The Cooperative Parallel: A Discussion About Run-time Schedulers for Nested Parallelism

Sara Royuela[1], Maria A. Serrano[1], Marta Garcia[1], Sergi Mateo[1], Jesús Labarta[1], and Eduardo Quiñones[1]

Barcelona Supercomputing Center
{sara.royuela,maria.serranogracia,marta.garcia,sergi.mateo, jesus.labarta,eduardo.quinones} @bsc.es

Abstract. Nested parallelism is a well-known parallelization strategy to exploit irregular parallelism in HPC applications. This strategy also fits in critical real-time embedded systems, composed of a set of concurrent functionalities. In this case, nested parallelism can be used to further exploit the parallelism of each functionality. However, current run-time implementations of nested parallelism can produce inefficiencies and load imbalance. Moreover, in critical real-time embedded systems, it may lead to incorrect executions due to, for instance, a work non-conserving scheduler. In both cases, the reason is that the teams of OpenMP threads are a *black-box* for the scheduler, i.e., the scheduler that assigns OpenMP threads and tasks to the set of available computing resources is agnostic to the internal execution of each team.
This paper proposes a new run-time scheduler that considers dynamic information of the OpenMP threads and tasks running within several concurrent teams, i.e., concurrent parallel regions. This information may include the existence of OpenMP threads waiting in a barrier and the priority of tasks ready to execute. By making the concurrent parallel regions to *cooperate*, the shared computing resources can be better controlled and a work conserving and priority driven scheduler can be guaranteed.

Keywords: Resource Allocation · Concurrency · Runtime Scheduler

1 Introduction

OpenMP, widely used in the High Performance Computing (HPC) domain, is increasingly gaining attention in others domains [23, 22, 15, 36] due to its efficient parallel execution model in shared memory systems, and also its support for heterogeneous computing. This is the case of critical real-time embedded systems, in which new computational intensive functionalities are being developed (e.g., autonomous driving). Here, OpenMP allows to efficiently exploit the performance capabilities of the newest highly parallel and heterogeneous embedded architectures, while benefiting from its programmability and portability capabilities. Moreover, OpenMP has been proven to be time predictable [38, 35, 36], a key aspect to introduce this model in the critical real-time embedded domain.

Royuela, S. [et al.]. The cooperative parallel: A discussion about run-time schedulers for nested parallelism. A: International Workshop on OpenMP. "OpenMP: Conquering the Full Hardware Spectrum: 15th International Workshop on OpenMP, IWOMP 2019: Auckland, New Zealand, September 11-13, 2019: proceedings". Berlín: Springer, 2019, p. 171-185.
The final authenticated version is available online at https://doi.org/10.1007/978-3-030-28596-8_12

OpenMP implements a fork-join model in which the parallel execution is initiated when a `parallel` construct is encountered. Then, a new team of threads (and implicit tasks) is created, associated to the corresponding parallel region. Moreover, OpenMP supports *nested parallelism* in which new parallel regions can be created in contexts that are already being executed in parallel.

Nested parallelism has a number of benefits in both HPC and critical real-time systems: (1) it is a well-known parallelization strategy to support irregular (imbalanced) applications, and (2) it can be used to boost performance at the different levels of a complex system or application, where parallelism is exposed. By using nested parallel regions, applications can benefit from an outer `parallel` construct for exploiting coarse-grain parallelism, and multiple inner `parallel` constructs for exploiting fine-grain parallelism.

However, this strategy presents two important issues: (1) it may result in load imbalance and hence, loss of performance [17], and (2) in the case of critical real-time systems, it may result in an incorrect (or too pessimistic) timing analysis [38, 35, 37, 33]. The reason is that timing analysis is based on work-conserving scheduling policies [35], in which computing resources cannot be idle if there is pending work to do, and priority driven scheduling strategies, where the preference to execute is given to high priority tasks. These properties are not guaranteed between different concurrent parallel regions in OpenMP.

