ILAN: The Interference- and Locality-Aware NUMA Scheduler

ILAN: The Interference- and Locality-Aware NUMA Scheduler

Edvin Mellberg*
Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden
edvin.mellberg@gmail.com

Axel Carlsson*
Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden
axel.00@live.se

Jing Chen
Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden
chjing@chalmers.se

Miquel Pericàs
Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden
miquelp@chalmers.se

Abstract

Modern HPC platforms increasingly adopt NUMA architectures, where OpenMP task-based programming model is a standard for enabling dynamic parallelism. However, the default OpenMP runtime is topology-agnostic, and the existing affinity policies are insufficient to ensure optimal performance on modern NUMA architectures. This lack of topology awareness results in suboptimal data locality and performance degradation. Additionally, the current OpenMP standard lacks mechanisms for detecting and mitigating the interference between concurrently executing tasks, further exacerbating the performance degradation. To enhance the performance of OpenMP task-based applications on NUMA architectures, we propose the *ILAN* scheduler: an interference- and locality-aware scheduler, employing moldability to dynamically minimize interference combined with hierarchical scheduling for improved data locality. We implement *ILAN* as an extension of LLVM OpenMP runtime. The results on a 64-core AMD Zen 4 platform show that *ILAN* achieves an average speedup of 13.2%, and a maximum speedup of 45.8% compared to the default scheduler.

CCS Concepts

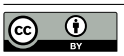
• **Software and its engineering** → *Runtime environments*; **Scheduling**; • **Computer systems organization** → *Multicore architectures*.

Keywords

HPC, parallel computing, scheduling, OpenMP, NUMA, interference, data locality

ACM Reference Format:
Edvin Mellberg, Axel Carlsson, Jing Chen, and Miquel Pericàs. 2025. ILAN: The Interference- and Locality-Aware NUMA Scheduler. In *Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC Workshops '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3731599.3767701

*Both authors contributed equally to this research.



SC Workshops '25, St Louis, MO, USA

ACM ISBN 979-8-4007-1871-7/25/11
https://doi.org/10.1145/3731599.3767701

1 Introduction

In high-performance computing (HPC), parallel execution on multicore architectures is essential to achieve high throughput. Task-based programming models have emerged as effective methods to express and manage parallelism, as they enable dynamic load balancing and flexible scheduling strategies at runtime. Efficient task scheduling, therefore, plays a critical role in achieving optimal utilization of computational resources. One fundamental challenge inherent to parallel computing is mitigating the interference between concurrently executing tasks, often arising due to contention for shared hardware resources such as memory buses, caches, or interconnection networks.

While traditional consumer-level computing platforms typically adopt symmetric multiprocessing (SMP) architectures, characterized by uniform memory access latencies for all cores, these systems exhibit limited scalability beyond approximately 8 to 12 cores due to excessive memory bus contention [17]. To address this limitation, modern HPC platforms typically employ non-uniform memory access (NUMA) architectures, wherein processor cores are physically grouped around dedicated memory units interconnected through high-speed interconnects. Such architectures inherently reduce memory bus contention by distributing memory resources; however, they introduce additional complexities. Specifically, memory access latency becomes dependent on data locality, the proximity between executing cores and the memory units containing the accessed data [14].

Consequently, ensuring optimal data locality becomes a critical factor in maximizing performance on NUMA platforms. Poor data locality can significantly degrade performance, as frequent remote memory accesses incur higher latencies and increase coherence traffic. Conversely, aggressively optimizing for locality can inadvertently cause resource contention within local memory domains, thus leading to uneven distribution of tasks across cores and impaired overall performance [1]. Achieving an optimal balance between data locality and load balance, therefore, represents a non-trivial challenge, strongly dependent upon runtime characteristics and memory access patterns of applications. As NUMA topologies become increasingly prevalent in processor design, the need for

scheduling mechanisms that are aware of the underlying platform architecture becomes important.

OpenMP represents one of the most widely adopted programming models and runtime libraries for shared-memory parallelism [24]. OpenMP supports parallel execution primarily through two mechanisms: work-sharing constructs, such as `omp for`, which distribute loop iterations among threads, and task-based constructs, exemplified by `omp taskloop`, which dynamically create and schedule tasks at runtime, assigning loop iterations to tasks without regard to loop indexes. While task-based parallelism inherently provides excellent load-balancing capabilities, the default OpenMP runtime lacks automatic consideration of hardware topology, leading to task migrations across NUMA domains. Consequently, tasks frequently incur performance penalties associated with poor locality and elevated resource contention. The standard does offer ways of setting the thread-to-core affinity, such as the *proc_bind* clause. However, it relies on appropriate configuration from the programmer, which cannot be guaranteed. Furthermore, the built-in affinity policies *close* and *spread* only provide coarse guidance for thread placement, without consideration of underlying data locality or interference aspects. Additionally, the lack of interference awareness can result in further performance degradation due to resource congestion or data contention, an issue that can be mitigated through the addition of interference-aware scheduling [9].

Motivated by these deficiencies, this paper introduces a new scheduler *ILAN*: The Interference- and Locality-Aware NUMA scheduler. Specifically designed for the OpenMP `taskloop` construct, the scheduler integrates locality-awareness through hierarchical scheduling techniques and interference-awareness through moldability principles. In this work, hierarchical scheduling refers to task placement decisions that are made in layers, aligned with hardware topology. The first layer assigns tasks to NUMA domains to preserve data locality. In the second layer, fine-grain scheduling decisions within the NUMA domains are handled through intra-domain work-stealing. Inter-domain work-stealing is selectively enabled towards the end of the taskloop execution to achieve load balancing if necessary. This task distribution strategy improves data locality while decentralizing scheduling decisions to minimize scheduler overhead. Moldability, in the context of our work, refers to the online selection of the number of threads to employ for the execution of a taskloop. By reducing the active threads and constraining execution to the fastest nodes when interference is high, moldability lowers resource contention and mitigates dynamic performance asymmetry. By coupling these two strategies, the scheduler is able to minimize interference and improve data locality on NUMA platforms, thereby enhancing performance and execution stability on these architectures. Importantly, the proposed scheduler does not necessitate source-level modifications to existing OpenMP applications, provided they already utilize OpenMP taskloops. For applications not currently employing taskloops, adapting loop constructs into taskloops would be required to benefit from the proposed scheduler. To this end, we developed a simple tool to convert `omp for` constructs into `omp taskloop`, used solely as an experimental aid.

