# HSF: A Hierarchical Scheduling Framework for Application-Tailored Scheduling on Tasking Runtimes

Vincent A. Arcila Larrea
Barcelona Supercomputing Center
Barcelona, Spain
vincent.arcila@bsc.es

David Álvarez Robert
Barcelona Supercomputing Center
Barcelona, Spain
david.alvarez@bsc.es

Antoni Navarro Muñoz
Barcelona Supercomputing Center
Barcelona, Spain
antoni.navarro@bsc.es

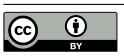
Vicenç Beltran Querol
Barcelona Supercomputing Center
Barcelona, Spain
vbeltran@bsc.es

## Abstract

Modern many-core processors combine deep memory hierarchies with a large number of cores, making both data locality and load balance critical for performance. Fork–join constructs with static scheduling can preserve locality through fixed work-to-core mappings but may suffer severe load imbalance, whereas tasking runtimes balance irregular workloads via dynamic scheduling while offering limited control over task placement and execution order, often at the expense of locality. As different kernels may prioritize locality, load balance, or an intermediate trade-off, current scheduling mechanisms are insufficient to adapt to diverse and evolving application needs.

In this paper, we address this limitation through HSF, a hierarchical scheduling framework built on two core abstractions: predefined scheduling policies and taskgroups. Taskgroups act as independent scheduling domains, allowing tasks from a specific kernel or subcomputation to be executed under a locally defined policy. By composing and nesting these domains, the framework enables complex and dynamic scheduling strategies, naturally supporting hierarchical and multi-kernel applications. The framework supports FIFO, LIFO, priority, and affinity policies and their combinations, with affinity exposed as a first-class mechanism to tune the trade-off between data locality and load balance.

We demonstrate the integration of the framework in the OmpSs-2 tasking model, while keeping the design general enough to be applicable to other programming models. Our evaluation shows that application performance is tightly coupled with underlying scheduling policies, and that HSF widens the available decisions and improves expressivity. On average, HSF achieves a speedup of 1.84x compared to baseline scheduling strategies incorporating advanced NUMA-aware techniques. Overall, this work provides a generic and extensible design towards a scheduling framework that enabled application-tailored decisions in task-based models.



This work is licensed under a Creative Commons Attribution 4.0 International License.
*PACT '26, Chicago, IL, USA*
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2915-7/2026/10
https://doi.org/10.1145/3838684.3846880

## CCS Concepts

• **Computing methodologies** → **Parallel computing methodologies**; • **Software and its engineering** → **Scheduling**.

## Keywords

Scheduling, Task-based programming models, Runtime systems, HPC, Affinity, Locality, Task scheduling

## 1 Introduction

Per-socket core counts have grown into the hundreds while memory hierarchies have grown correspondingly deep, spanning multiple cache levels, several NUMA domains, and now stacked last-level caches. At this scale, both locality and load balance are critical yet difficult to preserve under irregular workloads and runtime variability. Traditional fork–join parallelism with static scheduling can preserve locality via fixed work-to-core mappings, but it often suffers from severe imbalance when workloads are dynamic, irregular, or nested. Task-based programming models address this gap by expressing applications as large numbers of tasks and relying on runtime scheduling to adapt execution, an approach that now underpins kernels in adaptive simulations, sparse linear algebra, graph analytics, and complex workflows.

At the core of task-based execution lies the runtime scheduler, which is responsible for mapping ready tasks onto idle cores. Dynamic scheduling effectively mitigates load imbalance by redistributing work as tasks complete. However, most task-based runtimes provide developers with only limited and coarse-grained control over task placement and execution order. In practice, the available scheduling policies are typically restricted to a small set of predefined strategies (such as FIFO, LIFO, or priority-based queues) that must be selected at program startup and remain fixed throughout execution. Scheduling policy choices have a profound impact on performance, as they determine the ability to exploit data locality across multi-level memory hierarchies, the prioritization of tasks along the critical path, and the ability to overlap computation with communication or I/O. Although dynamic schedulers aim to be work-conserving by avoiding idle resources when ready tasks exist, an exclusive focus on load balancing can degrade locality or delay latency-sensitive tasks, ultimately limiting scalability and efficiency. Consequently, supporting richer scheduling semantics is

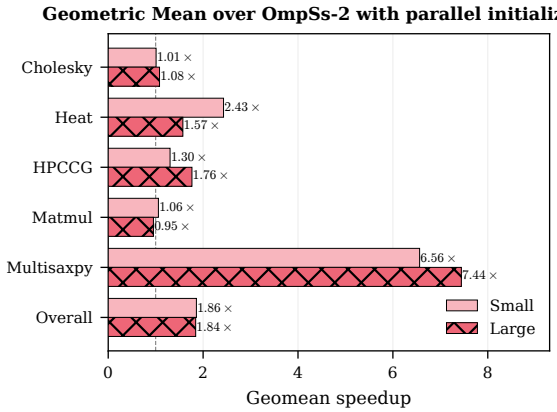



**Figure 1: Geometric mean speedup of HSF over *OmpSs-2 + parallel initialization* across five HPC benchmarks (Cholesky, Heat, HPCCG, Matmul, Multisaxpy) at two problem scales. For each benchmark and scale, the best HSF configuration is compared against the best *OmpSs-2 + parallel initialization* configuration over the same set of block sizes, isolating the contribution of the scheduling framework from that of NUMA-aware initialization. HSF delivers consistent speedups across all benchmarks and both problem scales, with an overall geometric mean improvement of 1.84× on large problems.**

essential: priority mechanisms are required to favor critical activities, while affinity and locality awareness are necessary to exploit modern multicore architectures, all without compromising the load balancing and work-conserving properties that motivate task-based execution.

Many applications would benefit from scheduling that goes beyond a single, global strategy. They are often structured in distinct phases with different performance characteristics (for example, computation-heavy phases followed by communication-dominated ones), and hierarchical applications decompose a large problem into subproblems whose structure, resource usage, and affinity constraints all differ. The established response is to make the *scheduler* hierarchical: queue trees that mirror the cache and NUMA levels of the machine, or disjoint resource partitions each handed to an independent scheduler. Such hierarchies are derived from the hardware, and they express locality preferences well, keeping this task near the core that last touched its data. What they cannot express is an *ordering* that differs between two groups of tasks competing for the same cores, because the hardware offers no basis on which to distinguish those groups.

The structure that does distinguish them belongs to the application. A tiled factorization decomposes work by matrix column, a stencil by block row, a matrix product by output superblock, and a solver by kernel class; each such group has its own reuse pattern and its own preferred execution order. HSF therefore builds its hierarchy over the application's decomposition rather than the machine's: a tree of independent scheduling domains, called *taskgroups*, in which every node carries its own local policy and in which affinity to a topology domain is an orthogonal per-group attribute rather than the organizing principle. This separation is what allows a single application to run two kernels under different orderings on the same cores. Our HPCCG configuration does exactly that, placing the vector operations in a pool partitioned by row range while the sparse matrix–vector product runs under FIFO with strict CCD affinity. No machine-derived hierarchy can represent that distinction.

Providing predefined implementations for every meaningful combination of scheduling policies is neither feasible nor practical: the design space is too large, and application requirements are too diverse and evolving. We instead expose a small fixed set of primitives (FIFO, LIFO, priority, and affinity) that applications compose and nest into the strategy they need. The main contribution of this work is *generality*: the same abstractions and the same runtime deliver gains across five HPC benchmarks with no regression on any of them, obtained by composing this fixed set of policies rather than by writing an ad-hoc scheduler per application, and without modifying the applications' algorithms; scheduling intent is expressed as clauses on task pragmas the code already has. Figure 1 previews the result: across five HPC benchmarks spanning two problem scales, HSF achieves a geometric mean speedup of 1.84× over *OmpSs-2 + parallel initialization*, isolating the contribution of the scheduling framework from that of NUMA-aware data placement (Section 4 details the methodology and per-benchmark results).

