskloops, could result in further performance gains and mitigate some of the drawbacks of the current implementation. Another interesting improvement would be to explore configurations continuously throughout the execution and not stop the exploration completely after the exploration stage. This would allow the scheduler to adapt to changing environments, such as variable taskloop properties or even changes in the hardware, for example, the effects of DVFS.

Another future improvement to the *ILAN* scheduler could be to integrate performance counters into an online performance model. The attempt to use performance counters and metrics to make scheduling decisions was unsuccessful in this thesis work. However, a deeper analysis of performance counters on a platform with full privileges might yield a different result. The insights could be used for developing a more predictive scheduling approach that mitigates the cost of the exploratory approach presented in the scheduler proposed in this work. Alternatively, performance counters can be used for memory tracing to study task data dependencies. The *ILAN* scheduler is oblivious to what data a taskloop operates on and whether the data accesses are contiguous across loop iterations. Detailed memory access tracing, as possible with privileged system access, could provide a way of further improving the data locality awareness in the *ILAN* scheduler. However, a more fine-grained use of performance counters can soon lead to a performance penalty due to overhead, an important trade-off to consider.

# 8

# Conclusion

The goal of this thesis work has been to improve the OpenMP task scheduler by incorporating data locality and interference awareness, two critical aspects for performance on NUMA platforms. By leveraging hardware topology information, NUMA awareness was implemented, enabling the detection of data locality effects across NUMA nodes. Through the use of moldability, data locality could be optimized while simultaneously detecting the degree of inter-taskloop interference, allowing the scheduler to reduce the number of cores to minimize interference effects.

The work presented in this thesis addresses the gap between the current OpenMP task-based scheduler and previous research. Firstly, our work demonstrates that the impact of both interference and data locality on a parallel application running on a NUMA platform can reliably be detected. This was achieved through the *Moldability Framework* in combination with offline analysis of core-based performance metrics. Secondly, it has been shown that the OpenMP runtime can be enhanced to leverage these insights, improving scheduling performance through interference- and locality-aware strategies. Lastly, hardware topology information was proven to be effective in guiding scheduling decisions, as shown through implementation and evaluation of the proposed *NUMA-aware Task Distribution*. As a result of the above-mentioned strategies, the proposed *ILAN* scheduler achieves an average speedup of 9.7% (13.2% when excluding the *MG* benchmark) and a maximum speedup of 45.8% compared to the default OpenMP work stealing scheduler.

# Bibliography


[1] C. Lameter, "NUMA (Non-Uniform Memory Access): An Overview," *Queue*, vol. 11, no. 7, p. 40–51, Jul. 2013. [Online]. Available: https://doi.org/10.1145/2508834.2513149

[2] U. Drepper, "What every programmer should know about memory," *Red Hat, Inc*, vol. 11, 2007.

[3] M. Diener, E. H. Cruz, and P. O. Navaux, "Locality vs. Balance: Exploring Data Mapping Policies on NUMA Systems," in *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2015, pp. 9–16.

[4] M. Agung, M. A. Amrizal, R. Egawa, and H. Takizawa, "DeLoc: A Locality and Memory-Congestion-Aware Task Mapping Method for Modern NUMA Systems," *IEEE Access*, vol. 8, pp. 6937–6953, 2020.

[5] *OpenMP Application Programming Interface*, OpenMP Architecture Review Board, 2021. [Online]. Available: https://www.openmp.org/specifications/

[6] J. Chen, P. N. Soomro, M. Abduljabbar, M. Manivannan, and M. Pericas, "Scheduling Task-parallel Applications in Dynamically Asymmetric Environments," in *Workshop Proceedings of the 49th International Conference on Parallel Processing*, ser. ICPP Workshops '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3409390.3409408

[7] LLVM Developer Group. LLVM OpenMP Runtime Library. [Online]. Available: https://openmp.llvm.org/

[8] M. A. Suleman, M. K. Qureshi, and Y. N. Patt, "Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on CMPs," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIII. New York, NY, USA: Association for Computing Machinery, 2008, p. 277–286. [Online]. Available: https://doi.org/10.1145/1346281.1346317

[9] J. Ren, C. Liao, and D. Li, "Opera: Similarity Analysis on Data Access Patterns of OpenMP Tasks to Optimize Task Affinity," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2019, pp. 382–386.