We hereby highlight the main contributions of our paper:

- We develop an interference-aware scheduling mechanism, as part of the proposed *ILAN* scheduler, where the active thread-count is "molded" for each taskloop execution to mitigate performance degradation based on online tracing of the execution.
- We present a structured, hierarchical, task placement policy tailored for optimal data locality on NUMA platforms, also embedded in the *ILAN* Scheduler.
- We implement the *ILAN* scheduler within the LLVM OpenMP runtime. The evaluation shows that our scheduler outperforms the default OpenMP task scheduler on a 64-core AMD Zen 4 NUMA platform, achieving an average speedup of 13.2% and a maximum of 45.8% across seven benchmarks.

The outline for the rest of this paper is as follows. Section 2 introduces the background of relevant topics and previous related research. Our proposed *ILAN* scheduler is described in Section 3. Section 4 explains the experimental methodology used, with the evaluation results being presented in Section 5. Finally, the work is summarized and findings from the research are highlighted in Section 6.

2 Background

In the OpenMP runtime, when a thread encounters an `omp taskloop` construct, it partitions the loop iterations into variable number of chunks, each of which is assigned to an explicit task for parallel execution. The main implementation of the OpenMP runtime, namely the LLVM OpenMP runtime library [19], uses dynamic load balancing for task distribution. Threads that finish all the assigned tasks in their own task queues try to steal tasks from other threads' task queues, thus achieving load balance among threads. The dynamic task scheduling strategy of taskloops is simple and effective, especially when there is an imbalance of workload among tasks in the taskloop.

However, such a scheduling strategy may yield suboptimal data locality on NUMA platforms. Without explicit consideration of hardware topology, tasks and their corresponding data may become scattered across different NUMA domains due to the unstructured and random task placement. This results in higher cache coherence overhead and intensified resource contention, leading to performance degradation. Although dynamic scheduling addresses load imbalance by continuously redistributing tasks, the default OpenMP scheduler lacks automatic topology-awareness and instead relies on affinity settings explicitly provided by the programmer. These manually defined affinity strategies may be suboptimal or neglected in practice, exacerbating locality problems and performance asymmetry.

2.1 Hierarchical Schedulers

Hierarchical scheduling addresses the fundamental shortcomings of the conventional flat dynamic scheduling approach employed by the default OpenMP tasking scheduling, where task placement is random in favor of load balancing. On NUMA architectures, such unstructured task placement results in tasks with data dependencies frequently migrating across NUMA domains, which leads to increased remote memory accesses, inflated memory latency, and significant cache coherence overhead.

In contrast, hierarchical scheduling mitigates these data locality issues by explicitly embedding hardware topology into the scheduling, employing a structured, multi-level decision-making process[3][12][15][23][25]. Typically, this involves partitioning the computing resources into locality domains aligned with the physical hierarchy of the hardware, such as sockets or NUMA nodes. Scheduling decisions then follow a layered strategy where one scheduling mechanism may delegate tasks at the domain level, while another scheduling mechanism controls the more fine-grained task placement within the domain. Thus, under normal operation, task scheduling and stealing occur predominantly within local domains, preserving spatial locality by ensuring that tasks and their data remain collocated. More advanced hierarchical schedulers may include memory allocators that proactively determine data placement based on data dependencies among tasks, to improve spatial locality, and possibly temporal locality as well[10][11][13][21][27].

For instance, a previous work proposes to utilize the shepherd-based abstraction within the Qthreads runtime to enable hierarchical scheduling [23]. Shepherds align with hardware locality domains, such as sockets or NUMA nodes, and maintain a group of workers alongside a shared LIFO deque. Tasks execute depth-first within a shepherd to exploit spatial locality from cache reuse. Workers perform remote task steals only upon local domain depletion, transferring chunks of tasks to reduce the required number of steal operations. While this strategy preserves data locality and reduces remote memory accesses, one of the main limitations of purely hierarchical scheduling is its inability to dynamically adapt to runtime conditions. A fixed degree of parallelism might be suboptimal, as workloads commonly consist of several recurring components, some highly parallel, others achieving peak efficiency with reduced parallelism. In this paper, we show that incorporating adaptive mechanisms that adjust the degree of parallelism (i.e., moldability) can significantly improve performance.

2.2 Moldability

In this paper, moldability refers to the property of parallel tasks to dynamically adjust their computational resources, specifically the number of executing threads, in response to runtime conditions. The initial idea of moldability was introduced by Suleman et al. [26], referred to as Feedback-Driven Threading (FDT). Moldability, or FDT, enables scheduling systems to adapt task parallelism based on observed interference metrics, such as synchronization overhead, memory bandwidth saturation, or dynamic performance asymmetry. This adaptability allows tasks to efficiently manage contention by modulating their execution width, effectively balancing resource utilization with interference minimization.

In the context of dynamic runtime environments, moldability offers a significant advantage for mitigating dynamic performance asymmetry caused by resource contention. When parallel tasks experience heightened levels of interference, such as increased contention for shared caches or memory channels, moldability enables the scheduler to constrain the thread count (i.e., task width) dynamically. Reducing the task width alleviates resource pressure, thus mitigating interference and potentially reducing execution time. Conversely, when contention subsides, the scheduler can incrementally increase the task width to improve resource utilization, maintaining efficient throughput.

A prior work extends the moldability concept to mitigate dynamic performance asymmetry caused by factors such as DVFS and interference from unrelated workloads [9]. Their scheduler maintains a *Performance Trace Table* to profile task execution under varying thread and core configurations. By combining criticality-aware task placement with adaptive task-width modulation, the system prioritizes critical tasks on historically faster cores and adjusts the width of non-critical tasks based on performance history, improving throughput while reducing the impact of runtime interference. Our proposed *ILAN* scheduler extends this previous research by utilizing the method of performance tracing and dynamically adapting the thread count based on interference levels. However, our work extends the moldability concept by employing a topology-based, hierarchical scheduling together with a structured task distribution strategy, which together achieves improved data locality and further reduced intra-application interference on NUMA platforms.

