nOS-V: Co-Executing HPC Applications
Using System-Wide Task Scheduling

1st David Álvarez
Barcelona Supercomputing Center
Barcelona, Spain
david.alvarez@bsc.es

2nd Kevin Sala
Barcelona Supercomputing Center
Barcelona, Spain
kevin.sala@bsc.es

3rd Vicenç Beltran
Barcelona Supercomputing Center
Barcelona, Spain
vbeltran@bsc.es

Abstract—Future Exascale systems will feature massive parallelism, many-core processors and heterogeneous architectures. In this scenario, it is increasingly difficult for HPC applications to fully and efficiently utilize the resources in system nodes. Moreover, the increased parallelism exacerbates the effects of existing inefficiencies in current applications. Research has shown that co-scheduling applications to share system nodes instead of executing each application exclusively can increase resource utilization and efficiency. Nevertheless, the current oversubscription and co-location techniques to share nodes have several drawbacks which limit their applicability and make them very application-dependent.

This paper presents co-execution through system-wide scheduling. Co-execution is a novel fine-grained technique to execute multiple HPC applications simultaneously on the same node, outperforming current state-of-the-art approaches. We implement this technique in nOS-V, a lightweight tasking library that supports co-execution through system-wide task scheduling. Moreover, nOS-V can be easily integrated with existing programming models, requiring no changes to user applications. We showcase how co-execution with nOS-V significantly reduces schedule makespan for several applications on different scenarios, outperforming prior node-sharing techniques.

Index Terms—HPC, parallel programming, co-location, co-execution, task-based programming

I. INTRODUCTION

Traditionally, HPC clusters have been partitioned into subsets of nodes that execute applications exclusively, without being shared for the duration of the application’s execution to maximize performance. This differs from other workloads like cloud computing, which strive to share single cluster nodes with as many users as possible to maximize throughput.

The exclusive execution model is efficient as long as HPC applications can fully exploit all their allocated nodes. However, typical HPC workloads can display various issues that prevent them from efficiently using a whole node. For example, most applications have serial initialization, finalization or even communication phases, where there is limited parallelism. Fork-join or distributed applications can display load imbalance due to an ineffective distribution of available work. Some applications even lack parallelism when scaling to nodes with hundreds of cores. Additionally, with the rise of heterogeneous systems in HPC, some applications cannot overlap accelerated and non-accelerated computations, leaving resources idle.

Moreover, the end of Dennard scaling and the rise of many-core systems and heterogeneous machines exacerbate the aforementioned issues. Exascale systems not only have to worry about scalability and resilience problems but also the decaying node-level efficiency hindering total cluster-level efficiency.

Co-scheduling applications on the same node has been proposed as a way to mitigate the problems mentioned above. These techniques modify cluster schedulers (e.g. SLURM [1]) to identify applications to co-schedule and execute simultaneously on the same nodes. The main goal is to increase available parallelism by running more than one application, filling the parallelism gaps in one application with work from another. However, resources must be shared fairly and efficiently amongst applications.

HPC applications make use of runtime systems that divide their workloads into discrete work units (tasks or threads), which can be executed in parallel. Runtimes assume they are running exclusively in each node, and they schedule tasks over the system’s resources. These presumptions of exclusivity over the node’s resources and sparse preemptions by the OS allow runtimes to apply aggressive optimizations such as busy waiting or custom synchronization techniques. In practice, this causes that system noise preempting runtime threads can significantly influence application performance.

Therefore, while in non-HPC workloads we could simply oversubscribe the applications on the same node and let the underlying Operating System (OS) perform preemptive time-sharing, this approach does not work well in the case of HPC applications. Due to their reliance on exclusive access, HPC applications can have strong interferences when executed simultaneously [2]–[6]. This can result in co-scheduled workloads displaying unpredictable behavior and significant slowdowns. Two well-known problems contribute to this behavior. First, a large number of created threads can cause a scalability collapse scenario [7], [8]. Second, the OS preempting threads that rely on user-space spinlocks can cause the widely-known Lock Holder Preemption and Lock Waiter Preemption problems [9]–[11].

For HPC workloads, static co-location [12]–[15] has been proposed to reduce application interference. With static co-location, system resources are statically partitioned, and each application runs confined in a specific partition. This approach
simulates each application running exclusively on a subset of the node. However, this fails to adapt to the dynamic nature of HPC applications, which display different amounts of parallelism at different points in time.

We propose a new technique called application co-execution, which can improve overall node efficiency by leveraging the instant parallelism of all co-scheduled applications. With co-execution, we maintain a global view of the available tasks from all applications running in the system. This is more similar to the OS’s view, but work can be scheduled in a suitable granularity for HPC applications, preventing oversubscription problems. Moreover, scheduling is cooperative between the different co-executed applications, instead of preemptive.

We implement this approach through the novel nOS-V tasking library. nOS-V is a portable library that leverages widely available inter-process communication (IPC) facilities and requires no kernel modifications. When using nOS-V for co-execution, users can execute several HPC applications simultaneously in the same node. However, a single library (nOS-V) manages system resources, instead of one per application. Additionally, nOS-V supports global scheduling policies based on process priority or data locality.

Specifically, our contributions in this work are:

1) We present the technique of application co-execution for HPC workloads
2) We describe nOS-V, a novel tasking library that enables application co-execution and
3) We evaluate in detail the performance of nOS-V integrated with the LLVM OpenMP runtime and the OmpSs-2 programming model and show how it outperforms other node-level resource sharing techniques such as oversubscription, static co-location and dynamic co-location.

II. BACKGROUND

Several solutions to tackle and improve node-level efficiency and increase job throughput in HPC clusters have been studied for a while.

The simplest solution is known as oversubscription, which is shown in Figure 1. Although oversubscription is extensively used for Cloud workloads [2], the use of busy-waiting policies [16] and user-space locking [17] can cause significant slowdowns on co-scheduled HPC applications [3]–[6]. In essence, the slowdowns are caused by the OS pre-empting HPC runtimes while holding locks, busy waiting or performing critical communication.