## 2 Background

Contemporary multicore processors are characterized by increasingly deep memory hierarchies and rapidly growing core counts, which jointly amplify both performance potential and scheduling challenges. Modern multicores integrate multiple cache levels and NUMA domains to scale bandwidth and improve data locality. As core counts increase, efficient load balancing and locality-aware task scheduling become central to achieving performance scalability without incurring prohibitive synchronization or data-movement overheads.

**Modern Multicore Memory Hierarchy.** Large-scale multicore processors employ hierarchical cache structures (L1, L2, L3) and multiple memory channels to reduce average memory access latency while maximizing sustainable bandwidth. Per-core private caches provide low-latency access to frequently reused data, whereas shared last-level caches (LLC) reduce pressure on off-chip memory and enable data sharing among nearby cores. NUMA architectures further partition main memory across sockets and internal dies, resulting in non-uniform access latencies depending on the relative placement of cores and memory controllers. Consequently, data locality is a first-order performance concern: threads executing on cores close to the memory region holding their working set observe significantly lower latency and higher effective bandwidth than those accessing remote memory, a property that runtime schedulers must explicitly consider.

Modern high-end multicore processors, such as the AMD EPYC 9684X (Genoa-X) used in our evaluation, organize cores into clusters with private L1/L2 caches and shared L3 slices, distributed across several NUMA domains per socket [8]. Access latency and bandwidth therefore depend not only on cache level but also on

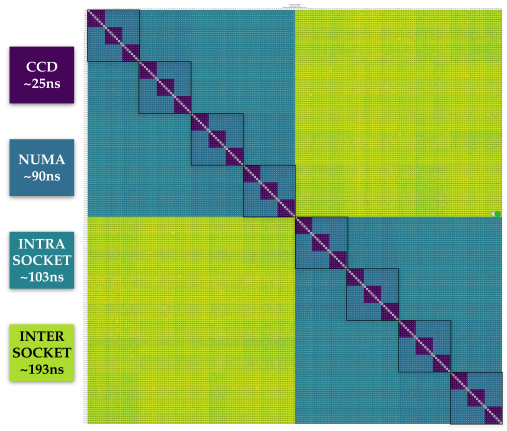



**Figure 2: Core-to-core latencies on a dual socket AMD EPYC 9684X (Genoa-X) system. Squares were added to highlight NUMA domain boundaries.**

whether data resides within the same chiplet, the same NUMA domain, or a remote socket.

In Figure 2, we present core-to-core latency measurements on a Genoa-X system, illustrating the significant variation in access times based on core placement and memory locality (the detailed configuration of the platform we use is described in Section 4). These latency differences underscore the critical importance of locality-aware scheduling policies that can effectively exploit the memory hierarchy to maximize performance.

**Scheduling Policies for Task-Based Runtime Systems.** One of the primary responsibilities of runtime systems is to map application work efficiently onto the underlying hardware resources. Most task-based runtimes create and pin one worker thread per core. When a core becomes idle, its worker thread interacts with the scheduler to obtain the next ready task, with the scheduling policy determining which task is selected and, implicitly, where it is executed within the memory hierarchy.

Common scheduling policies include LIFO, FIFO, and PRIORITY. In practice, these policies are often augmented with mechanisms that improve data locality and scalability. As core counts increase and tasks become fine-grained, the overhead of scheduling itself becomes a dominant factor: contention in the scheduler can outweigh useful computation. Consequently, achieving scalability hinges on minimizing synchronization costs, while at the same time balancing the simplicity of implementing expressive and complex scheduling policies, which directly motivates the choice of locking strategies used to protect scheduler state. Runtime systems typically adopt one of three synchronization strategies below, each representing a different tradeoff between scalability, scheduling expressiveness, and implementation complexity.

**1. Single Global Lock.** A single lock protects all scheduler state, simplifying the implementation of complex policies since the scheduler has a complete and consistent global view. However, this approach often becomes a scalability bottleneck under a high number of workers, or when there is many fine-grained tasks, as all worker threads must serialize on the same lock to create, enqueue, or dequeue tasks. While conceptually simple, such designs can suffer from severe contention on large multicore systems. GNU OpenMP `libgomp` runtime implements this pattern, where task management relies on globally synchronized structures. They implement multi-priority queues for enforcing strict global priority, without having support for the affinity clause.

**2. One Lock per Core (Per-Queue Lock).** In this approach, each worker maintains a private ready queue protected by its own lock. Workers prioritize executing tasks from their queue to preserve locality; if the local queue is empty, they try to steal from other workers' queues. This design distributes contention and generally scales better than a single global lock, but it complicates the implementation of global scheduling policies (e.g., strict priorities), as each worker only has a partial view of the scheduler state. These designs are well known for their scalability and effectiveness on large multicore systems. LLVM OpenMP `libomp` runtime implements this design, which employs per-thread deques and randomized work stealing. They support priority by having globally shared, lock-protected, priority queues for each priority level. The need for a global lock for priority tasks shows how even for simple scheduling policies this design pattern poses complications. As of today, affinity support is non-existent in upstream LLVM, even though attempts to support it have been presented [13].

**3. Global Delegation Lock.** A hybrid approach in which a single lock still protects all scheduler state, but operations are delegated or batched to reduce contention and improve data locality on the scheduler thread. This enables relatively straightforward support for complex scheduling policies while mitigating the serialization costs of a naive global lock. In many scenarios, delegation locking achieves scalability comparable to, or even better than, per-core locking-particularly in asymmetric workloads where only one core generates most tasks. However, for highly recursive applications in which all cores frequently create tasks, per-core queue designs may still be preferable for extremely fine-grained workloads. The nOS-V runtime [19] for OmpSs-2 employs a lock-free multiple-producer single-consumer (MPSC) design for task submission, allowing worker threads to enqueue ready tasks without acquiring a global scheduler lock. A designated delegating worker, which holds the lock, integrates these tasks into the scheduling data structures [5].

## 3 Design and Implementation

We implement HSF on top of nOS-V [19], the runtime backing OmpSs-2, which already exposes a global delegation lock and an MPSC submission path. We reuse the delegation lock to protect the new taskgroup hierarchy and its queues, and expose the taskgroup and affinity APIs to applications without changes to the OmpSs-2 programming model.

### 3.1 Hierarchical Scheduling Framework

**Scheduling Domains and Hierarchical Composition** The scheduling framework is structured around two core abstractions: a fixed set of predefined scheduling policies (e.g., FIFO, LIFO, and priority-based) and taskgroups, which serve as independent scheduling domains. Each taskgroup encapsulates a local ready queue (or an equivalent internal structure) together with a selected policy that defines the order in which ready work is chosen within that domain.

Taskgroups are first-class entities managed by the runtime and are organized hierarchically. A taskgroup may contain both tasks and other taskgroups, forming a tree rooted at a global root taskgroup, which acts as the entry point to the scheduling hierarchy. All tasks and taskgroups belong to exactly one parent taskgroup, except for the root, which has no parent. This organization allows each level of the hierarchy to enforce a local scheduling policy while still participating in a coherent global scheduling process.

Taskgroups can be created, destroyed, and composed dynamically during program execution. This dynamic nature enables applications to construct custom scheduling hierarchies at runtime, switch scheduling policies across different execution phases, and naturally express recursive or nested parallel algorithms. By associating each level of nesting with its own scheduling domain and policy, the framework provides fine-grained control over execution order and resource usage without requiring custom scheduling policies.

**Runtime Scheduling Model** Scheduling decisions are initiated by the server thread holding the delegation lock, starting from the root taskgroup. At each taskgroup, the local scheduling policy is evaluated over the contents of that domain. Policy evaluation yields one of two outcomes: either a ready task directly contained in the taskgroup is selected, or the decision is delegated to one of its child taskgroups.

Delegation proceeds recursively down the hierarchy until a ready task is found. To avoid unnecessary traversal, empty taskgroups are dynamically pruned from the hierarchy. When a taskgroup becomes empty, it is automatically removed from its parent; conversely, when a task is inserted into an empty taskgroup, the taskgroup is re-attached to its parent. As a result, traversal only occurs when ready work exists, ensuring that hierarchical scheduling does not introduce unnecessary overhead.