Another related work combines hierarchical scheduling with the support for *elastic tasks* [15]. Elastic tasks are tasks that can be co-executed by a variable number of worker threads, depending on the load-imbalance in the system, with the aim of improving data locality. The concept of elastic tasks are similar to the concept of moldability. While this approach was shown to be effective, the strategy relies on affinity hints provided by the programmer. The dependence on programmer input makes the concept less robust, and a fully dynamic approach for determining the degree of *elasticity* or *moldability* can adapt to various types of tasks, applications, or hardware architectures.

3 The ILAN Scheduler

Intra-application interference manifests as dynamic performance asymmetry, where the execution time of a given taskloop can vary significantly depending on runtime contention for shared memory resources such as caches, memory controllers, and coherence traffic. NUMA architectures exacerbate these issues, as task placement directly influences memory latency and interference. The default LLVM OpenMP tasking scheduler employs a random work-stealing algorithm, placing initial tasks onto selected queues arbitrarily and enabling idle threads to steal tasks without considering NUMA topology or contention levels. Consequently, a degradation in performance is to be expected, with an even greater propensity for performance variability.

3.1 Overview

To address these limitations of the current OpenMP runtime, we propose the *ILAN* scheduler, a tasking scheduler that dynamically adapts the number of active threads and determines the task placement for each specific taskloop, minimizing the interference and resource contention. In addition, the scheduler employs hierarchical task scheduling by prioritizing intra-node task stealing over full processor stealing to optimize data locality.

The execution of each taskloop is controlled by three parameters: (1) the number of active threads - `num_threads`, (2) a bitmap

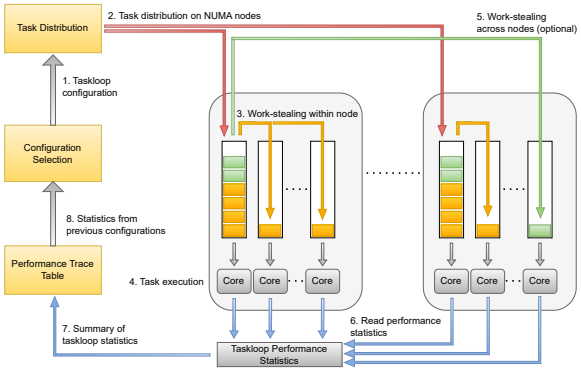


Figure 1: Overview of the *ILAN* scheduler. Grey boxes represent computing domains, containing one task queue for each thread-core pair. Yellow tasks are domain-bound, while green tasks are allowed for inter-node stealing. Green tasks are only created if *steal_policy=full*.

defining active NUMA nodes - `node_mask`, and (3) a task steal policy - `steal_policy` specifying whether inter-node stealing is permitted (`full`) or restricted to intra-node stealing (`strict`). The portion of tasks being stealable when the steal policy is set to `full` is implementation-specific and should be set such that the stealing initiates as intra-node but can switch to inter-node as soon as required. In the *node_mask*, each bit corresponds to one NUMA node, and the bits that are set identify the nodes eligible for task execution, analogous to how a CPU affinity mask specifies the processors on which a thread can execute. The scheduler maintains a Performance Trace Table (PTT), linking different taskloop configurations to measured execution times. The PTT is used during an exploration stage to find the optimal configuration for each specific taskloop, after which the optimal configuration will be used for the remainder of the application execution.

Figure 1 presents an overview of the proposed scheduler. When encountering a taskloop construct, the first step is to select the taskloop configuration. The details of the configuration selection are explained further in Section 3.2. Next, all tasks are distributed onto the NUMA nodes in a structured manner designed to maintain good data locality on NUMA platforms. Tasks are categorized as either node-specific or stealable across nodes, depending on the `steal_policy` as explained above. All tasks are placed in the task queue of the primary thread of each node. After being assigned to a node, tasks are distributed within the node through dynamic work-stealing, which serves as the fine-grained layer of the hierarchical scheduling. The details of the task distribution are explained in further detail in Section 3.3. The fourth step in the workflow is the execution of tasks, where performance tracing of each thread-core pair is performed. The fifth step is optional, and only applies if the `steal_policy` is set to `full`. Here, threads may steal tasks across NUMA nodes upon work depletion within the node. Whether this is allowed is based on the impact of maintaining good data locality versus good load balancing, a decision made during taskloop configuration selection. After all tasks have been executed, the performance statistics are updated in the PTT. Previous executions found in the PTT are used to make more informed taskloop configuration selection decisions the next time the same taskloop is encountered.

3.2 Configuration Selection

The initial step in configuring a taskloop is determining the optimal number of threads. This is achieved by profiling the taskloop's performance across a range of thread counts, with work-stealing strictly confined to the local NUMA node. This exploration phase yields the following crucial information for the scheduler:

- Data locality profile of the taskloop: depending on performance asymmetry across nodes, the taskloop sensitivity to data locality can be estimated, where the nodes with good data locality are prioritized for execution.
- The level of interference: depending on memory intensity and task data dependencies of the taskloop, the taskloop may exhibit improved performance from a reduced number of active threads.

The optimal number of threads is found by exploring different configurations in a binary search-like fashion, starting with the maximum number of available threads and reducing depending on the resulting impact on taskloop performance. The exploration is non-exhaustive, and a local optimum may be selected to favor a reduced exploration cost over performing an exhaustive search. When a configuration with a reduced number of threads is executed, the `node_mask` is selected such that the nodes with the best data locality are utilized. This is found from the PTT when evaluating performance asymmetry between nodes. Once the optimal values for (`num_threads`) and (`node_mask`) have been found, the optimal `steal_policy` is assessed by evaluating the full processor stealing with the optimal (`num_threads`) and (`node_mask`). The exploratory approach necessitates that taskloops within the application execute numerous times, to cover the cost of exploring while benefiting from the optimal configuration.