Modern approaches are usually based on application co-location, shown in the second place of Figure 1. Co-location executes independent applications in the same node by splitting system resources at the level of core or socket and assigning each application to an exclusive set of resources. Co-location can work well if the applications to run concurrently in the same node are chosen carefully and the partitions are correctly sized. However, finding the ideal partition for a set of applications may not be trivial or even feasible, as many applications feature dynamic parallelism in distinct phases.

Dynamic co-location techniques extend static co-location with dynamic resource sharing between partitions [18], [19]. An example of dynamic co-location is DLB [19], which leverages malleability in the application’s programming models and acts as a broker that oversees how applications lend and reclaim cores to each other, as shown third in Figure 1. Essentially, dynamic co-location improves over traditional co-location by changing the static partition between processes at some points during runtime.

However, dynamic co-location does not have a global view of the available work on the node and only oversees how applications voluntarily yield resources. Thus, dynamic co-location cannot take informed node-wide scheduling decisions. Moreover, like static co-location, there is a limit on the number of applications that can be executed simultaneously, as the initial resource partitions need to have at least one core.

All approaches to improve node-level efficiency are translated into cluster-level improvements by fitting batch schedulers like SLURM [1] with co-scheduling capabilities that consider how node-level resources are shared [12]–[14], [20]–[22]. The co-scheduling problem is related to node-level efficiency: better alternatives to in-node resource sharing can be combined with existing scheduling policies to deliver better cluster-level efficiency.

In this work, we focus on HPC applications, which rely on programming models such as OpenMP [23] and OmpSs-2 [24]. Those programming models are, in turn, based on lower-level threading or tasking libraries [25], [26]. These tasking libraries provide lightweight parallel work units referred to as user-level threads, tasks or tasklets.

Basing programming model’s runtimes upon these tasking libraries provides several benefits over the traditional POSIX thread interface [27]. User-level threads are lighter weight and can be complemented with customizable user-space schedulers that implement custom policies. Several available state-of-the-art tasking libraries implement ULTs, including Argobots [28], Qthreads [29] and massiveThreads [30].

Usually, a tasking library assumes exclusive access to the entire node and executes tasks in every core, regardless of any other processes running in the system. Execution is generally done by creating a pool of threads pinned to specific cores and scheduling tasks as if they were being directly mapped to physical cores. This ultimately allows implementing scheduling policies which maximize data locality.

Co-execution is the main contribution of this paper, where we execute several applications simultaneously by sharing the same threading or tasking library instance, as opposed to performing oversubscription or co-location. This way, a single scheduler is shared by multiple applications, while maintaining the property of having only one Pthread running per system core at any time.

The novel nOS-V tasking library implements co-execution by sharing a single scheduler located in a shared memory region containing tasks from multiple processes and, if en-
abled, from different users. The fourth diagram of Figure 1 illustrates this concept by showing two applications with nOS-V enabled tasking runtimes, which use nOS-V to share system resources. Resources are not statically partitioned into sets, and applications’ tasks can still be executed on any core. Moreover, the application runtimes may be different, as long as they are integrated with nOS-V. By sharing the tasks to be scheduled and applying a node-wide scheduling policy, we can guarantee that there is no oversubscription because there is no scenario where two tasks are concurrently scheduled to the same core.

Note that this is similar to the operating system’s scheduling but done on a task granularity instead of a thread granularity and cooperatively instead of preemptively, allowing applications to not suffer from the drawbacks of either oversubscription or co-location. Additionally, as this shared scheduler lives in user-space, it can be fitted with customized scheduling policies.

III. THE nOS-V TASKING LIBRARY

nOS-V is a new tasking library that provides a simple interface to create and manage tasks, similar to other user-level thread libraries. The key feature of nOS-V is that it enables the co-execution of different applications on a node through a novel inter-process tasking mechanism. Unlike other tasking libraries that implement context-switching from user-space, our library leverages Pthreads to context-switch threads that belong to different applications.

Like other user-level thread libraries, nOS-V provides an abstraction layer based on tasks that hides the complexities of many-core management, thread bindings and cooperative task scheduling. However, in nOS-V, there is only one instance of the library managing all the cores in the node. This abstraction layer can replace the low-level Pthreads [27] API in applications and runtimes. In this way, tasks from different applications can be transparently co-executed.

nOS-V manages pools of Pthreads located in shared memory for each process being co-executed. These threads are pinned to specific cores using sched_setaffinity, and allow nOS-V to execute tasks on specific cores by running them on these threads. When a task finishes, its Pthread is re-used to execute other tasks. However, if a task blocks, its associated Pthread is blocked with it. This allows nOS-V tasks support any feature supported by a regular Pthread, such as Thread Local Storage (TLS), unlike tasking libraries based on user-level context switching. The trade-off of using Pthreads that remain attached to tasks is a higher context-switch cost only when a task blocks or yields. However, this extra overhead is usually negligible [31], and it ultimately enables inter-process tasking.

A. Architecture

Figure 2 displays the architecture of nOS-V, which will be referenced throughout this section. The main difference between nOS-V and other tasking libraries is that most of nOS-V’s components are allocated on a POSIX Shared Memory [27] segment, which is then mapped onto all processes in the node using the library. Shared memory is then the primary IPC mechanism used throughout our tasking library. It has a very low overhead, does not require system calls to
communicate between processes, and allows re-using regular data structures without having to design ad-hoc alternatives.

As shown in Figure 2, we allocate the following data structures in the nOS-V's shared memory segment:

- A thread manager for every nOS-V enabled process running.
- A CPU manager that stores the current mapping of cores to threads.
- The task descriptors.
- A centralized scheduler based on a Delegation Ticket Lock [32].

In the example of Figure 2, three applications, each in a different process and using a different programming model, are running simultaneously. The nOS-V library will manage thread pools for each application, which can submit tasks for execution. Submitted tasks will be placed in a shared scheduler (see Section III-D), containing work from multiple applications. Later on, when the nOS-V CPU manager has an idle core not running any task, it will grab a ready task from the scheduler and execute it on an available POSIX thread from the task’s creator process on that idle core.