**Expressing Application-Specific Scheduling**

Application-specific scheduling is expressed programmatically through a dedicated nOS-V API for managing taskgroups. The API allows applications to create and destroy taskgroups, select a predefined policy per domain, and associate tasks with specific domains. The affinity parameter, also exposed by the API, is discussed in a later section. This API is integrated into OmpSs-2 by extending the task construct with the `taskgroup` clause, which specifies that the task will be inserted into the given taskgroup once it becomes ready. Because taskgroup construction is programmatic rather than expressed through a fixed set of runtime calls, the API is intended primarily as a backend for advanced developers and higher-level abstractions, including libraries, DSLs, autotuners, and LLM-based tooling.

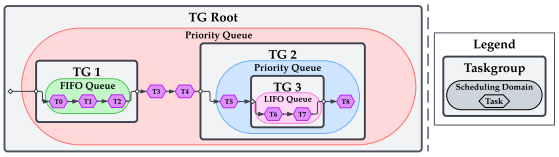



**Figure 3: Example of hierarchical taskgroup composition and scheduling order tuning.**

The example depicted in fig. 3 illustrates the construction of a hierarchical taskgroup tree and the resulting scheduling behavior. Three taskgroups are created in addition to the root taskgroup. Two of them, TG1 and TG2, are direct children of the root. A third taskgroup, TG3, is nested under TG2. Each taskgroup has its own scheduling policy. Tasks are explicitly assigned to taskgroups at creation time: tasks T0–T2 are inserted in TG1, while T3 and T4 are directly inserted into the root taskgroup, and additional tasks are placed within TG2 and TG3 as shown in the figure. The root taskgroup applies a priority policy over its children, and scheduling decisions proceed by hierarchical traversal of the taskgroup tree, combining the local policies at each level. The task numbering and arrows reflect the exact execution order assuming no other tasks become ready. This example demonstrates how the composition of simple, predefined scheduling policies (FIFO, LIFO, and priority) within a hierarchical structure yields non-trivial global scheduling behavior.

## 3.2 Affinity Support

Affinity-aware scheduling aims to improve performance by exploiting hardware locality, executing tasks close to their data to reduce cache miss rates and memory access latency. By constraining execution to processors that share relevant resources (e.g., caches or NUMA-local memory), affinity can significantly enhance temporal and spatial locality.

However, enforcing strict affinity constraints narrows the set of processors eligible to execute a task. Under uneven or phase-varying workloads, this restriction can reduce scheduling flexibility, leading to load imbalance and underutilization of available parallelism.

```c
enum nosv_sched_policy {
  NOSV_TG_AFF_FIFO_POLICY,
  NOSV_TG_AFF_LIFO_POLICY,
  NOSV_TG_AFF_PRIO_POLICY,
  NOSV_TG_PRIO_AFF_POLICY,
  NOSV_TG_FIFO_POLICY,
  NOSV_TG_LIFO_POLICY,
  NOSV_TG_PRIO_POLICY,
  NOSV_TG_NUM_POLICIES,
};

int nosv_task_group_create(
  nosv_task_group_t *grp,
  nosv_sched_policy_t policy,
  nosv_task_group_t parent,
  nosv_affinity_t *affinity,
  int priority,
  const char *label,
  nosv_flags_t flags);

int nosv_task_group_destroy(
  nosv_task_group_t grp);

int nosv_set_task_taskgroup(
  nosv_task_t task,
  nosv_task_group_t grp);
```

**Listing 1: Task Groups API**

```c
nosv_affinity_t nosv_affinity_get_strict(
  nosv_affinity_level_t level, uint32_t sid);

nosv_affinity_t nosv_affinity_get_flexible(
  nosv_affinity_level_t acc_level,
  nosv_affinity_level_t pref_level,
  uint32_t pref_sid);

int nosv_set_task_affinity(nosv_task_t task,
  nosv_affinity_t *affinity);
```

**Listing 2: Affinity API**

Consequently, affinity introduces a fundamental tradeoff between locality and work-conservativeness: stronger locality guarantees typically come at the cost of reduced load balancing.

Our design addresses this tradeoff by explicitly separating preference from admissibility over topology domains in the hardware hierarchy, defined as sets of processing elements derived from the hardware topology hierarchy. These topology domains capture the natural locality boundaries of the system, ranging from coarse-grain scopes such as a compute node (NODE) or NUMA node, to finer-grain groupings including user-defined complex sets of cores sharing a specific resource (e.g., an L3 cache), individual cores, and hardware threads. By expressing affinity over such domains, the scheduler can reason explicitly about locality while retaining a structured view of the available execution resources.

Within our scheduling framework, we support two affinity modes. **Strict affinity** constrains a task to execute exclusively within a single topology domain. This mode provides strong locality guarantees, as execution is limited to processors sharing the targeted resources, but it also restricts scheduling freedom and may lead to underutilization.

In contrast, **flexible affinity** separates locality intent from admissible execution scope: a task declares a *preferred* topology domain together with a broader *acceptable* domain. Tasks run in the preferred domain whenever possible, and may migrate to cores within the acceptable domain when the preferred scope is saturated. This provides a middle ground between unconstrained and strict affinity, preserving locality when resources are available while avoiding idle cores under uneven workloads. For instance, a task may prefer an L3 cache group while accepting any core within the enclosing NUMA node, retaining cache reuse and enabling load balancing across the same memory locality otherwise.

The affinity API, shown in listing 2, provides constructor functions for strict and flexible affinity objects, and a setter to associate affinity specifications with tasks.

**Impact of Supporting Affinity in Hierarchical Scheduling.** Affinity constraints substantially complicate hierarchical task scheduling. In a taskgroup hierarchy, a traversal that is structurally valid does not necessarily lead to a task that can execute on a given CPU. A naïve traversal strategy may descend into subtrees that contain ready tasks, yet none for which their affinity constraints allow running on the requesting processor. In the worst case, the scheduler may traverse a large fraction of the hierarchy and still return no executable task. Such failed traversals introduce significant overhead, increasing scheduling latency and harming scalability, particularly as core counts grow.

To address this problem, the implementation ensures that hierarchical traversal is initiated only when the existence of a suitable task is guaranteed. For each affinity domain A in which a taskgroup contains ready tasks, the taskgroup propagates a representative indication to its parent, signaling that processors belonging to domain A can successfully execute work within that subtree. This information is propagated recursively up the hierarchy until it reaches the root taskgroup.

Consequently, the root maintains an aggregated view of which affinity domains (and, by extension, which CPUs) have executable work available. This knowledge enables the scheduler to avoid descending into subtrees that are guaranteed to fail affinity checks, thereby eliminating unnecessary traversal and reducing scheduling overhead.

**Algorithm 1:** Affinity-aware task extraction. ATTACH maintains the invariant that makes DESCEND backtrack-free; GETTASK exhausts every preferred domain before any acceptable one.

**Notation:** $c$: requesting CPU, with ancestor domain $c.\text{anc}[\ell]$ at topology level $\ell$ (CPU finest → NODE coarsest). $d = (\ell, id, \tau)$ is a domain of class $\tau \in \{\text{PREF}, \text{ACC}\}$. Each affinity-aware taskgroup holds one ready-work counter $cnt[d]$ per domain. Peek($g$,$d$) applies $g$'s local policy within $d$, returning a task, a child taskgroup, or NIL.

```
1  Procedure GetTask(g, c):     // policies AffFifo, AffPrio
2      for τ ← PREF down to ACC do
3          for ℓ ← CPU down to NODE do
4              d ← (ℓ, c.anc[ℓ], τ); u ← Descend(g, d);
5              if u ≠ NIL then return Retrieve(Peek(u, d), c, d);
6          end
7      end
8      return NIL;
9  Procedure Descend(g, d):  // deepest taskgroup with work for d
10     if Peek(g, d) = NIL then return NIL;
11     u ← g;
12     while Peek(u, d) is a taskgroup do
13         u ← Peek(u, d) ;                          // recurse
14     end
15     return u;
16 Procedure Attach(g, d): // on insertion; Detach symmetric
17     cnt[d] ← cnt[d] + 1;
18     if cnt[d] = 1 and g is not the root then
19         link g into parent(g) under d; Attach(parent(g), d);
20     end
```

**Invariant:** $g$ is linked under $d$ exactly while $cnt[d] > 0$, so a child returned by PEEK always has a task below it.