The control flow for selecting the next taskloop configuration is shown in Algorithm 1. Before this algorithm is called, two prior executions are required to get the fastest and second fastest configurations. The initial configuration explored will always utilize the maximum number of threads, m_{max}, and the second configuration will utilize half of that, i.e. $\frac{m_{max}}{2}$. After these two initial executions, *ILAN* will employ Algorithm 1 for taskloop configuration selection. To ensure that configurations with fewer active threads than $\frac{m_{max}}{2}$ can be explored, the third iteration, $k = 3$, provides a special case where the fewest possible number of threads will be employed, if the configuration with $\frac{m_{max}}{2}$ threads was faster than the m_{max} configuration. The `else if` $threadsDiff \leq g$ statement ensures that the granularity is respected. The last `else` statement in the control flow constitutes the general case, where the next configuration to explore lies between the previously fastest and second fastest configurations. In this way, the taskloop configuration selection over many iterations assumes a binary search-like exploration to find the optimal configuration for each taskloop.

Algorithm 1: Taskloop Configuration Selection

Input : Performance Trace Table PTT,
Current configuration cfg_{cur},
Taskloop iteration count k
Thread-count granularity g

Output: Updated cfg_{cur}, flag `search_finished`

$cfg_{best} \leftarrow$ GetFastest(PTT);
$cfg_{second} \leftarrow$ GetSecondFastest(PTT);
$threadsDiff \leftarrow |cfg_{best}.threads - cfg_{second}.threads|$;
$lowerBound \leftarrow \min(cfg_{best}.threads, cfg_{second}.threads)$;
/* Midpoint rounded down to meet granularity */
$midpointThreads \leftarrow lowerBound + \left\lfloor \frac{threadsDiff/2}{g} \right\rfloor \times g$;

if $k = 3$ ***and*** $cfg_{best}.threads < cfg_{second}.threads$ **then**
  /* Best previous cfg is smallest in PTT, explore the smallest possible cfg */
  $cfg_{cur}.threads \leftarrow g$;
  if $cfg_{cur}.threads = g$ **then**
    `search_finished` $\leftarrow$ **true**;
else if $threadsDiff \leq g$ **then**
  /* Thread counts are within one granularity step, optimal cfg found */
  $cfg_{cur} \leftarrow cfg_{best}$;
  `search_finished` $\leftarrow$ **true**;
else
  /* Explore midpoint unless already executed */
  if $cfg_{cur}.threads = midpointThreads$ **then**
    $cfg_{cur} \leftarrow cfg_{best}$;
    `search_finished` $\leftarrow$ **true**;
  else
    $cfg_{cur}.threads \leftarrow midpointThreads$;

$cfg_{cur}.node\_mask \leftarrow$ GetNUMAMask(cfg_{cur}, PTT, k);
if `search_finished` **then**
  $cfg_{cur}.steal\_policy \leftarrow$ GetStealPolicy(cfg_{cur}, PTT);

The selection of *node_mask* is performed for every configuration selection. The NUMA nodes to execute the taskloop are selected based on the performance of previous executions. The fastest NUMA node is retrieved from the PTT and is selected as the first node of the node mask. To maintain good data locality and efficient inter-node data communication, any additional nodes are chosen according to the NUMA topology. That is, nodes within the same socket are prioritized over nodes crossing socket domains.

The `steal_policy` attribute is kept as `strict`, meaning only stealing within NUMA nodes is allowed, until the `search_finished` flag has been set. Once the search is finished, the steal policy is evaluated by allowing inter-node stealing (`steal_policy = full`) for one execution. After this, the `steal_policy` is kept as the policy that provided the highest performance.

3.3 Task Distribution

The *ILAN* scheduler employs a hierarchical task scheduling approach, where tasks within a taskloop are first distributed across NUMA nodes, after which a more fine-grained task distribution occurs within each node. In addition to benefits in terms of reduced interference and improved data locality, the hierarchical approach also decentralizes most task scheduling decisions, resulting in a less complex scheduling approach. The task distribution strategy employed by *ILAN* is designed to optimize data locality and thereby reduce interference. The following assumptions were made:

- More often than not, data dependencies are higher between adjacent loop iterations than non-adjacent loop iterations, both in terms of spatial and temporal locality.
- Placing loop iterations with high data dependencies between themselves on the same node reduces performance asymmetry through better data locality, lower memory latencies, and thus less interference.

During task generation, tasks are deterministically mapped to individual NUMA nodes based on logical loop iteration indices. All tasks assigned to a specific node are initially enqueued to the node's primary thread and subsequently distributed within the node via OpenMP's local work-stealing mechanism. To preserve locality, the initial fraction of tasks on each node is designated as NUMA-strict, disallowing inter-node task stealing. The remaining tasks are stealable across NUMA-nodes, but will only migrate from their assigned node if the stealing node is fully idle. This approach prioritizes locality but allows for load balancing between NUMA nodes when necessary. Since task mapping and stealing behavior are attributes of the tasks themselves, this decentralized policy operates with minimal scheduling overhead.

The task distribution strategy employed by *ILAN* takes inspiration from previous hierarchical scheduling approaches used in task-based models, such as the multi-threaded shepherds proposed by [23]. However, the performance tracing employed in *ILAN* adds flexibility and awareness to the scheduling, which allows for online tuning of inter-node task migration levels depending on the workload characteristics.

3.4 Affinity Support in OpenMP

OpenMP 5.0 introduced the `affinity` clause on `task` directives, and OpenMP 6.0 extended this support to `taskloop` and `task_iteration`, enabling programmers to provide hints that associate tasks with specific memory locations. The clause can be used to specify data affinity for individual tasks, which enables optimizing data locality and improving cache utilization.