B. Tasking Interface

The main API of nOS-V contains five basic operations to handle tasks coming from multiple processes: nosv_create, nosv_submit, nosv_pause, nosv_attach and nosv_destroy.

The nosv_create operation provides to the caller a task descriptor on which all the relevant information to the task can be stored. The three main items that this task descriptor contains are: the PID of the process on which the task was created, a callback to run the task, and a callback to notify when the task has finished its execution. nOS-V users can also embed a metadata pointer in this task descriptor to implement task arguments or data dependency information.

The nosv_submit operation inserts a previously created task in the shared scheduler and marks it as ready for execution. The task will then remain in the scheduler until there is an idle core and the current scheduling policy decides to execute this task. The task is executed by calling the run callback in the descriptor in a thread that belongs to the same PID as the task descriptor.

The nosv_pause operation blocks a currently running task. This call can be used to yield the current core if a task needs to perform a blocking operation or wait for an event. The operation must be called from the blocked task’s context, as in a yield operation. Once a task is paused, its associated Pthread is blocked and remains attached to the task. The core where this task was running will be marked as idle, and it will become available to run other ready tasks from the scheduler. Unblocking a task is done by calling the nosv_submit operation again, which will insert it back into the scheduler.

The nosv_attach operation (with its corresponding detach counterpart) transforms a currently running thread into a nOS-V task. Upon returning from the attach operation, the calling thread has an associated nOS-V task which can be blocked and scheduled by the library. This operation is designed to ease porting existing programming models to nOS-V.

The nosv_destroy operation returns a task descriptor to nOS-V and frees the associated shared memory. This operation has to be called after the task has finished its execution.

Figure 2 showcases how these operations behave in the overall nOS-V architecture. The first application performs a submit operation, which inserts the task into the shared scheduler. Meanwhile, the second and third applications create and destroy tasks, thus interacting with the task descriptor allocator.

C. nOS-V Life cycle

Whenever an application linked with nOS-V is executed, the library checks during startup for the existence of a specific POSIX shared memory segment [27] on the node and initializes the segment if it does not exist. This shared memory is then mapped into the process’ address space and will be used to communicate with other nOS-V enabled processes.

The first process registered into this shared memory region spawns a new thread for each core in the node, ready to execute available work. Each spawned thread is pinned to a specific idle core and will ask the scheduler for work until a task is obtained or the thread is shut down. If the obtained task belongs to a different process (the PID in the descriptor is different), the current thread is suspended, and a thread belonging to the ready task’s process is obtained from its thread pool. The selected thread is then pinned to the current core and woken up with the assigned task. Note that tasks are always executed in a thread belonging to the process that created them.

When a task is paused through nosv_pause, the Pthread currently executing the task is blocked in a condition variable. Then, another PThread (spawned or recycled from an idle thread pool) is resumed on the current core and executes other ready tasks. When the paused task becomes ready again, it is put back into the scheduler’s ready queue, so eventually, it will resume its execution. When a worker thread gets a ready task from the scheduler, it checks, before executing it, if the task already has an attached thread. If so, the current worker thread is blocked and added to a pool of idle threads. Then, the worker thread attached to the task is unblocked, and the task can finally resume its execution in the current core. Note that the woken-up thread’s affinity is set before unblocking it, effectively performing a context switch between both threads.

Finally, a nOS-V application finishes when all the tasks have been executed. When this happens, the process is unregistered from the shared memory structures, and the last process to unregister will delete the whole shared memory segment.

D. Shared Scheduler

The shared scheduler implementation we have used is a centralized scheduler based on a Delegation Ticket Lock (DTLock) [32]. We chose this design because, unlike work-stealing, we can implement node-wide scheduling policies...
with a consistent view of all the tasks in the node while maintaining state-of-the-art performance.

This centralized scheduler is designed with a delegation mechanism, where a single thread enters the scheduler’s critical region and serves the other waiting threads and then itself. Task insertion is managed through lock-free submission queues, eliminating the need for locking when submitting tasks into the scheduler. With this model, the single thread entering the critical region has a global view of every ready task in the node and can make informed scheduling decisions. Inside the scheduler, tasks are organized in data structures according to their affinity and priority, which mandates where and when a task is executed.

To enforce fairness in time distribution between co-executed programs, the scheduler has a round-robin mechanism based on a time quantum. While the quantum is not over, the scheduler prioritizes assigning tasks with the same PID as the waiting thread, minimizing context switches. When the time quantum runs out, the scheduler will assign tasks from a different PID, forcing the context switch and distributing execution time fairly between processes. This quantum is user-configurable, and the best results are achieved when the quantum is a multiple of the typical task execution times of the co-executed programs.

There are three built-in scheduling policies in nOS-V, and more can be implemented by modifying the scheduling algorithm. First, there are user-configurable priorities per-application and per-task. Per-application priorities are similar to traditional process priorities in operating systems. Per-task priorities allow users to specify which tasks they want to be executed first, and can be used to prioritize communication or critical tasks, for example. Finally, users can use locality-based scheduling by specifying a specific core or NUMA node where a task will be executed. This configurable affinity can be either preferred (best-effort) or strict. This locality-based scheduling policy allows nOS-V applications to adapt to NUMA systems.

E. Memory Management

Memory allocation is a performance-critical part of nOS-V and requires efficiently managing the POSIX shared memory region and fast allocation of shared data structures.

Although many state-of-the-art and user-space oriented scalable memory allocators exist [33], some of them rely on mmap-like semantics or use static per-process metadata, making them unsuitable to use by nOS-V. We settled on a custom memory allocator that splits the shared memory region in chunks and uses a similar approach to the well-known SLAB allocator [34], [35], together with per-cpu chunk caches, to provide scalable memory allocation. Our implementation proved to be competitive with other memory allocators and is much better suited for nOS-V, as it allows to free a pointer allocated by a different process because the allocator’s metadata resides in the shared memory.