With this mechanism in place, hierarchical traversal is conditioned jointly on task readiness and affinity compatibility. Paths that cannot yield an affine task for the requesting CPU are never explored, and traversal considers only those child taskgroups whose propagated state indicates the presence of affinity-compatible ready tasks. This design bounds traversal cost and ensures that hierarchy walks are proportional to productive scheduling work, preserving the scalability benefits of hierarchical scheduling while enabling efficient enforcement of affinity constraints. 1.

## 4 Evaluation

This section evaluates HSF on three fronts: the performance it delivers, how applications configure it to obtain that performance, and the overhead its new scheduling infrastructure introduces. We start with a case study of the multisaxpy kernel, which is a simple benchmark that benefits from both affinity and policy tuning. We then evaluate HSF on a set of five benchmarks, comparing against three baselines: (a) an *AMD-Math/LLVM OpenMP* baseline (a vendor-optimised BLAS/LAPACK implementation or LLVM

OpenMP, depending on the benchmark; defined per-benchmark in Section 4.3), (b) *OmpSs-2 unmodified*, and (c) *OmpSs-2 + parallel initialization*. Finally, we present an overhead study using *SchedStress*, a synthetic micro-benchmark we developed to compare HSF against other runtimes across a wide range of task granularities.

## 4.1 Environment

The experiments were run in a dual-socket AMD EPYC 9684X (Genoa-X). This processor integrates 96 cores per socket, based on Zen 4. On each socket, cores are organized into 12 Core Complex Dies (CCDs), each with 1 Core Complex (CCX), which in turn each has 8 cores [8]. Each CCX has a private L3 cache, shared by cores in that CCX. In the Genoa-X design, stacked 3D V-Cache increases the effective L3 capacity of each CCX from 32MiB to 96MiB, yielding an aggregate L3 of approximately 1.1 GB per socket. The NUMA configuration of the system is such that there are 4 NUMA nodes per socket, each comprised of 3 CCXs.

This machine runs NixOS 25.11. For our baseline, we used nOS-V 4.0.0, NODES 1.4.0 and the 2025.11 OmpSs-2 LLVM compiler. The HSF implementation branches from these versions implementing the new scheduling infrastructure, and uses a modified version of the compiler supporting the taskgroup clause for tasks. For some benchmarks relying on CBLAS or LAPACKe, we used AMD-BLIS 5.1 and AMD-Libflame 5.1 [1, 2, 18] which are optimized for AMD systems. For OpenMP executions, we use LLVM Clang 21.1.2 with its bundled LLVM OpenMP runtime, and GCC 14.3.0 with `libgomp` 14.3.0. Where we need explicit NUMA-aware initialization, we use libnuma 2.0.14.

## 4.2 Multisaxpy case-study

Multisaxpy is a simple kernel that performs multiple saxpy operations in parallel, i.e., $Y_i = \alpha_i X_i + Y_i$ for $i = 1, \ldots, N$ iteratively. This benchmark is designed to be memory-bound and benefits from both spatial and temporal locality.

**Taskgroup Configuration.** Tasks are distributed across taskgroups using $\mathrm{TG}_{\mathrm{id}} = T_{\mathrm{blockid}} \bmod N_{\mathrm{domains}}$, where $T_{\mathrm{blockid}}$ is the task's block identifier and $N_{\mathrm{domains}}$ is the number of topology domains. Each taskgroup has affinity to a specific topology domain, grouping tasks operating on contiguous data blocks within the same memory hierarchy region.

Within each taskgroup, priority-based scheduling assigns priorities as $P_t = N_{\mathrm{tasks}} - \mathrm{blockid}$. This encourages newly released tasks with the same block identifier to execute immediately on the core that just completed their predecessor, increasing the probability of cache reuse.

**Table 1: Performance counter comparison between FIFO per CORE and Priority per CORE configurations. L3 numbers are calculated over the L2 misses. – pti = per thousand instructions.**

| Metric | FIFO per CORE | Priority per CORE | Delta |
|---|---|---|---|
| L2 Miss | 25.33 pti | 28.45 pti | 12.32% |
| L2 Hit | 42.19 pti | 71.29 pti | 68.97% |
| L3 Miss | 70.79% | 25.49% | -63.99% |
| L3 Hit | 29.21% | 74.51% | 155.08% |

**Experimental Setup.** We configured the benchmark so that total vector size is ten times (10x) the aggregate L3 cache, with block sizes occupying 90% of a core's L2, yielding 13107 tasks per iteration on 192 cores. We evaluate four configurations: (1) Priority per NUMA node (2) Priority per complex set (L3 domain) (3) Priority per core (4) FIFO per core.

**Analysis of the impact of policies and taskgroups.** In Figure 4, Priority per CORE (third row) yields the lowest wall time and is the best of the four. Under all priority configurations the block-ID and iteration-ID columns confirm the intended behaviour: each core progresses depth-first through its assigned blocks and reuses them across successive outer iterations, indicating that successor tasks on the same data are scheduled to the core that just touched it.

Table 1 reports performance counters (gathered with AMD uProf 5.1 [3]) for the two best configurations, FIFO per CORE and Priority per CORE. Relative to FIFO, Priority increases L2 hits by 68.97% and L3 hits by 155.08%, while reducing the L3 miss rate by 63.99%. L2 misses also rise modestly (+12.32% pti), but the dominant signal is the sharp drop in L3 misses, which directly reflects reduced off-chip memory pressure and explains Priority's wall-time advantage.

Together, the traces and counters establish two complementary points. First, taskgroup affinity binds tasks to a topology domain, providing spatial locality. Second, the per-taskgroup priority policy reorders ready tasks within each domain to exploit application-level structure (here, the per-block successor relationship) producing temporal locality that the runtime cannot infer on its own. These two mechanisms compose: neither alone is sufficient, and the framework allows them to be specified independently per scheduling domain.

## 4.3 Benchmark evaluation

We compare four versions: (1) *OmpSs-2 unmodified*, the existing nOS-V scheduler without taskgroups; (2) *OmpSs-2 + parallel initialization*, which extends (1) with parallel, NUMA-aware initialization of application data structures so that pages are first-touched across all NUMA nodes; (3) *HSF*, which builds on (2) by using our flexible scheduling framework with per-benchmark taskgroup configurations; and (4) an *AMD-Math/LLVM OpenMP* baseline (vendor-optimised BLAS or LAPACK, or LLVM OpenMP, depending on the

**Table 2: Per-benchmark parameters for both small and large problem sizes. Small configurations execute on a single socket; large configurations execute on both sockets.**

| Benchmark | Size | Problem size | Memory | Iters |
|---|---|---|---|---|
| Cholesky | Small | 6,144 | 288 MiB | 1 |
| | Large | 33,792 | 8.5 GiB | 1 |
| Heat | Small | 12,288 | 1.125 GiB | 800 |
| | Large | 49,152 | 18 GiB | 60 |
| HPCCG | Small | 288×192×768 | ≈19.6 GiB | 100 |
| | Large | 384×384×1,536 | ≈105.1 GiB | 40 |
| Matmul | Small | 6,144 | 864 MiB | 100 |
| | Large | 49,152 | 54 GiB | 1 |
| Multisaxpy | Small | 56,524,800 | 862 MiB | 1000 |
| | Large | 771,740,160 | 11.5 GiB | 100 |