We believe that the *ILAN* scheduler is complementary to this effort but extends beyond affinity in three important ways. First, the `affinity` clause is interpreted by the runtime as a hint, and its effect may vary across implementations. By contrast, *ILAN* enforces structured, hierarchical task distribution with NUMA-aware stealing policies. Second, affinity by itself does not provide interference-awareness: it cannot dynamically reduce contention by adapting the number of active threads or by adjusting to runtime conditions. *ILAN* incorporates such mechanisms through moldability and online performance tracing. Third, while `affinity` can be applied to

`taskloop`, current implementations are limited, and in practice programmers may need to emulate this behavior using explicit tasks and manual annotations. In contrast, *ILAN* requires no source-level modifications for applications that already use taskloops.

In summary, *ILAN* builds upon the locality-awareness enabled by affinity and augments it with adaptivity and automation, providing a more comprehensive solution for NUMA-aware scheduling of taskloops.

3.5 Implementation in LLVM

To evaluate the performance of the *ILAN* Scheduler, the scheduler was implemented in the LLVM OpenMP runtime[1], building on the default work-stealing scheduler. Recall from Figure 1, performance statistics are sampled after each taskloop execution and stored in the PTT. To make the implementation platform agnostic, only execution time has been sampled as a performance statistic. Note that hardware performance counters can easily be integrated into the *ILAN* scheduler and used as a basis for the selection of taskloop configuration. This allows the scheduler to be flexible and can, for example, instead be used to locate and employ the optimal configuration based on other metrics, such as energy efficiency [7, 8]. More performance statistics can also reduce the exploration overhead by utilizing the additional information to arrive at the optimal configuration more quickly. We leave integrating performance counters into the proposed scheduler as a future study.

The thread-count granularity, g, was set equal to the NUMA node size, meaning that full NUMA nodes were employed for each configuration. In other words, nodes were not split as a means to always maximize the spatial data locality even when fewer threads were employed for execution. Furthermore, initial testing showed that maintaining spatial data locality within a compute domain provided performance gains compared to distributing threads across all nodes, even though this would result in lower resource congestion. For other platforms, g equal to the NUMA node size may be suboptimal depending on the platform topology. The thread-count granularity can easily be customized to fit the current platform, and can be assigned any value from 1 to $\frac{m_{max}}{2}$.

To ensure that the performance tracing accurately computes the performance differences between computing domains, thread-to-core pinning is required. This was achieved through existing LLVM OpenMP runtime macros for thread-to-core affinity, together with the *hwloc*[4] API for topology information. The logical threads created by the OpenMP runtime were pinned 1-to-1 to the physical cores, so that the scheduler is able to track which cores perform best for certain taskloops.

4 Experimental Methodology

4.1 Experimental Platform

The scheduler implementation was evaluated on the Vera compute cluster provided by NAISS, utilizing compute nodes based on AMD EPYC 9354 ("Zen4") processors [2]. Each node comprises 64 cores, organized into eight NUMA nodes, with eight cores per NUMA node and four NUMA nodes per socket. Each core features private L1 and L2 caches, while a 32 MB L3 cache is shared among groups of four cores within each Core Complex Die (CCD). Each compute node provides a total memory capacity of 768 GB. Inherent to the NUMA architecture of this platform, significant performance variations arise from local versus remote memory access. Consequently, thread affinity policies greatly influence overall performance. Memory-intensive kernels and benchmarks executed on NUMA-based systems substantially benefit from NUMA-optimized OpenMP runtimes, making this an ideal environment to evaluate the impact of newly introduced runtime features.

[1]Version v19.1.7 of the LLVM OpenMP Runtime Library

4.2 Benchmarks

Seven benchmarks were used to evaluate the performance of *ILAN*. Five applications from the *NAS Parallel Benchmarks (NPB)* suite [22] were used, a versatile and balanced test suite. Furthermore, the *LULESH* [16] benchmark was used, which is representative of typical HPC hydrodynamic workloads with diverse computational loops. Finally, a *Matrix Multiplication (Matmul)* kernel was included to evaluate the scenario of structured memory access and high arithmetic intensity.

The selected NPB kernels were *Conjugate Gradient (CG)*, focusing on memory locality and cache behavior, and *Fourier Transform (FT)*, involving extensive long-distance memory communication. Besides the kernels, the three pseudo-applications included in the NPB test suite were used as well. The 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 used in this work was the C++ implementation developed by Löff et al. [20]. Input size D was used for all NPB applications, with *FT* iterations increased from 25 to 200. *LULESH* was run with a problem size of 400 over 200 iterations, and the *Matmul* benchmark was executed with a loop size of 3500 and 200 iterations.

Since the chosen benchmarks are originally data-parallel and implemented with OpenMP for loops, we convert them to use taskloop to enable evaluation of task scheduling. We acknowledge that this may not reflect inherently task-based applications, therefore we provide a direct comparison against the natural data-parallel baseline in Section 5.6.

5 Performance Evaluation

In this section, we investigate how the proposed scheduler compares to the default OpenMP tasking scheduler (baseline) in several key aspects. The metric used in the evaluation is overall execution time. In addition, we also provide performance variability and scheduling overhead analysis.

5.1 Overall Performance

Figure 2 illustrates the normalized speedup achieved by the proposed *ILAN* scheduler relative to the baseline. Across all evaluated benchmarks, the proposed scheduler generally outperforms the baseline, on average achieving a speedup of 13.2%, with a maximum speedup of 45.8% in the SP benchmark. One notable exception is the standard matrix multiplication kernel, which shows a slight reduction in performance, a result that is expected due to the high computational intensity of the workload and limited sensitivity to NUMA-aware optimizations. Therefore, it scales exceedingly

well with increased parallelism, making moldability ineffective, and hierarchical scheduling unnecessary.

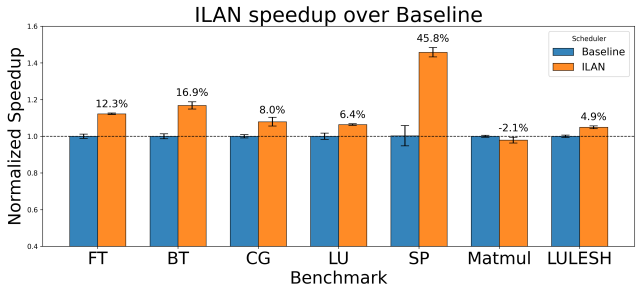


Figure 2: Normalized speedup of the *ILAN* scheduler compared to the default OpenMP work-stealing scheduler (Baseline). Higher is better. The execution variance for both schedulers is shown for each of the benchmarks. All benchmarks were run 30 times to ensure reliable results.