F. Threat Model and Security

nOS-V is built on a threat model that assumes co-executed applications are trusted, and hence are not ill-intentioned. This model is analogous to other libraries which leverage shared memory across applications, such as DLB, Arachne and NVIDIA’s CUDA MPS [36]. However, this trust does not imply that applications are bug-free, only that there are no malicious agents.

Reliability is a critical factor when designing HPC software, even when applications are trusted. Programs are expected to operate while failures are the norm rather than an exception, and it is expected that nOS-V can recover when one or more of the co-executed applications encounter a bug, without corrupting state or crashing other unrelated co-executed programs.

This was the main reason for sharing only one memory segment over alternatives that share the full address space of processes such as Process-in-Process (PiP) [37]. Additionally, we intend to explore the use of hardware features that limit the impact in nOS-V of bugs in user applications. For example, Intel Memory Protection Keys (MPK) [38] or Pointer Authentication on ARMv8.3-A [39], [40].

With nOS-V, it is technically possible to co-execute applications belonging to different users. In HPC settings, we imagine this feature to be used between users from the same group, where users trust each other and may share compute-hour quotas. By default, this feature is disabled, and nOS-V instances may only co-execute applications from the same user. However, it can be activated through a configurable option, and then different users can co-execute applications using the same nOS-V instance.

IV. Adapting HPC Runtimes to Leverage nOS-V

In the previous sections, we have defined co-execution, and we have presented the nOS-V tasking library that allows co-executing tasks from multiple applications in one node. The library’s main objective is to be integrated into tasking runtime systems that support parallel programming models, such as OpenMP [23] and OmpSs-2 [24]. However, the simplicity of the nOS-V library API would allow using it directly from user applications if required.

Traditional runtimes supporting dynamic co-location work like depicted in Figure 3. Two different user applications will
have two different runtime instances when running in a single node. These runtimes will have independent schedulers and CPU Managers. This latter is the component that tracks the available computing resources and manages their occupancy. To support dynamic co-location, both runtimes must interact with a Dynamic Co-location Library such as DLB, which acts as a broker. In this modality, runtimes must support core hot-swapping to add and remove computing resources dynamically.

When adapting a runtime to use nOS-V, there is no need for a scheduler or a CPU manager, as the tasking library provides those components. However, it is still possible to have a hybrid approach where the original runtime retains control over scheduling, working with nOS-V in a two-level strategy. Both approaches can coexist, and runtimes delegating all of their scheduling can be co-executed with runtimes using two-level scheduling as shown in Figure 4.

For the experimental evaluation of this paper, we have integrated the nOS-V library into two programming models: OpenMP and OmpSs-2. For OpenMP [23] we adapted the LLVM OpenMP runtime applying a conservative strategy consisting in using nosv_attach to transform worker threads into nOS-V tasks. Then, we kept LLVM’s work-stealing scheduler, which works in a two-level approach with nOS-V scheduling. This way, OpenMP workers for each parallel region are turned into nOS-V tasks, supporting both fork-join and task OpenMP applications.

We took a different approach for the OmpSs-2 programming model, which is a data-flow task-based programming model, similar to OpenMP tasking, but with some additional features [32], [41]–[43]. In this case, we used its Nanos6 [44] reference runtime implementation and removed the scheduling and CPU managing subsystems completely, delegating every scheduling decision to nOS-V.

Both approaches can be applied to other programming models, and the preferred approach depends on the specific implementation for each runtime. Since OpenMP features certain scheduling restrictions (task scheduling constraints, untied tasks, etc.), keeping the original scheduling algorithm was the preferred option. On the other hand, since OmpSs-2 has a more flexible scheduling policy, it could be completely delegated to nOS-V.

V. EXPERIMENTAL EVALUATION

This section shows how nOS-V minimizes the interference between applications and improves overall resource utilization for several co-location scenarios.

For all experiments, we use the Nanos6 and the LLVM OpenMP runtimes modified to run on top of nOS-V. To ensure the modified runtimes are suitable baselines, we compare in the first experiment the modified Nanos6 runtime, which underwent the largest changes, against an unmodified Nanos6 runtime which has state-of-the-art performance for task-based applications [32]. The goal is to check whether using nOS-V introduces any additional overhead when not using the co-execution capabilities, ensuring that the changes to task allocation, scheduling or thread managing do not introduce any performance penalty.

In the second experiment we take a set of 7 HPC applications and prepare for each one a version implemented in OmpSs-2. OpenMP tasks and OpenMP fork-join, having thus a set of 21 variants to test with. Then, we execute each one of the 231 pairwise variant combinations and measure the speedup against executing one application after the other using the full node. We compare the speedup obtained using nOS-V as a co-execution agent versus using co-location (each application gets half the node), dynamic co-location with DLB and oversubscription.

In the third experiment, we scale the co-execution experiments beyond pairwise combinations, co-executing sets of 3, 4, 5 and 6 applications in the same node, comparing the speedup achieved versus exclusive execution.

Finally, we perform a fourth experiment where we evaluate pairwise combinations of the same 7 benchmarks using distributed MPI+X versions with 8 and 32 nodes. The goal of this experiment is to show how the improvements in node-level co-execution can be translated to distributed environments by sharing nodes between MPI+X applications.

The first three experiments were conducted on an AMD Rome cluster where each node has a single AMD EPYC 7742 64-core processor with SMT turned off and 1TiB of main memory. The software stack is based on Rocky Linux 8.5 with a 4.18 kernel, Intel MKL 2021.1.1 and GCC 10.2.0.

The last experiment with distributed applications was conducted on a dual-socket Intel Xeon Platinum 8160 24-core cluster, with a total of 48 cores per node and an Intel OmniPath 100Gbit/s interconnect. The software stack is based on SLES 12 with a 4.4 kernel, Intel MPI 2018.4, Intel MKL 2021.4 and GCC 11.2.0.

The process quantum for nOS-V is configured at 20ms for all experiments.