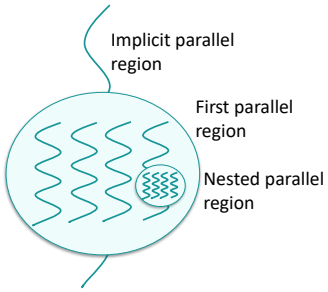
In both HPC and critical real-time systems, the reason to obtain worse or wrong results is that each parallel region operates independently, as a *black-box*, over a set of computing resources, either software resources (e.g., pthreads) or hardware resources (e.g., cores). The scheduler implemented at the OpenMP runtime level is agnostic of the internal execution of each team of threads. As a result, a team can have idle OpenMP threads waiting in a barrier, and occupying computing resources, while there is another team with pending work. The *black-box* problem in critical real-time systems was already identified [36], so the use of a unique team of threads was proposed to parallelize such systems.

In this paper, we propose a new run-time scheduler in which concurrent parallel regions *cooperate* by sharing internal execution information between different teams of threads, e.g., the highest priority among the ready tasks and whether there are idle OpenMP threads waiting in a barrier. This cooperation is used to (1) share computing resources among different (cooperative) teams by defining a new OpenMP thread scheduler, and (2) ensure a work-conserving and priority-driven scheduling, so the timing analysis for critical real-time systems, defined at analysis time, remains valid at runtime.

It is important to remark that our proposed run-time implementation is fully compatible with the current OpenMP specification [2]: the number of OpenMP threads within a parallel region remains fixed, the parallel work defined within each parallel region is executed exclusively by the corresponding team of OpenMP threads, and the thread affinity is preserved. Moreover, since the behavior of this implementation can be essential for some systems, e.g., critical real-time systems, we propose to provide the programmer with a new OpenMP feature to enforce parallel regions to cooperate.

```
1 #pragma omp parallel
2 {
3   if (th_work >= THRESLHOLD) {
4     #pragma omp parallel
5     ditribute_and_compute();
6   }
7   else {
8     compute();
9   }
10 }
```

(a) Code snippet.



(b) Parallel regions.

Fig. 1: Example of nested parallelism with OpenMP.

2 Motivation: The Importance of Nested Parallelism

This section presents the use of nested parallelism in the HPC and critical real-time embedded domains, and motivates the need for a more flexible and controllable scheduler regarding computing resources and OpenMP teams.

2.1 Nested Parallelism in HPC

Before the introduction of the tasking model into OpenMP (specification v.3.0 [1]), nested parallelism was a well-known pattern used to address irregular HPC applications (e.g., tree traversal, adaptive mesh refinement [6], and dense linear algebra [25]). This strategy, which consists on creating new parallel regions in contexts that are already executed in parallel, may help to reduce load-balancing issues, because threads that get more work may decide to solve their work in parallel opening a new parallel region. Figure 1a illustrates this behavior, and Figure 1b shows a diagram of the parallel execution of that code.

Although in several cases the tasking model has replaced nested parallelism to exploit irregular applications [3, 39], the latter still outperforms the former in some cases. This is, for example, the case of imbalanced loops, where dynamic scheduling or tasking may suffer from poor cache behavior and low data reuse due to the inability to bind tasks to cores [8]. This, and the high overhead typically introduced by the runtime to manage the tasking model [26], makes nested parallelism a valid and still valuable mechanism. Particularly, for modern SMP machines with hierarchical memory systems, where outer teams can be created at core level, and inner teams can be created at hardware thread context [30].

The use of nested parallelism may however introduce problems by itself: on one hand, the overhead associated to the creation of parallel regions and the synchronizations [13]; on the other hand, the difficulty of tuning the number of threads of each parallel region. Regarding the former, different works try to mitigate the overhead of OpenMP parallel regions [13, 21] by reusing structures when possible (the most significant techniques are introduced in Section 3).

Regarding the latter, the problem explodes, because an inappropriate definition of the number of threads in nested parallel regions may entail several issues: (1) loss of programmability, because more responsibilities are pushed to the programmer; (2) loss of portability, because a particular set of values might be optimal for one architecture and mediocre in a different one; (3) situations of load imbalance, because threads are waiting at synchronization points while there might be work to do; and (4) oversubscription of the system resources.