5.2 Configuration Selection

Our evaluation demonstrates significant variation in how NUMA effects influence each benchmark. Consequently, the optimal thread (core) count for a taskloop is not uniform, but instead depends critically on the specific workload being executed.

Our results show that dynamically reducing the number of threads as a means of reducing interference proves to be very effective for CG and SP. These two applications exhibit irregular memory access patterns, leading to memory contention, which is effectively reduced by reducing the number of interfering cores. This argument is further corroborated by the fact that for CG and SP, our scheduler most actively reduced the number of utilized cores, as seen in Figure 3. Across all taskloops in the CG benchmark, *ILAN* on average utilized only 25 cores out of the 64 cores available. This leads to a speedup of 8% compared to the baseline.

For workloads that are either insensitive to interference or scale well with parallelism, *ILAN* does not reduce the allocated cores. In these scenarios, performance gains are attributable to the hierarchical scheduling techniques of the scheduler rather than interference

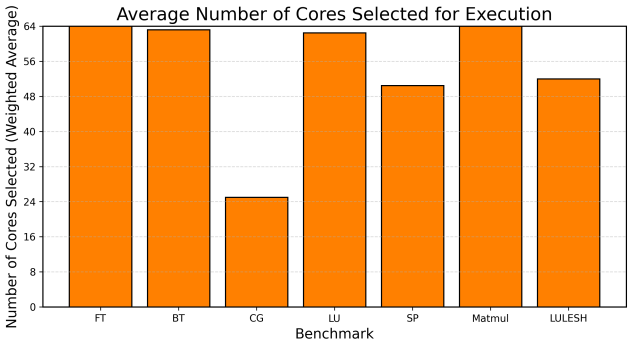


Figure 3: The weighted average number of threads (cores) selected by the *ILAN* scheduler in each benchmark.

mitigation. For instance, the FT and BT benchmarks achieved significant speedups of 12.3% and 16.9%, respectively. These improvements stem from enhanced data locality provided by the hierarchical scheduler, as the thread count was not reduced, meaning that no moldability was employed.

5.3 Performance Gains from Hierarchical Scheduling vs Moldability

To better understand which features of the *ILAN* scheduler provide improvements in different scenarios, the proposed scheduler was also evaluated without one of its core features, the moldability, meaning that all 64 cores were always utilized. The resulting performance is presented in Figure 4. On average, this version of

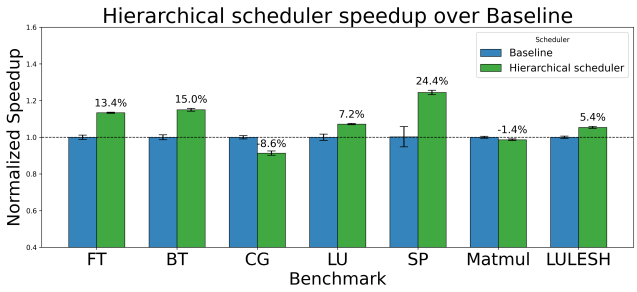


Figure 4: Normalized speedup of the *ILAN* scheduler without the moldability feature compared to the baseline.

ILAN achieved a performance increase of 7.9%. The most notable differences compared to the full version are in the *CG* and *SP* benchmarks. In *CG*, a clear performance decrease of 8.6% is observed in contrast to the 8.0% speedup in *ILAN*, clearly demonstrating the effectiveness of moldability. In *SP*, moldability provides a significant increase in performance. In all other benchmarks, this version of the scheduler slightly outperforms *ILAN*, indicating that they do not benefit from moldability.

5.4 Performance Variability

In terms of performance variability, the proposed scheduler showed a smaller variance compared to the baseline in 3 out of 7 benchmarks, see Table 1 for details.

Table 1: Standard deviation in execution time for each benchmark when using the baseline and *ILAN* respectively. We report the arithmetic average of 30 runs for each benchmark.

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

We believe that reductions in performance variability observed in the benchmarks are a direct consequence of *ILAN's* hierarchical scheduling. The predictable and deterministic nature of the task distribution intuitively yields a smaller variability compared to the random task distribution used in the baseline. However, some of the benchmarks appear to contradict this trend by showing increased variability. For instance, the increase in variability for the *BT* benchmark is an artifact of a single outlier run, attributable to external system noise or the system configurations outside the control of the scheduler, e.g., frequency scaling. In contrast, the baseline exhibits a homogeneous spread of execution times with no clear outlier. Excluding this outlier for *BT* reduces *ILAN's* standard deviation to 0.0033, a significant reduction in variability compared to the baseline.

5.5 Scheduling Overhead

In supporting more advanced scheduling techniques, the proposed scheduler intuitively introduces additional scheduling overhead. Due to the complexity of the OpenMP runtime, accurately isolating scheduler overhead is difficult. Therefore, this analysis attempts to approximate the overhead by accumulating the time spent in the core scheduling components of the runtime. The accumulated time spent by the baseline and the proposed scheduler is summarized and presented in Figure 5.

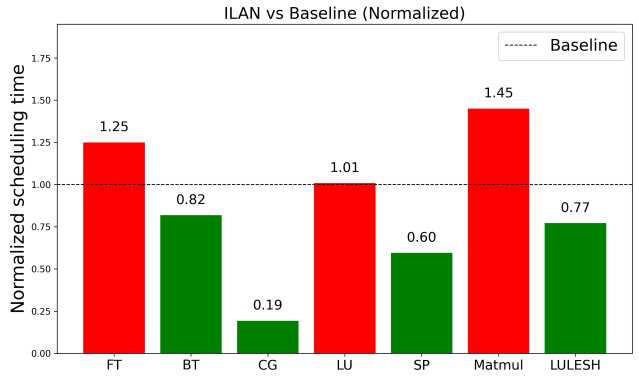


Figure 5: Total accumulated scheduling overhead for the *ILAN* scheduler compared to the baseline (normalized), lower is better.