A. nOS-V Baseline Performance

The baseline experiment uses a benchmark set [32] that includes a matrix multiplication, a vector dot-product, a Gauss-

Fig. 4: Diagram of two applications co-executed using nOS-V.
Seidel heat equation simulation, the HPCCG proxy application, an N-Body simulation, a Cholesky factorization, and the Lulesh 2.0 [45] proxy application.

For each benchmark, we measure the performance on two points: one with an ideal task granularity which results in peak performance for the application, and another where the task granularity is too small, and thus its performance is bound by the overhead introduced by the task-based runtime, including task creation, allocation and scheduling. The point chosen for the small granularity is at around 50% of the peak benchmark performance. With these two points, we can verify any significant performance difference in peak performance or high runtime overhead scenarios between the original Nanos6 and our adapted Nanos6+nOS-V version.

Figure 5 shows the results of the baseline experiment, which are an average of 10 executions using a full node exclusively. As we can see, there is no relevant speedup or slowdown of the version of Nanos6 with nOS-V when running a single application aside from minor variances for small granularities. This experiment confirms that despite using the shared memory regions and adapting our memory allocation to use shared task descriptors, we have not introduced any relevant performance penalties.

B. Co-execution Performance

We can now proceed to compare the performance of co-execution versus other node-sharing techniques. For this experiment, we want to check whether nOS-V can successfully overlap the execution of two HPC applications. Using a set of 21 variants belonging to 7 different HPC applications, we tuned the parameters of each application to obtain two different behaviours:

1) A high resource usage behaviour, when an application has enough parallelism saturate one or more resources (cpu or memory bandwidth) of the machine.

2) A low resource usage behaviour, when an application has constrained parallelism and thus cannot saturate any resource in the machine, causing resources to be wasted.

Note that even in high resource usage behaviour, applications may have sequential initialization phases, reduced parallelism stages, or memory-bound parts where co-location or co-execution could improve overall efficiency.

We tested the following co-execution strategies for every pairwise application combination:

1) One application after the other, using the whole node (exclusive execution).

2) Both applications simultaneously on the whole node, letting the OS manage time-sharing. Both the Nanos6 runtime and the LLVM OpenMP runtime use default parameters to block threads when idle. (oversubscription).

3) Each application on an equal node slice, statically partitioned (co-location).

4) Each application on an equal node slice using the Dynamic Load Balancing library at runtime (dll). DLB is only compared for OmpSs-2 applications, since for OpenMP DLB can only balance at the beginning of parallel regions.

5) Using nOS-V to execute all applications simultaneously (co-execution).

Furthermore, to comprehend the exact characteristics under which each co-execution strategy performs better, we studied the obtained speedups when executing pairs of applications with high resource usage, with low resource usage, and a high resource usage with a low resource usage application.

For every result, we calculate the speedup comparing the makespan of executing two applications simultaneously using one of the evaluated co-execution techniques against the makespan of executing both applications exclusively using the whole node.

Results are first presented in Figure 6 in a summarized violin plot. Subfigures 6a, 6b and 6c show the violin plots for combinations of high resource applications, high-low resource applications, and low-resource applications respectively. Results for dynamic co-location with DLB are shown instead on Figure 7, which includes only OmpSs-2 application variants, since those were the only ones evaluated with DLB.

For both violin plots which include high resource applications, nOS-V is the best performing co-execution strategy, delivering a median speedup of 1.21x with practically no slowdowns. Both co-location and oversubscription deliver speedups compared to exclusive execution, but show slowdowns in particular application combinations. On combinations of low resource usage applications, co-location slightly outperforms nOS-V, although in both cases have a similar median speedup.

A more detailed overview of the speedup of each application combination for each co-execution methodology is available in the Heatmaps displayed in Figures 8, 9 and 10. For each heatmap, the speedup versus the exclusive execution of that particular application combination is shown in the intersection between both numbers, which correspond to the applications
Fig. 6: Average performance achieved on pairwise combinations for each strategy

Fig. 7: Average performance achieved on pairwise combinations for each strategy, including only OmpSs-2 application variants

Fig. 8: Heatmap for pairwise speedup vs exclusive execution for combinations of high resource usage applications

listed in Table I. Higher speedups tend to lighter yellows while lower speedups are represented as darker greens. Inside each square we show the rounded speedup. Squares that have speedups in white text represent values lower than 1.

In general, the challenge in the co-execution field for HPC has been to find methods which allow to co-locate high resource usage applications, which are the most common. Since most HPC applications are either compute-bound or memory-bound, and do not necessarily block often like some datacenter use-cases, existing methods can cause slowdowns which would hinder the overall system efficiency.

What we see in the violin plot for high resource usage application combinations in Figure 6a, as well as the detailed heatmaps presented in Figure 8 is that executing multiple high resource usage applications in an HPC machine by oversubscribing yields very variable results, with some combinations achieving 1.8x speedups and others up to a 0.6x slowdown. In contrast, co-locating the applications by partitioning the nodes delivers better results, although still has several combinations where slowdowns are present. Finally, when we co-locate the OpenMP and OmpSs-2 applications using the nOS-V library, we find that we achieve speedups for most application combinations, while only slowing down slightly two combinations involving the OpenMP matmul benchmarks.