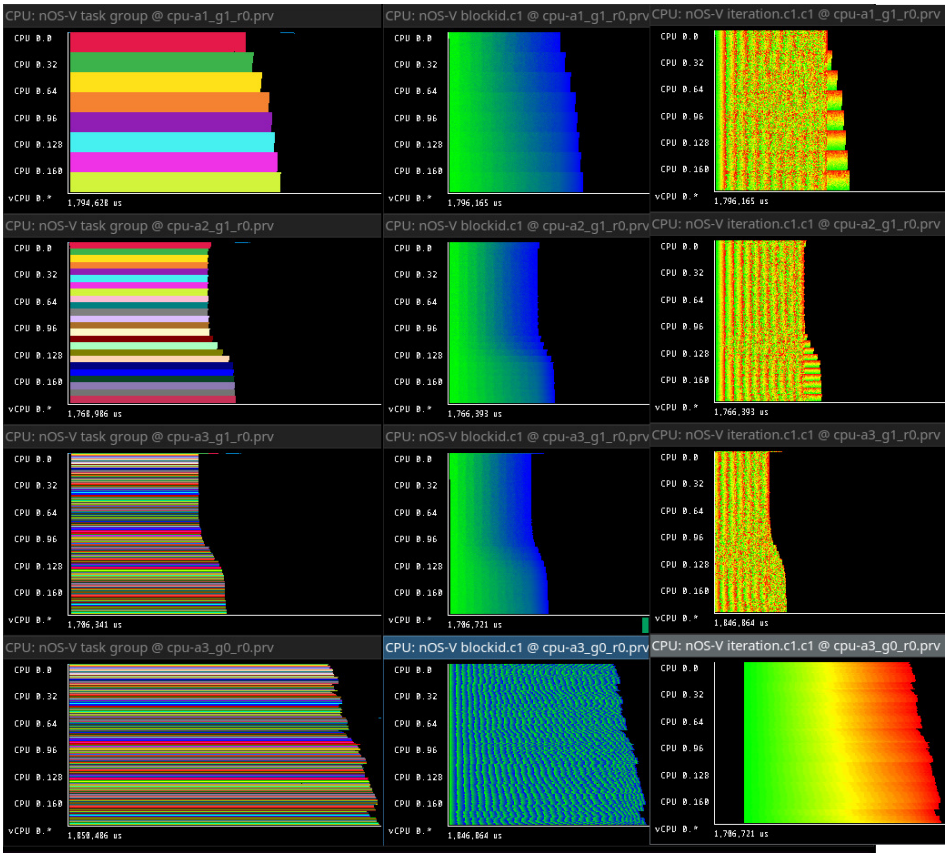



**Figure 4: Execution traces of Multisaxpy for the four scheduling configurations (one per row, top to bottom: Priority per NUMA, Priority per CCX, Priority per CORE, FIFO per CORE), each on the same run of the kernel. Within each row, the horizontal axis is wall time and the vertical axis lists the 192 worker cores; each colored segment represents one task executing on that core for the duration of the segment. The three columns visualize the same execution coloured by a different attribute: *(left)* taskgroup affinity domain assigned to the task, exposing the topology level at which affinity is enforced; *(middle)* task block identifier, where a depth-first colour pattern indicates that successor tasks on the same block reuse the core that touched it; *(right)* outer iteration identifier, with a smooth green-to-red transition indicating that consecutive iterations of the same block run on the same core (i.e., temporal locality is preserved across iterations).**

benchmark). The progression (1) → (2) → (3) isolates two distinct sources of speedup: NUMA-aware data placement at the application level (which increases memory system performance), and scheduling decisions at the runtime level. Comparing (3) against (2) therefore measures the marginal contribution of the scheduling framework alone, while (4) provides an independent reference point against widely used non-OmpSs-2 baselines.

We evaluate five HPC benchmarks: Cholesky, Heat, HPCCG, Matmul, and Multisaxpy. Each is evaluated at two problem sizes and three task granularities, and statistics are reported from a minimum of 10 executions per configuration. Small problem sizes execute on a single socket; large problem sizes use both sockets. Task granularities are selected on the *OmpSs-2 unmodified* version to avoid biasing the comparison against a baseline-unfavorable choice: we identify the optimal block size and add two additional granularities, one larger and one smaller, both within 50% of the optimum. The same three granularities are then applied identically across all four versions. The only exception is Multisaxpy, where the optimal block size already yields approximately one task per core; in that case the two additional granularities are both smaller than the optimum, since further coarsening would reduce the task count below the number of cores and limit parallelism.

The *AMD-Math/LLVM OpenMP* baseline differs across benchmarks: AMD-Libflame's POTRF for Cholesky, AMD-BLIS's DGEMM for Matmul, OpenMP tasks for Heat (the fork-join version performs worse), and OpenMP fork-join with thread binding for HPCCG and Multisaxpy (`OMP_PROC_BIND=1`). Fork-join was preferred over tasks for HPCCG and Multisaxpy because LLVM OpenMP currently lacks task-affinity support, which the fork-join version compensates for via static thread binding. To ensure a fair comparison, all baseline versions use `libnuma` with interleaved memory allocation, matching the NUMA distribution achieved by parallel initialization in versions (2) and (3). For Heat we use the OpenMP block sizes that empirically minimize wall time: 768 for the small problem and 512 for the large problem.

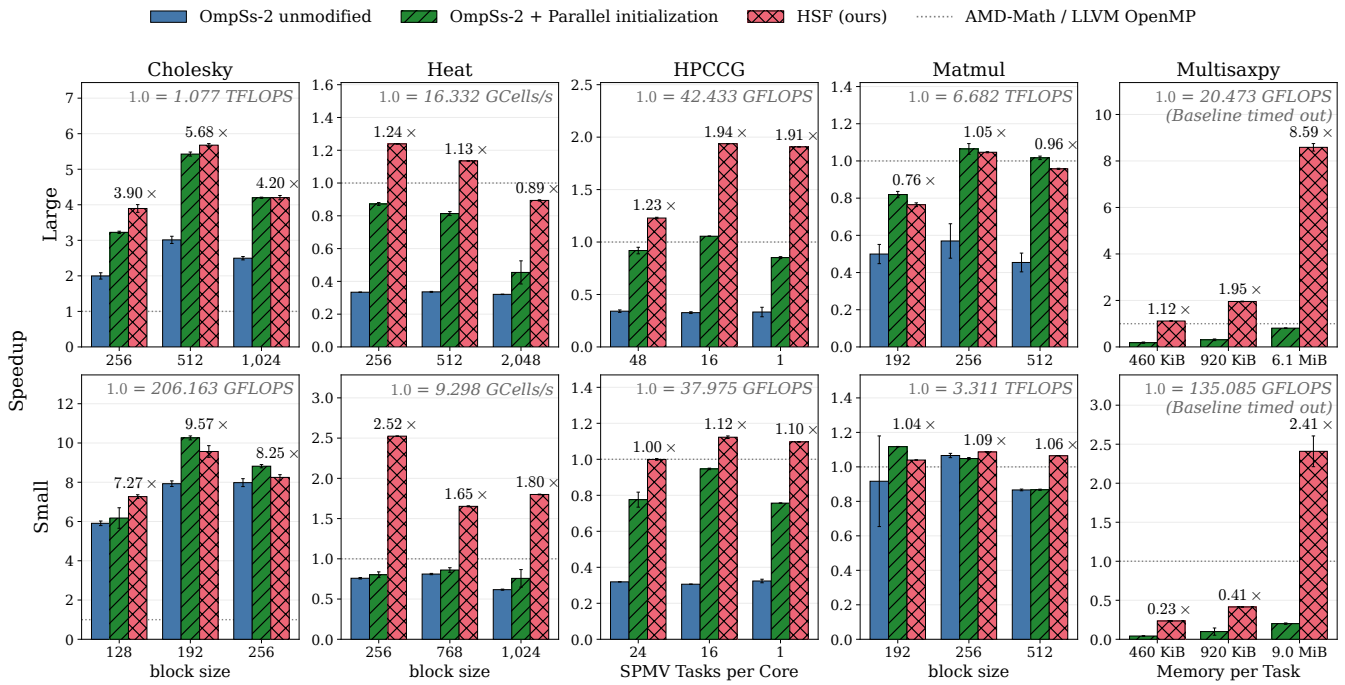



**Figure 5: Speedup of the five benchmarks (small and large problem sizes) across three block sizes, normalized to the *AMD-Math/LLVM OpenMP* baseline. All bars are normalized to the best execution on all granularities per size. The speedup of *HSF* over the best *AMD-Math/LLVM OpenMP* configuration is annotated above each bar.**

The five configurations below were not derived independently. They follow a short recipe, which we state here so that the per-benchmark descriptions read as instances of it rather than as isolated tuning results:

(1) **Group by reuse.** Create one taskgroup per set of tasks that touch the same data.
(2) **Bind each group to a topology level**, choosing the level whose cache holds the group's instantaneous working set.
(3) **Choose affinity strength from the working-set-to-cache ratio.** Use strict affinity when the working set fits and remote traffic dominates, and flexible affinity when load balance dominates. Our Cholesky and Heat configurations differ between the small and large problems on exactly this criterion.
(4) **Choose the intra-group policy from the dependence structure.** Use priority when a successor relation or a critical path exists, as in Multisaxpy and Cholesky.
(5) **Separate pools per kernel class** when phases differ in their access patterns, as in HPCCG.