ILAN demonstrates lower scheduling overhead than the baseline in four of the seven benchmarks. This reduction correlates with the strategy of selecting fewer threads for specific taskloops. The effect is most pronounced in *CG*, where the most aggressive reduction in thread count minimizes synchronization costs. This confirms that for workloads where maximal parallelism is not beneficial, reducing the number of synchronized threads is a key driver of performance. Conversely, for benchmarks like *Matmul* that benefit from all available cores, this scheduling scheme results in a predictable increase in overhead.

5.6 Comparison to OpenMP Work-sharing

Figure 6 shows the normalized speedup of the *ILAN* scheduler and the OpenMP work-sharing scheduler compared to the baseline. The proposed scheduler outperforms the work-sharing scheduler on most benchmarks, with a couple of exceptions. The most notable exception is *FT*, where work-sharing outperforms not only the baseline but also *ILAN*. As seen in Figure 3, the average number of cores selected for this benchmark is the maximum number of cores, indicating that the performance gain can mainly be attributed to the hierarchical scheduling techniques. The lack of load imbalance in the workload makes the work-sharing approach effective.

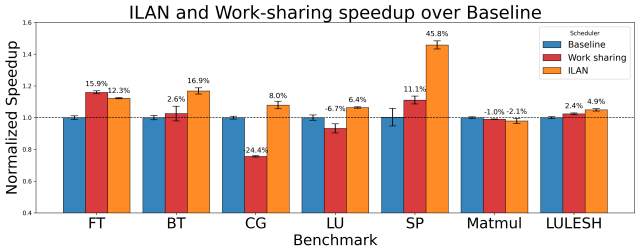


Figure 6: Normalized speedup of the *ILAN* scheduler and the OpenMP work-sharing scheduler compared to the baseline. Higher is better. The execution variance for all schedulers is shown for each of the benchmarks. All benchmarks were run 30 times to ensure reliable results.

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.

6 Conclusion

In this work, we have presented the *ILAN* scheduler, a tasking scheduler implemented as an extension of the LLVM OpenMP runtime, designed to address the issues of suboptimal data locality and high interference encountered when executing taskloops on NUMA architectures. By integrating a hierarchical task distribution strategy, our scheduler explicitly leverages hardware topology information, allowing for improved spatial and temporal data locality. Hierarchical scheduling confines work-stealing primarily to node-local domains, thereby reducing inter-node data migration, coherence traffic, and memory latency. Furthermore, the moldability component dynamically adapts the active thread count for each taskloop, effectively minimizing runtime interference by aligning thread utilization with perceived levels of resource congestion and task data dependencies.

Experimental evaluation on a 64-core AMD Zen 4 platform, using benchmarks from the NPB test suite, LULESH, and Matrix Multiplication, demonstrated consistent performance gains from using *ILAN* in most scenarios, with little-to-no performance degradation in the worst case. The observed average speedup across benchmarks is 13.2%, with performance improvements reaching as high

as 45.8% for the SP benchmark. Our proposed scheduler introduces minimal additional runtime complexity, maintains compatibility with existing OpenMP constructs, and notably reduces performance variability in several workloads due to forced thread pinning. These findings underline the performance gains achieved through the use of the *ILAN* scheduler, and also demonstrate the success of combining interference-awareness, using moldability, and data-locality optimized task distribution through hierarchical scheduling.

Acknowledgments

This work has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No. 956702 (eProcessor), and under grant agreement No. 101034126 (The European PILOT). The JU receives support from the European Union's Horizon 2020 research and innovation programme and Spain, Sweden, Greece, Italy, France, Germany. The computations were enabled by resources provided by Chalmers e-Commons at Chalmers.

References

[1] Mulya Agung, Muhammad Alfian Amrizal, Ryusuke Egawa, and Hiroyuki Takizawa. 2020. DeLoc: A Locality and Memory-Congestion-Aware Task Mapping Method for Modern NUMA Systems. *IEEE Access* 8 (2020), 6937–6953. doi:10.1109/ACCESS.2019.2963726

[2] AMD. Accessed 2025-08-11. *AMD Documentation Hub.* https://www.amd.com/en/search/documentation/hub.html

[3] Joshua Dennis Booth and Phillip Lane. 2024. A NUMA-Aware Version of an Adaptive Self-Scheduling Loop Scheduler. *ACM Trans. Archit. Code Optim.* 21, 4, Article 75 (Nov. 2024), 22 pages. doi:10.1145/3680549

[4] Francois Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. 2010. hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications. In *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*. 180–186. doi:10.1109/PDP.2010.67

[5] Axel Carlsson and Edvin Mellberg. 2025. LULESH. https://github.com/Nosslrac/LULESH A fork of https://github.com/LLNL/LULESH.

[6] Axel Carlsson and Edvin Mellberg. 2025. Nas parallel benchmarks. https://github.com/Nosslrac/NPB-CPP A fork of https://github.com/GMAP/NPB-CPP.

[7] Jing Chen, Madhavan Manivannan, Bhavishya Goel, and Miquel Pericàs. 2023. JOSS: Joint Exploration of CPU-Memory DVFS and Task Scheduling for Energy Efficiency. In *Proceedings of the 52nd International Conference on Parallel Processing* (Salt Lake City, UT, USA) *(ICPP '23)*. Association for Computing Machinery, New York, NY, USA, 828–838. doi:10.1145/3605573.3605586

[8] Jing Chen, Madhavan Manivannan, Bhavishya Goel, and Miquel Pericàs. 2024. SWEEP: Adaptive Task Scheduling for Exploring Energy Performance Trade-offs. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 325–336. doi:10.1109/IPDPS57955.2024.00036

[9] Jing Chen, Pirah Noor Soomro, Mustafa Abduljabbar, Madhavan Manivannan, and Miquel Pericas. 2020. Scheduling Task-parallel Applications in Dynamically Asymmetric Environments. In *Workshop Proceedings of the 49th International Conference on Parallel Processing* (Edmonton, AB, Canada) *(ICPP Workshops '20)*. Association for Computing Machinery, New York, NY, USA, Article 18, 10 pages. doi:10.1145/3409390.3409408