When we combine low resource usage with high resource usage applications we observe a very similar trend. Results
TABLE I: Evaluated applications and variants displayed in the heatmaps

| ID | Application | Variant     |
|----|-------------|-------------|
| 1  | Heat        |             |
| 2  | N-Body      |             |
| 3  | Cholesky    |             |
| 4  | Dot-product | OpenMP Tasks|
| 5  | HPCCG       |             |
| 6  | Lulesh      |             |
| 7  | Matrix Multiply |       |
| 8  | Heat        |             |
| 9  | N-Body      |             |
| 10 | Cholesky    |             |
| 11 | Dot-product | OpenMP Fork-Join |
| 12 | HPCCG       |             |
| 13 | Lulesh      |             |
| 14 | Matrix Multiply |       |
| 15 | Heat        |             |
| 16 | N-Body      |             |
| 17 | Cholesky    |             |
| 18 | Dot-product | OmpSs-2     |
| 19 | HPCCG       |             |
| 20 | Lulesh      |             |
| 21 | Matrix Multiply |       |

Fig. 9: Heatmap for pairwise speedup vs exclusive execution for combinations of high resource with low resource usage applications

Fig. 10: Heatmap for pairwise speedup vs exclusive execution for combinations of low resource usage applications

shown in the violin plot in Figure 6b and the heatmap in Figure 9 show how both oversubscription and co-location can achieve speedups but also slowdowns in these application combinations. Unsurprisingly, since the partitions assigned by co-location are static, they do not allow the high resource usage applications to take advantage of the resources left unused by low resource usage applications. This causes a significantly worse performance for co-location in some specific application combinations. Finally, nOS-V co-execution shows the most consistent performance for all cases, having only two cases with minor slowdowns.

Low resource usage applications display a different behaviour. When combining two applications which do not saturate the node’s resources, static partitioning is an adequate strategy, since often there is no need to dynamically redistribute resources between both applications to achieve maximum performance. In the violin plot in Figure 6c and the heatmaps in Figure 10 we observe that oversubscription remains the worst strategy, but co-location slightly outperforms nOS-V, although the slowdowns caused in specific combinations for both strategies are generally small.

As for dynamic co-location, we see DLB can achieve some improvements over static co-location in the execution of high resource usage applications as seen in Figures 7a and 7b. However, it still falls behind the performance achieved by nOS-V in the OmpSs-2 applications. DLB’s restriction of balancing OpenMP applications exclusively at the beginning of parallel regions also limits its applicability versus nOS-V, although this limitation is expected to change with the incorporation of Free Agents [46] to the OpenMP standard.

C. Co-execution of multiple applications

After exploring the co-execution of application pairs, we can expand the experimentation to co-execute larger numbers of applications simultaneously. This scenario will reveal if the
proposed nOS-V runtime can scale adequately when increasing the number of co-executed applications. Since we have a total of 7 different benchmarks, we will explore combinations of 3, 4, 5 and 6 applications simultaneously, and calculate the achieved speedup versus the baseline, which is the exclusive execution of each application one after the other.

Since executing all possible combinations of the 7 benchmarks with high and low resource usage using the 3 different programming models would be unfeasible, we developed an experimental design to execute a representative subset of combinations. For combinations of $N$ applications (where $N$ will be either 3, 4, 5 or 6), we follow these steps:

1. Calculate all possible $N$-combinations of the 7 evaluated benchmarks.
2. For each combination, choose a programming model for each of the applications in the $N$-combination by assigning them in a round-robin fashion (the first application in OmpSs-2, the second in OpenMP tasks, etc.).
3. For each combination, choose a resource usage configuration for each application by assigning them in a round-robin fashion (the first application will be configured as high-resource usage, the second as low, etc.).
4. Execute the application combination 10 times with the chosen programming models and resource usage configurations.

The results of the experimentation are shown in Figure 11. For each of the three tested co-execution approaches (oversubscription, co-location and nOS-V), we plot the speedup obtained against the exclusive execution. Each subfigure corresponds to a different number of simultaneously executing applications.

We highlight several insights from these results. First, the results are coherent with the ones obtained in the co-execution of pairwise combinations. However, adding more applications delivers higher median speedups until the 6-application mark, where the gains are slightly lower.

Secondly, for every tested combination, nOS-V always delivers better performance than exclusive execution, while this only happens with co-location starting at 5 simultaneous applications. Moreover, nOS-V provides the highest median speedup for every number of co-executed applications, with lower variability than the other approaches.

Like the pairwise single-node experiment, co-location delivers the maximum speedup for some specific combinations but also provides significant slowdowns for 3 and 4 application combinations. Thus, the most consistent strategy is still nOS-V, having no slowdowns.

D. Co-execution in distributed applications

We further expand our experimentation by co-executing distributed (MPI+X) applications. We leverage distributed versions of the 7 evaluated benchmarks, combining the existing programming model with Intel MPI for inter-node communication. The versions written in OpenMP Tasks or OmpSs-2 also use TAMPI [47] to overlap communication and computation phases. We executed all pair-wise combinations of applications in 8 and 32 nodes, scaling the input sizes appropriately. In every execution, we included one high-resource usage application and one low-resource one.

In the oversubscription configuration, both co-executed applications run on the same nodes by oversubscribing both application ranks to use the 48 available cores. For co-location, we assign one application to one socket and the other to a different socket, but sharing the same nodes. And finally, in nOS-V, we let the library manage the co-execution on both applications on the same nodes.

Results are shown in Figure 12 for 8 and 32 node executions, and show similar trends to the previous experiments. In both cases, nOS-V shows a higher median speedup while having minimal slowdowns, while both oversubscription and co-location reach significant slowdowns. Oversubscription has a median speedup lower than our baseline, which leads us to conclude that it is not a suitable strategy for co-execution of distributed applications.

Co-location can deliver the maximum speedups for specific combinations, but nOS-V remains the most consistent strategy.

We can also observe the trend that each strategy shows when we increase the number of nodes. For oversubscription and co-location, both the median and minimum speedups worsen when we increase from 8 to 32 nodes, while for nOS-V both metrics improve with the larger node count, achieving a 1.26$x$ median speedup with 32 nodes.

VI. RELATED WORK

Dynamic Load-Balancing (DLB) [18], [19] is a dynamic co-location library, which acts as an arbiter between co-located processes, but lack a global view of the available work on the node. Colocation-aware libraries such as Arachne [48] and Hermes [49] follow this arbiter mechanism. These libraries feature a separate user process called a core arbiter or core broker. Applications are linked to a library component that communicates with this arbiter process, and requests cores according to the instantaneous needs of the application. Note that, exactly like the case for DLB, the arbiter has no global view of the available tasks for each application, only the requested cores, and therefore cannot take node-wide scheduling decisions.