Applying this recipe requires no program restructuring: taskgroup assignment is a clause on the task pragmas the algorithm already has. We now describe the resulting configuration for each benchmark. In all cases, taskgroups are organized to align ready work with topology domains so that the scheduler can preserve locality without sacrificing load balance.

**Cholesky.** The configuration combines critical-path priorities (giving precedence to tasks that unblock the longest dependency chain) with column-wise affinity, where tasks operating on the same matrix column share a taskgroup and taskgroups are distributed round-robin across topology domains. The two problem sizes use different affinity strengths: flexible core-preferred / NUMA-acceptable affinity for the small problem, where the working set fits in cache and load balance dominates; and strict NUMA affinity for the large problem, where reducing remote-memory traffic outweighs the cost of restricted scheduling.

**Heat.** Each topology domain is assigned one taskgroup. Blocks are mapped to taskgroups by row index in a round-robin pattern, so each domain owns a strided subset of rows that its CPUs repeatedly access across iterations. Strict CCD-level affinity is used for small matrices, maximising L3 reuse within a CCX; for large matrices, flexible CCD-preferred / NUMA-acceptable affinity preserves CCD locality while permitting load balancing within the same NUMA node when CCDs become saturated.

**HPCCG.** Two separate taskgroup pools are used, one per kernel class. The vector operations (waxpby, ddot) use a pool in which rows are partitioned into contiguous blocks assigned linearly across topology domains, so each domain owns a contiguous row range matching the first-touch NUMA placement established during matrix initialization. The sparse matrix-vector product uses a separate pool with FIFO ordering (to preserve row-traversal order and maximise sequential L3 reuse) combined with strict CCD-level affinity.

**Matmul.** HSF partitions the output tile space into a 2D grid of rectangular superblocks; each DGEMM task is assigned to the taskgroup of the superblock that owns its output tile, with the taskgroup pinned via strict affinity to a topology domain. Superblock dimensions are sized so that the instantaneous working set at any $k$-step fits within the L3 of the assigned domain.

**Multisaxpy.** The configuration combines topology-aware taskgroup affinity with intra-taskgroup priorities so that execution follows data locality rather than task creation order; the rationale and traces are given in detail in Section 4.2.

**Results.** Figure 5 reports per-benchmark performance: for each (benchmark, problem size, granularity) combination, bars are normalized to the best execution across all granularities for that size over the *AMD-Math/LLVM OpenMP* baseline, and show the standard deviation. The speedup of HSF over the best *AMD-Math/LLVM OpenMP* configuration is annotated above each bar. The geometric-mean view in Figure 1 compares HSF directly against *OmpSs-2 + parallel initialization*, thereby isolating the contribution of the scheduling framework from that of NUMA-aware initialization. HSF delivers consistent speedups across all benchmarks; in the worst cases its performance matches the best alternative rather than regressing, indicating that the scheduling abstractions impose no measurable cost when application-tuned scheduling does not help.

The two cases where HSF does not improve over *OmpSs-2 + parallel initialization*, Matmul and Cholesky, share a common cause: their wall time is dominated by AMD-BLIS DGEMM tasks, which are already tuned for AMD's cache hierarchy and amortize memory latency through internal blocking. Once first-touch NUMA placement supplies the right page distribution, the remaining performance margin attributable to scheduling is small. The other three benchmarks (Heat, HPCCG, and Multisaxpy) are more memory-bound and irregular, and consequently benefit substantially from HSF's ability to compose affinity with priority or FIFO ordering on a per-kernel basis.

## 4.4 Overhead study

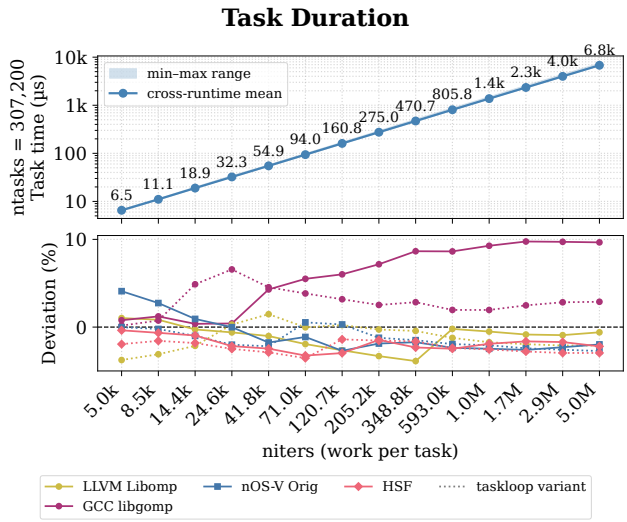



**Figure 6: Per-task execution time as a function of workload size (`niters`). *Top:* cross-runtime mean and the minimum and maximum across all runtimes. *Bottom:* per-runtime relative deviation of its own mean-of-medians from the cross-runtime mean.**

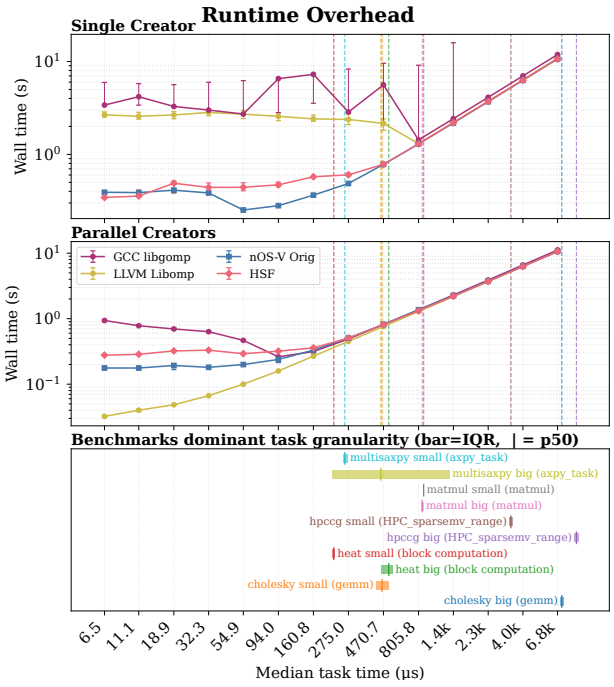



**Figure 7: Wall time of *SchedStress* (192 cores, `ntasks` = 307,200) depending on per-task workload, for two submission regimes: *Single Creator* (top) and *Parallel Creators* (middle). Markers and error bars are the median and IQR of wall time. Dashed vertical lines mark the median dominant-task duration of each HSF benchmark. The bottom panel shows the IQR of that duration across all task instances in one representative execution.**

HSF maintains more complex scheduling data structures and algorithms than the baseline nOS-V scheduler or the OpenMP runtimes (`libomp`, `libgomp`): a hierarchy of taskgroups, per-task affinity metadata, and propagated affinity-availability info up the tree. This added machinery is unavoidable cost on the task-creation and task-pop paths, so a fair claim of practicality requires us to (i) quantify the per-task overhead it introduces against simpler runtimes across a wide granularity range, and (ii) check whether that overhead is small enough to be invisible or at least manageable at the task durations used by real applications.