[10] Quan Chen, Minyi Guo, and Haibing Guan. 2014. LAWS: locality-aware work-stealing for multi-socket multi-core architectures. In *Proceedings of the 28th ACM International Conference on Supercomputing* (Munich, Germany) *(ICS '14)*. Association for Computing Machinery, New York, NY, USA, 3–12. doi:10.1145/2597652.2597665

[11] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. 2013. Traffic management: a holistic approach to memory placement on NUMA systems. *SIGARCH Comput. Archit. News* 41, 1 (March 2013), 381–394. doi:10.1145/2490301.2451157

[12] Justin Deters, Jiaye Wu, Yifan Xu, and I-Ting Angelina Lee. 2018. A NUMA-Aware Provably-Efficient Task-Parallel Platform Based on the Work-First Principle. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. 59–70. doi:10.1109/IISWC.2018.8573486

[13] Andi Drebes, Antoniu Pop, Karine Heydemann, Albert Cohen, and Nathalie Drach. 2016. Scalable Task Parallelism for NUMA: A Uniform Abstraction for Coordinated Scheduling and Memory Management. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation* (Haifa, Israel) *(PACT '16)*. Association for Computing Machinery, New York, NY, USA, 125–137. doi:10.1145/2967938.2967946

[14] Ulrich Drepper. 2007. What every programmer should know about memory. *Red Hat, Inc* 11 (2007).

[15] Vivek Kumar. 2020. PufferFish: NUMA-Aware Work-stealing Library using Elastic Tasks. In *2020 IEEE 27th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. 251–260. doi:10.1109/HiPC50609.2020.00039

[16] Lawrence Livermore National Labratory. [n. d.]. *LULESH Programming Model.* https://asc.llnl.gov/codes/proxy-apps/lulesh

[17] Christoph Lameter. 2013. NUMA (Non-Uniform Memory Access): An Overview. *Queue* 11, 7 (July 2013), 40–51. doi:10.1145/2508834.2513149

[18] LLVM Project. 2025. LLVM Project. https://github.com/llvm/llvm-project Accessed: 2025-09-18.

[19] LLVM Project. 2025. *OpenMP — LLVM Project Documentation.* LLVM Foundation. https://openmp.llvm.org/

[20] Júnior Löff, Dalvan Griebler, Gabriele Mencagli, Gabriell Araujo, Massimo Torquati, Marco Danelutto, and Luiz Gustavo Fernandes. 2021. The NAS Parallel Benchmarks for evaluating C++ parallel programming frameworks on shared-memory architectures. *Future Generation Computer Systems* 125 (2021), 743–757. doi:10.1016/j.future.2021.07.021

[21] Marcos Maroñas, Antoni Navarro, Eduard Ayguadé, and Vicenç Beltran. 2023. Mitigating the NUMA effect on task-based runtime systems. *J. Supercomput.* 79, 13 (April 2023), 14287–14312. doi:10.1007/s11227-023-05164-9

[22] NASA. [n. d.]. *Nasa Parallel Benchmark.* https://www.nas.nasa.gov/software/npb.html

[23] Stephen L Olivier, Allan K Porterfield, Kyle B Wheeler, Michael Spiegel, and Jan F Prins. 2012. OpenMP task scheduling strategies for multicore NUMA systems. *The International Journal of High Performance Computing Applications* 26, 2 (2012), 110–124. arXiv:https://doi.org/10.1177/1094342011434065 doi:10.1177/1094342011434065

[24] OpenMP Architecture Review Board 2021. *OpenMP Application Programming Interface.* OpenMP Architecture Review Board. https://www.openmp.org/specifications/

[25] Shumpei Shiina and Kenjiro Taura. 2019. Almost deterministic work stealing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 47, 16 pages. doi:10.1145/3295500.3356161

[26] M. Aater Suleman, Moinuddin K. Qureshi, and Yale N. Patt. 2008. 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* (Seattle, WA, USA) *(ASPLOS XIII)*. Association for Computing Machinery, New York, NY, USA, 277–286. doi:10.1145/1346281.1346317

[27] Han Zhao, Quan Chen, Yuxian Qiu, Ming Wu, Yao Shen, Jingwen Leng, Chao Li, and Minyi Guo. 2018. Bandwidth and Locality Aware Task-stealing for Manycore Architectures with Bandwidth-Asymmetric Memory. *ACM Trans. Archit. Code Optim.* 15, 4, Article 55 (Dec. 2018), 26 pages. doi:10.1145/3291058

Artifact Availability

The source code for *ILAN* is publicly available on https://github.com/Nosslrac/llvm-project.

ILAN is implemented as a fork of the official LLVM project [18], and builds on the LLVM OpenMP runtime with a set of scheduling modifications. The repository's *README* file provides detailed instructions for building the runtime and enabling HWLOC support.

The benchmark suites used in our evaluation, LULESH [5] and the NAS Parallel Benchmarks (NPB) [6], are also publicly available. Their repositories include build instructions, and minor modifications to compiler flags may be required in order to link against the OpenMP runtime built with *ILAN*.

Support for performance counters is available via the `PERF_COUNTERS` macro. Although this feature can be enabled at compile time, it is not currently used by the scheduler to make task placement decisions in the experiments reported in this paper.

Artifact Execution

The workflow of *ILAN* proceeds as follows. First, *ILAN* runtime should be built according to the instructions in *README*. Next, one

or more benchmark programs can be compiled, linking against the *ILAN* runtime built in the previous step. Compilation should be performed for the target platform or directly on the target platform.

All benchmarks depend on the C++ standard library, HWLOC, and the *ILAN* OpenMP runtime. Deployment on the target platform requires the benchmark executable and the shared OpenMP library created from the *ILAN* build, with the C++ standard library and HWLOC available on the target system. ILAN is compatible with any OpenMP-compliant program that utilizes `taskloop` constructs. In some cases, the Linux utility `patchelf` may be used to specify library paths on the compute node.