Callisto [50] and LIRA [51] are co-location libraries that improve the dynamic co-location approach by developing an abstraction for runtimes to express their available parallelism in the form of work-tickets. Additionally, runtimes using Callisto also need to use its provided synchronization primitives, to prevent preemption while holding locks. This allows a more fine-grained co-location approach, but still does not provide the same global view a single runtime would have, as information is lost in the work ticket abstraction.

AMCilk [52] is a Cilk runtime framework that supports running multiple Cilk applications in the same node under a single runtime. As applications run under the same runtime, AMCilk does have a global view of the node in the finest granularity. While nOS-V and AMCilk target the same problem, two key distinctions enhance the broader applicability of
nOS-V. Firstly, AMCilk is confined to the Cilk programming model, whereas nOS-V’s abstraction allows integration with various programming models. Secondly, AMCilk mandates linking all potentially co-executed programs into a single binary—an unfeasible requirement for HPC clusters with a dynamic user base. In contrast, nOS-V facilitates the co-execution of applications from diverse programming models and users, dynamically co-executing applications as they are started, without prior knowledge of each other, nor any need to rebuild programs and link them together. Consequently, the proposed approach is more flexible, more secure (since there is no need to share the full address space) and less ad-hoc compared to AMCilk.

We highlight that the presented evaluation for nOS-V is also unique upon that of previous works, as we evaluate applications implemented in different programming models and with different resource usage patterns.

VII. Future Work and Challenges

In this work, we have focused on establishing the fundamentals of co-execution, and we have showcased the potential benefits and use-cases for this approach. However, there are some remaining challenges and some possible extensions.

We believe that the applicability of nOS-V is not limited to co-executing HPC applications, but can be leveraged by any software that can benefit from cooperatively executing in a shared machine. As nOS-V exposes a generic user-level threading interface, it can be leveraged by any application that can use user-level scheduling, even outside of the HPC domain.

There are also many possible extensions for co-execution. For example, to develop automatic mechanisms that can detect the best applications to co-execute to maximize system throughput and efficiency and integrate this mechanism to cluster job schedulers, similar to what has been proposed for co-location [12], [14].

Finally, we plan to tackle another interesting research line to extend co-execution to include GPUs and accelerators, supporting heterogeneous architectures and allowing applications to co-execute on these devices.

VIII. Conclusions

This paper presents nOS-V, a lightweight tasking library that supports application co-execution using node-wide scheduling. Application co-execution is shown to be a powerful alternative to both static and dynamic co-location as well as oversubscription to increase throughput in modern HPC clusters.

The integration with the NanoS6 runtime and the LLVM OpenMP runtime provides co-execution capabilities for OpenMP and OmpSs-2 applications, which are shown in multiple combinations of single-node and distributed benchmarks. During this evaluation, nOS-V is shown to perform better than the other evaluated node-sharing techniques in HPC scenarios, obtaining a median 1.21x speedup over exclusive execution.
for pairwise single-node application combinations, nOS-V delivers similar results when co-executing 3, 4, 5 and 6 application combinations. Furthermore, nOS-V is able to achieve a median 1.26× speedup over exclusive execution when running distributed applications in 32 nodes. Additionally, nOS-V is the only evaluated technique which does not cause significant slowdowns in any combination of high resource usage HPC applications, thus being a low-risk technique which could be implemented by default in HPC clusters.

Acknowledgment

This work was supported by the Spanish Ministry of Science and Innovation (grant PID2019-107255GB) and the Severo Ochoa Program (grant CEX2021-001148-S), both funded by MCIN/AEI/10.13039/50110011033. The Generalitat de Catalunya also supported this work via grant 2021-SGR-01007.

References

[1] A. B. Yoo, M. A. Jette, and M. Grondona, “Stlrm: Simple linux utility for resource management,” in Job Scheduling Strategies for Parallel Processing, D. Feitelson, L. Rudolph, and U. Schwiegelshohn, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 44–60.

[2] S. A. Baset, L. Wang, and C. Tang, “Towards an understanding of oversubscription in cloud,” in 2nd USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services, Hot-ICE’12, San Jose, CA, USA, April 2012, O. Bonaventure and R. R. Kompella, Eds. USENIX Association, 2012. [Online]. Available: https://www.usenix.org/conference/hot-ice12/workshop-program/presentation/baset

[3] Q. Xiong, E. Ates, M. C. Herbordt, and A. K. Coskun, “Tagram: Colocating hpc applications with oversubscription,” in 2018 IEEE High Performance Extreme Computing Conference (HPEC), 2018, pp. 1–7.

[4] F. Wende, T. Steinke, and A. Remenfeld, “The impact of process placement and oversubscription on application performance: A case study for exascale computing,” in Proceedings of the 3rd International Conference on Exascale Applications and Software, ser. EASC ’15. GBR: University of Edinburgh, 2015. p. 13–18.

[5] C. Iancu, S. Hofmeyr, F. Blagojevic, and Y. Zheng, “Oversubscription on multicore processors,” in 2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS), 2010, pp. 1–11.

[6] G. Utrera, J. Corbalan, and J. Labarta, “Scheduling parallel jobs on multicore clusters using cpu oversubscription,” Journal of supercomputing, vol. 68, no. 3, pp. 1113–1140, Jun 2014.

[7] D. Dice and A. Kogan, “Avoiding scalability collapse by restricting concurrency,” in Euro-Par 2014: Parallel Processing, R. Yayahpour, Ed. Cham: Springer International Publishing, 2019, pp. 363–376.

[8] Y. Cui, Y. Wang, Y. Chen, and Y. Shi, “Lock-contention-aware scheduler: A scalable and energy-efficient method for addressing scalability collapse on multicore systems,” ACM Trans. Archit. Code Optim., vol. 9, no. 4, Jan. 2013.

[9] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dammowski, “Towards scalable multiprocessor virtual machines,” in 3rd Virtual Machine Research & Technology Symposium (VM ’04), San Jose, CA: USENIX Association, May 2004. [Online]. Available: https://www.usenix.org/conference/vm-04/towards-scalable-multiprocessor-virtual-machines

[10] M. M. Michael and M. L. Scott, “Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors,” Journal of Parallel and Distributed Computing, vol. 51, no. 1, pp. 1–26, 1998.