**Workload and methodology.** *SchedStress* dispatches a fixed number of independent tasks (`ntasks` = 307,200) on all 192 cores of our system. Each task computes a configurable number `niters` of terms of the Leibniz series for $\pi$, so that per-task work is pure floating-point arithmetic with no memory traffic that could confound runtime cost with cache behavior. We sweep `niters` across fourteen values and execute each `(runtime, niters)` configuration in 30 independent runs. The HSF configuration creates a taskgroup per topology domain at levels complex set and NUMA, for a total of 28 taskgroups. Tasks were added to complex sets taskgroups, which have as parent NUMA nodes, so that the scheduler has to traverse the taskgroup hierarchy for serving tasks.

**Task-time calibration.** Figure 6 maps each `niters` value to its corresponding per-task duration, spanning four orders of magnitude from ∼ 6 µs to ∼ 6.8 ms and bracketing the task granularities for the dominant tasks observed in the application benchmarks of Section 4.3. No runtime deviates by more than 5 % from the cross-runtime mean, except for GCC `libgomp` which deviates by up to 9.76 %. Hence, we conclude that the wall-time differences in Figure 7 reflect runtime overhead rather than workload variability.

**Two task-creation regimes.** Results for overhead comparison between runtimes are shown in fig. 7. The wall-time figure separates two distinct submission patterns. In the *Single Creator* regime, one thread issues every task. This is a common structure of task-based applications: the producer creates most tasks as it walks the iteration space, and the runtime resolves data-flow constraints before releasing tasks for execution. In the *Parallel Creators* regime, every worker generates a slice of the tasks using a `taskloop`-style construct. This is convenient when there are no inter-task dependencies (or only trivial reductions), restricting it to embarrassingly-parallel kernels.

For single-creator results, both nOS-V versions (including HSF) dominate both OpenMP runtimes throughout the single-creator sweep. `libgomp` serialises task creation through a global lock. `libomp`, by contrast, normally relies on local worker queues plus work stealing, but in this benchmark a single creator feeds all tasks, so the work-stealing path effectively funnels through shared synchronisation and behaves much like a global lock. nOS-V's delegation ticket lock avoids that bottleneck and therefore performs better. HSF remains at or near the bottom of the band next to the original nOS-V scheduler across the entire sweep; by ∼ 120 µs tasks its overhead is essentially the same as the OmpSs-2 baseline (*nOS-V Orig*), confirming that the complex scheduling state introduced by HSF imposes no measurable cost on real-world workloads.

When all 192 workers create tasks concurrently, through LLVM `omp_taskloop` a very low overhead is maintained due to its distributed scheduler implementation, where each worker manages its own task queue requiring no synchronization. While HSF's complex scheduling state benefits greatly from a centralized scheduler under a global lock, with very fine grained tasks the glbal lock becomes a bottleneck. The gap closes as task duration grows: by ∼ 160 µs the absolute wall-time gap is below 100 ms, and from ∼ 470 µs onward all runtimes converge to within a few percent of the work-only lower bound.

HSF's complex scheduling state does introduce a measurable overhead at fine granularities (a few microseconds per task), but this regime sits well below where real applications operate. The dashed colored vertical lines in Figure 7 mark the dominant task duration of each benchmark (p50 of the task type accounting for the largest share of total runtime), with the lower panel showing its inter-quartile range across all instances. All dominant tasks sit comfortably above the regime where scheduling cost is detectable; even on the smaller problem sizes the shortest dominant task (Heat at 0.22 ms) lives one order of magnitude past HSF's single-creator dominance zone and well past the parallel-creator crossover. At these realistic granularities, HSF's per-task overhead is amortised over tasks that run for hundreds to thousands of microseconds.

## 5 Related Work

Several task-based runtime systems expose multiple scheduling policies to cope with task diversity, particularly on heterogeneous platforms. StarPU provides a unified scheduling framework in which alternative queue organizations (e.g., FIFO, LIFO, priority) and scheduling heuristics coexist across CPUs and accelerators [6]. Within this design, HeteroPrio structures scheduling by maintaining separate queues for different task classes, allowing workers to prioritize tasks best matched to their device type while retaining stealing mechanisms for load balance [4, 7]. Subsequent extensions incorporate data-locality criteria into the multi-queue decision process [10]. While these systems highlight the interplay between affinity and imbalance, they do not support hierarchical composition or user-defined nesting of policies.

Closer to our notion of composable scheduling domains, Hugo et al. [12] extend StarPU with *Scheduling Contexts*: abstract sets of resources, each encapsulating its own scheduler, that allow several independent parallel libraries to coexist on the same heterogeneous machine with limited mutual interference. A hypervisor periodically resizes contexts using runtime feedback (resource utilization, computation progress) to redistribute CPUs and GPUs among co-running libraries. Their goal is therefore inter-application isolation through resource partitioning, with locality emerging implicitly from the partition. Our work targets a different problem: rather than isolating co-running libraries, HSF lets a *single* application express composable, nested scheduling domains over a shared resource pool, with affinity exposed as a first-class per-task attribute and policies (FIFO, LIFO, priority, affinity) configured declaratively per taskgroup rather than chosen by a hypervisor. The two approaches are complementary: scheduling contexts could in principle host multiple HSF taskgroup hierarchies, one per library.

Argobots [16] pursues a related goal at the threading layer: it exposes execution streams (one per core) bound to per-stream main schedulers, with ready work units residing in pools that can be private or shared between streams, and supports stacking custom schedulers as schedulable units inside those pools so that a higher-level scheduler can invoke a lower-level one and resume on completion. Argobots is, however, deliberately a mechanism, not policy framework: it ships only minimal default schedulers, and treats affinity as the implicit consequence of pushing a work unit to the appropriate pool. It is designed to represent work from the core's perspective, while HSF takes the opposite stance trying to represent work from the application's perspective allowing to compose application's scheduling declaratively through nested taskgroups whose hierarchy mirrors the application's execution pattern, and it makes affinity a first-class attribute.

A complementary line of work extends programming models and schedulers to improve locality and load balancing. Cilk [9] introduced work-stealing in their runtime for the latter purpose. OpenMP affinity constructs for tasks and taskgroups [13, 17] enable NUMA-aware placement and temporal locality strategies (e.g., co-locating tasks that reuse data). Topology-aware scheduling [11] organizes multiple queues per hardware level and performs topology-aware work stealing that prefers nearby victims. Qthreads adopts a hierarchical organization with one shared LIFO queue per NUMA node, promoting depth-first execution for cache reuse and limiting

expensive remote steals via NUMA-aware stealing policies [15]. More broadly, architecture-aware scheduling studies show that steering work toward cores or caches close to the data can substantially reduce memory access costs [14]. In contrast to approaches based on fixed queue hierarchies or heuristic locality rules, our design makes scheduling structure a first-class, composable abstraction: it combines affinity constraints and load balancing within a hierarchy of user-defined taskgroups, each with its own policy, and supports explicit, configurable stealing across topology boundaries. Since our model is built on basic scheduling policies as building blocks, it can be easily extended to support new policies.

## 6 Conclusions and Future Work

This paper presented HSF, an extensible scheduling framework built on two complementary abstractions: an extensible set of predefined policies (FIFO, LIFO, priority, and affinity) and the notion of taskgroups as explicit, nestable scheduling domains. Together they provide a structured yet flexible foundation for expressing scheduling decisions inside task-based runtime systems while keeping a clear separation between application logic and scheduling concerns. Predefined scheduling policies and taskgroups can be composed and manipulated programmatically to assemble complex strategies, including hierarchical ones that mirror the application's data decomposition or the machine's topology. Affinity in particular is exposed as a first-class mechanism with strict and flexible modes, giving applications an explicit knob to bias execution toward locality when it pays off and toward load balance when locality constraints become detrimental.