[10] K. B. Wheeler, S. L. Olivier, A. K. Porterfield, and J. F. Prins, "Hierarchical Scheduling of OpenMP Tasks in Qthreads." Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2011.

[11] A. Muddukrishna, P. A. Jonsson, V. Vlassov, and M. Brorsson, "Locality-Aware Task Scheduling and Data Distribution on NUMA Systems," in *OpenMP in the Era of Low Power Devices and Accelerators*, A. P. Rendell, B. M. Chapman, and M. S. Müller, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 156–170.

[12] M. Durand, F. Broquedis, T. Gautier, and B. Raffin, "An Efficient OpenMP Loop Scheduler for Irregular Applications on Large-Scale NUMA Machines," vol. 8122, 09 2013.

[13] C. D. Polychronopoulos and D. J. Kuck, "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers," *IEEE Transactions on Computers*, vol. C-36, no. 12, pp. 1425–1439, 1987.

[14] J. Chen, M. Manivannan, B. Goel, and M. Pericàs, "JOSS: Joint Exploration of CPU-Memory DVFS and Task Scheduling for Energy Efficiency," in *Proceedings of the 52nd International Conference on Parallel Processing*, ser. ICPP '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 828–838. [Online]. Available: https://doi.org/10.1145/3605573.3605586

[15] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," *SIGPLAN Not.*, vol. 45, no. 3, p. 129–142, Mar. 2010. [Online]. Available: https://doi.org/10.1145/1735971.1736036

[16] M. Dubois, M. Annavaram, and P. Stenström, *Parallel Computer Organization and Design*. Cambridge University Press, 2012.

[17] A. Rico Carro, I. Sánchez Barrera, J. Joao, J. Randall, M. Casas, and M. Moretó, *On the Benefits of Tasking with OpenMP*, 08 2019, pp. 217–230.

[18] V. Rajput, S. Kumar, and V. Patle, "Performance Analysis of UMA and NUMA models," *International Journal of Computer Science Engineering and Technology*, vol. 2, pp. 1457–1458, 10 2012.

[19] (2024-12-01) Perf: Linux profiling with performance counters. [Online]. Available: https://perfwiki.github.io/main/

[20] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications," in *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, 2010, pp. 180–186.

[21] Linux man-pages project, *numactl(8) — Control NUMA policy for processes or shared memory*, https://man7.org/linux/man-pages/man8/numactl.8.html, The Linux kernel community, 2025, accessed 15 May 2025.

[22] P.-E. Polet, R. Fantar, and T. Gautier, "Introducing Moldable Tasks in OpenMP," in *OpenMP: Advanced Task-Based, Device and Compiler Programming: 19th International Workshop on OpenMP, IWOMP 2023, Bristol, UK, September 13–15, 2023, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2023, p. 51–65. [Online]. Available: https://doi.org/10.1007/978-3-031-40744-4_4

[23] AMD. (2024-10-11) AMD Documentation Hub. [Online]. Available: https://www.amd.com/en/search/documentation/hub.html

[24] L. L. N. Labratory. LULESH Programming Model. [Online]. Available: https://asc.llnl.gov/codes/proxy-apps/lulesh

[25] NASA. Nasa Parallel Benchmark. [Online]. Available: https://www.nas.nasa.gov/software/npb.html

[26] J. Löff, D. Griebler, G. Mencagli, G. Araujo, M. Torquati, M. Danelutto, and L. G. Fernandes, "The NAS Parallel Benchmarks for evaluating C++ parallel programming frameworks on shared-memory architectures," *Future Generation Computer Systems*, vol. 125, pp. 743–757, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X21002831

# A

# Test of Inter-Taskloop Data Locality Effects

To determine to which degree the NUMA awareness performance increases arose as a result of improved inter-taskloop data locality, a small test was conducted. The *NUMA-aware Scheduler* was modified to mitigate inter-taskloop data locality effects. Instead of always assigning the same iteration ranges to the same NUMA nodes, the scheduler was modified to shift the NUMA node assigned for every other taskloop encountered.