[11] L. I. Kontothanassis, R. W. Wisniewski, and M. L. Scott, “Scheduler-conscious synchronization,” ACM Trans. Comput. Syst., vol. 15, no. 1, p. 3–40, Feb. 1997. [Online]. Available: https://doi.org/10.1145/244764.244765

[12] F. V. Zacarias, V. Petrucci, R. Nishitaka, P. Carpenter, and D. Móssé, “Intelligent colocating of hpc workloads,” Journal of Parallel and Distributed Computing, vol. 151, p. 125–137, May 2021.
Parallel and Distributed Computing, ser. HPDC '21. New York, NY, USA: Association for Computing Machinery, 2020, p. 215–226.

[32] D. Álvarez, K. Sala, M. Maroñas, A. Roca, and V. Beltran, “Advanced synchronization techniques for task-based runtime systems,” in Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ser. PPoPP ’21. New York, NY, USA: Association for Computing Machinery, 2021, p. 334–347.

[33] J. Evans, “jemalloc,” 2020. [Online]. Available: https://jemalloc.net

[34] J. Bonwick, “The slab allocator: An object-caching kernel memory allocator,” in Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1, ser. USTC’94. USA: USENIX Association, 1994, p. 6.

[35] J. Bonwick and J. Adams, “Magazines and vmem: Extending the slab allocator to many cpus and arbitrary resources,” in 2001 USENIX Annual Technical Conference (USENIX ATC 01). Boston, MA: USENIX Association, Jun. 2001. [Online]. Available: https://www.usenix.org/conference/2001-usenix-annual-technical-conference/magazines-and-vmem-extending-slab-allocator-many

[36] NVIDIA, “Nvidia multi-process service (mps),” 2020. [Online]. Available: https://docs.nvidia.com/deploy/pdf/CUDA_Core-aware_thread_management.pdf

[37] A. Hori, M. Si, B. Gerofi, M. Takagi, J. Dayal, P. Balaji, and Y. Ishikawa, “Process-in-process: Techniques for practical address-space sharing,” in Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing, ser. HPDC ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 131–143.

[38] A. Vahldieck-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammier, P. Druschel, and D. Garg, “EIRIM: Secure, efficient in-process isolation with protection keys (MPK),” in 28th USENIX Security Symposium (USENIX Security 19). Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1221–1238. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/vahldieck-oberwagner

[39] H. Liljestrand, T. Nyman, K. Wang, C. C. Perez, J.-E. Ekberg, and N. Asokan, “PAC it up: Towards pointer integrity using ARM pointer authentication,” in 28th USENIX Security Symposium (USENIX Security 19). Santa Clara, CA: USENIX Association, Aug. 2019, pp. 177–194. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/liljestrand

[40] Q. P. Security, “Pointer authentication on armv8.3: Design and analysis of the new software security instructions,” Qualcomm Technologies, Inc., Tech. Rep., 2017. [Online]. Available: https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf

[41] Ferran Pallarès Roca, “Extending ompss programming model with task reductions: A compiler and runtime approach,” Bachelor’s Thesis, Barcelona School of Informatics, Universitat Politècnica de Catalunya, 1 2017.

[42] M. Maroñas, K. Sala, S. Mateo, E. Ayguadé, and V. Beltran, “Work-sharing tasks: An efficient way to exploit irregular and fine-grained loop parallelism,” in 2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HPC), Dec 2019, pp. 385–394.

[43] J. M. Perez, V. Beltran, J. Labarta, and E. Ayguadé, “Improving the integration of task nesting and dependencies in openmp,” in 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), May 2017, pp. 809–818.

[44] BSC, “Nanos6 source,” 2021. [Online]. Available: https://github.com/bsc-pm/nanos6

[45] I. Karlin, J. Keasler, and R. Neely, “Lulesh 2.0 updates and changes,” Tech. Rep. LLNL-TR-641973, August 2013.

[46] J. Criado, V. Lopez, J. Vinyals-Ylla-Catala, G. Ramirez-Miranda, X. Teruel, and M. Garcia-Gasulla, “Exploiting openmp malleability with free agent threads and dlb,” in High Performance Computing. ISC High Performance 2022 International Workshops, H. Anzt, A. Bienz, P. Luszczek, and M. Baboulin, Eds. Cham: Springer International Publishing, 2022, pp. 162–175.

[47] K. Sala, X. Teruel, J. M. Perez, A. J. Peña, V. Beltran, and J. Labarta, “Integrating blocking and non-blocking mpi primitives with task-based programming models,” Parallel Computing, vol. 85, pp. 153–166, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167819118303326

[48] H. Qin, Q. Li, J. Speiser, P. Kraft, and J. Ousterhout, “Arachne: Core-aware thread management,” in 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). Carlsbad, CA: USENIX Association, Oct. 2018, pp. 145–160. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/qin

[49] S. Liu, K. Li, and H. Huang, “Hermes: Improving server utilization by colocon-aware runtime systems,” in 2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPC/SmartCity/DSS), 2019, pp. 901–910.

[50] T. Harris, M. Maas, and V. J. Marathe, “Callisto: Co-scheduling parallel runtime systems,” in Proceedings of the Ninth European Conference on Computer Systems, ser. EuroSys ’14. New York, NY, USA: Association for Computing Machinery, 2014.

[51] A. Collins, T. Harris, M. Cole, and C. Fensch, “Lira: Adaptive contention-aware thread placement for parallel runtime systems,” in Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers, ser. ROS’S ’15. New York, NY, USA: Association for Computing Machinery, 2015.

[52] Z. Wang, C. Xu, K. Agrawal, and J. Li, “Adaptive scheduling of multi-programmed dynamic-multithreading applications,” Journal of Parallel and Distributed Computing, vol. 162, pp. 76–88, 2022.