We integrated our framework into the OmpSs-2 tasking model on top of the nOS-V runtime system. Through various benchmarks and problem sizes, on a system with a complex memory hierarchy and large core count, HSF delivers a 1.84x speedup against an optimized baseline, while also indicating that the additional scheduling abstractions impose no measurable cost when application-tuned scheduling does not help. The benchmarks split cleanly along a memory-bound versus compute-bound axis: Heat, HPCCG, and Multisaxpy benefited substantially from composing affinity with priority or FIFO ordering on a per-kernel basis, while Cholesky and Matmul (whose wall time is dominated by AMD-BLIS DGEMM) saw a much smaller margin once first-touch placement supplied the right page distribution. A complementary overhead study showed that, in the single-creator regime that dependency-heavy applications naturally use, HSF outperforms LLVM `libomp` and GCC `libgomp` by 7.8× and 9.9× respectively at the smallest task granularity; in the parallel-creators regime it pays a measurable but bounded cost at very fine granularity that disappears beyond $\sim 470\,\mu s$, well below the dominant task duration of every benchmark in our suite.

Several directions extend this work. The main one is to broaden HSF's applicability: although its underlying mechanisms are not tied to a specific programming model, we currently demonstrate them only inside OmpSs-2, and porting HSF to OpenMP and integrating it with additional parallel constructs such as parallel loops and taskloops would let us validate the approach on a much wider class of applications. A second direction is to optimise HSF for finer-grained tasks: the same expressiveness that makes it effective on irregular memory-bound workloads also concentrates state behind a global lock, which becomes the bottleneck on extremely fine-grained workloads with multiple concurrent task producers; preserving composability while supporting these regimes is the main runtime-side optimisation target. Overall, HSF advances the state of the art by combining flexibility, programmability, and generality, with measured benefits on a representative set of HPC kernels and a quantified understanding of when and why those benefits appear.

## Acknowledgments

This project is co-funded by the Ministerio para la Transformación Digital y de la función pública, within the framework of the Plan de Recuperación Transformación y Resiliencia, and by the European Union – NextGenerationEU. This work was also supported by the ST4HPC research project (PID2023-147979NB-C21), funded by MICIU/AEI/10.13039/501100011033 and by FEDER, UE. Additionally, this paper is co-funded by the European Union under grant agreement 101290331 (TRU-STACK project). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Innovation Council and SMEs Executive Agency (EISMEA). The European Union, the European Commission, and the granting authority cannot be held responsible for them.

The evaluation was performed in systems partly financed by MCIN/AEI/10.13039/501100011033 and ESF/10.13039/501100004895 [grant number RYC2019-027592-I], and by the Grup de Recerca SGR 01007 - Programming Models group BSC-CNS/UPC [grant number SGR 01007].

## References

[1] Advanced Micro Devices (AMD). 2026. *BLIS*. AMD. https://github.com/amd/blis
[2] Advanced Micro Devices (AMD). 2026. *libFLAME*. AMD. https://github.com/amd/libflame
[3] Advanced Micro Devices (AMD). 2026. *AMD uProf*. AMD. https://www.amd.com/en/developer/uprof.html
[4] Emmanuel Agullo, Olivier Beaumont, Lionel Eyraud-Dubois, and Suraj Kumar. 2016. Are Static Schedules so Bad? A Case Study on Cholesky Factorization. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (Chicago, IL, USA, 2016-05). IEEE, 1021–1030. doi:10.1109/IPDPS.2016.90
[5] David Álvarez, Kevin Sala, Marcos Maroñas, Aleix Roca, and Vicenç Beltran. 2021. Advanced synchronization techniques for task-based runtime systems. In *PPoPP '21: 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Virtual Event, Republic of Korea, February 27- March 3, 2021*, Jaejin Lee and Erez Petrank (Eds.). ACM, 334–347. doi:10.1145/3437801.3441601
[6] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2009. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *Euro-Par 2009 Parallel Processing*, Henk Sips, Dick Epema, and Hai-Xiang Lin (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 863–874.
[7] Olivier Beaumont, Lionel Eyraud-Dubois, and Suraj Kumar. 2017. Approximation Proofs of a Fast and Efficient List Scheduling Algorithm for Task-Based Runtime Systems on Multicores and GPUs. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (Orlando, FL, USA, 2017-05). IEEE, 768–777. doi:10.1109/IPDPS.2017.71
[8] Ravi Bhargava and Kai Troester. 2024. AMD Next-Generation "Zen 4" Core and 4th Gen AMD EPYC Server CPUs. 44, 3 (2024), 8–17. doi:10.1109/MM.2024.3375070
[9] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1995. Cilk: An Efficient Multithreaded Runtime System. *SIGPLAN Not.* 30, 8 (aug 1995), 207–216. doi:10.1145/209937.209958
[10] Bérenger Bramas. 2019. Impact Study of Data Locality on Task-Based Applications through the Heteroprio Scheduler. 5 (2019), e190. doi:10.7717/peerj-cs.190
[11] Jérôme Clet-Ortega, Patrick Carribault, and Marc Pérache. 2014. Evaluation of OpenMP Task Scheduling Algorithms for Large NUMA Architectures. In *Euro-Par 2014 Parallel Processing*, Fernando Silva, Inês Dutra, and Vítor Santos Costa (Eds.). Springer International Publishing, Cham, 596–607.

[12] Andra-Ecaterina Hugo, Abdou Guermouche, Pierre-Andre Wacrenier, and Raymond Namyst. 2013. Composing Multiple StarPU Applications over Heterogeneous Machines: A Supervised Approach. (2013), 1050–1059. doi:10.1109/IPDPSW.2013.217

[13] Jannis Klinkenberg, Philipp Samfass, Christian Terboven, Alejandro Duran, Michael Klemm, Xavier Teruel, Sergi Mateo, Stephen L. Olivier, and Matthias S. Müller. 2018. Assessing Task-to-Data Affinity in the LLVM OpenMP Runtime. In *Evolving OpenMP for Evolving Architectures*, Bronis R. De Supinski, Pedro Valero-Lara, Xavier Martorell, Sergi Mateo Bellido, and Jesus Labarta (Eds.). Vol. 11128. Springer International Publishing, 236–251. doi:10.1007/978-3-319-98521-3_16

[14] Ananya Muddukrishna, Mats Brorsson, and Vladimir Vlassov. 2011. A Locality Approach to Architecture-aware Task-scheduling in OpenMP. (2011).

[15] Stephen L. Olivier, Allan K. Porterfield, Kyle B. Wheeler, and Jan F. Prins. 2011. Scheduling task parallelism on multi-socket multicore systems. In *Proceedings of the 1st International Workshop on Runtime and Operating Systems for Supercomputers* (Tucson, Arizona) *(ROSS '11)*. Association for Computing Machinery, New York, NY, USA, 49–56. doi:10.1145/1988796.1988804

[16] Sangmin Seo, Abdelhalim Amer, Pavan Balaji, Cyril Bordage, George Bosilca, Alex Brooks, Philip Carns, Adrián Castelló, Damien Genet, Thomas Herault, Shintaro Iwasaki, Prateek Jindal, Laxmikant V. Kalé, Sriram Krishnamoorthy, Jonathan Lifflander, Huiwei Lu, Esteban Meneses, Marc Snir, Yanhua Sun, Kenjiro Taura, and Pete Beckman. 2018. Argobots: A Lightweight Low-Level Threading and Tasking Framework. *IEEE Transactions on Parallel and Distributed Systems* 29, 3 (2018), 512–526. doi:10.1109/TPDS.2017.2766062

[17] Christian Terboven, Jonas Hahnfeld, Xavier Teruel, Sergi Mateo, Alejandro Duran, Michael Klemm, Stephen L. Olivier, and Bronis R. de Supinski. 2016. Approaches for Task Affinity in OpenMP. In *OpenMP: Memory, Devices, and Tasks* (2016). Springer, Cham, 102–115. doi:10.1007/978-3-319-45550-1_8

[18] Field G. Van Zee and Robert A. van de Geijn. 2015. BLIS: A Framework for Rapidly Instantiating BLAS Functionality. *ACM Trans. Math. Software* 41, 3 (June 2015), 14:1–14:33.

[19] David Álvarez, Kevin Sala, and Vicenç Beltran. 2024. nOS-V: Co-Executing HPC Applications Using System-Wide Task Scheduling. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 312–324. doi:10.1109/IPDPS57955.2024.00035