For example, on the AMD Zen4 platform, if several sequential taskloops all have an iteration range of 1024, the iterations will always be placed on NUMA nodes according to the following assignments:

```
Node 0 = {0-127}, Node 1 = {128-255}, Node 2 = {256-383},
Node 3 = {384-511}, Node 4 = {512-639}, Node 5 = {640-767},
Node 6 = {768-895}, Node 7 = {896-1023}
```

This assignment will be the same for all taskloops if they have the same size. If the taskloops operate on the same data, this means that the same nodes will operate on the same data over several taskloops. Therefore, the mixed placement will shift the iterations for every other taskloop, to give the following assignments:

```
Node 0 = {512-639}, Node 1 = {640-767}, Node 2 = {768-895},
Node 3 = {896-1023}, Node 4 = {0-127}, Node 5 = {128-255},
Node 6 = {256-383}, Node 7 = {384-511}
```

With these changes to the scheduler, all benchmarks were run again. Figure A.1 illustrates the performance difference between the *NUMA-aware Scheduler* and the scheduler doing mixed placements.

As can be seen, there is quite some performance degradation when removing the inter-taskloop data locality. On average, the modified scheduler performs 5.2% worse.

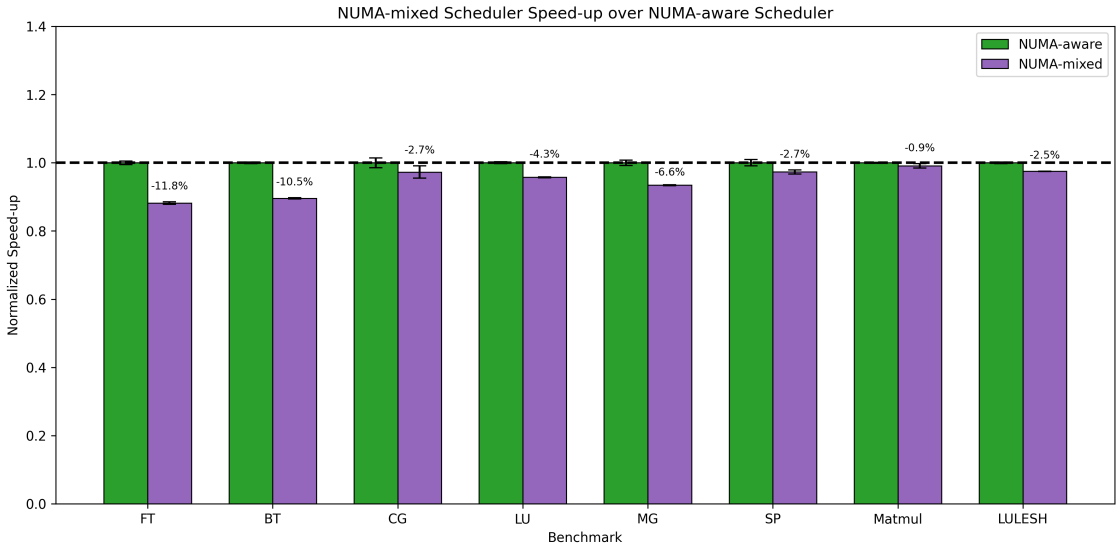



**Figure A.1:** Comparison of the *NUMA-aware Scheduler* and the modified scheduler for mixed iteration assignments between taskloops.

# B

# Repository for the ILAN scheduler

The ILAN scheduler is implemented as an extension of the LLVM OpenMP runtime. The source code used during evaluation of the ILAN scheduler presented in this thesis is publicly available at the release linked below:

`https://github.com/Nosslrac/llvm-project/releases/tag/1.0.0`

The repository includes usage instructions, system requirements, and a summary of known limitations to support reproducibility and further development.