Interestingly, the problem of load balancing nested parallel regions has been tackled widely, underscoring the importance of reusing the resources efficiently, reducing oversubscription and boosting data locality. Some solutions are based on a dynamic distribution of the resources between the different nested parallel regions [14], relieving the programmer from the burden of defining the number of threads of each parallel region, and thus enhancing programmability and portability. Others are based in work stealing strategies [29], crucial to ensure work-conserving schedulers that better exploit the possibilities of the system [7]. These works however, consider scheduling solutions in which the internal information about the execution status of the teams executed in parallel is not taken into account. This prevents teams to *cooperate* among them to, for example, avoid having idle threads when there is work to do in other teams.

Next paragraphs introduce an HPC application that presents limitations in the scheduling of different OpenMP parallel regions.

Human Respiratory Simulations: Alya

Coupled runs, consisting in simulations that solve different physics for a single run, are very common in HPC environments [10]. They can be found in a variety of examples from earth science, where some processes simulate the earth while other the ocean, to biological ones. This section describes the couple run applied to a biological simulation of the human respiratory system [18]. It is composed of the simulation of the air going through the human airways, and the simulation of the transport of particles inhaled through the bronchopulmonary tree.

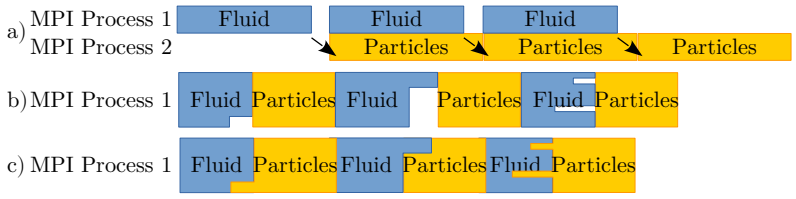


Fig. 2: Coupled run respiratory system.

Concretely, the simulation can be performed in two different instances of the program, one solving the fluid (air), and the other solving the transport of particles (particles inhaled). In this approach, shown in Figure 2a, when the pro-

cesses solving the fluid have computed its velocity, they send it to the processes computing the transport of particles, so both can be pipelined in parallel.

This simulation can also be performed by one instance, as shown Figure 2b, where the program first solves the velocity of the fluid, and then the transport of particles. Considering that both phases include OpenMP parallel loops, and given that the computation is completely independent, each physics (fluid and particles) can be encapsulated within a high level task so they can run in parallel. This approach may result in a load imbalance scenario, however, if the workload is not properly distributed among the threads in the nested parallel.

If the concurrent parallel regions within each high level task are cooperative, the imbalance present in the fluid phase can be used to compute some of the particles parallel region, as shown in Figure 2c.

2.2 Nested Parallelism in Critical Real-time Systems

OpenMP is increasingly being considered as a convenient parallel programming model to develop the most advanced critical real-time systems. One of the main reasons is that the semantics of OpenMP tasks resembles the limited preemptive scheduling models [33, 34, 37]. The preemption strategy is an important factor in real-time scheduling because it determines when real-time functionalities, *real-time tasks*, can be stopped and resumed. Limited preemptive scheduling has been shown to reduce preemption-related overheads compared to *fully-preemptive* systems, while limiting the amount of blocking typical of *fully-non-preemptive* systems [5]. In this regard, OpenMP defines *Task Scheduling Points* (TSPs) as points in the execution of a program at which an OpenMP task can be suspended, allowing the associated computing resource to execute other OpenMP tasks. TSPs are therefore well-identified preemption points of parallel execution that can be considered in the timing analysis of real-time systems [33, 34].

However, current timing analysis techniques are based on run-time schedulers with two important features: (1) a priority-driven execution, and (2) a work-conserving nature. Regarding the former, real-time systems typically assign priorities to real-time tasks and give preference to those tasks with a higher priority (based on the implemented preemption strategy) so that all tasks meet their deadlines. On the other hand, timing analysis for work non-conserving schedulers (i.e., there may be idle threads while there is still work to be done) have been proven to be very complex, and hence lead to unacceptable pessimistic results [35]. As a result, timing analysis techniques impose the real-time system to use a *single team of OpenMP threads* to execute all real-time tasks [36]. The reason lies in the black-box nature of concurrent parallel regions: the execution of each parallel region is governed by the team associated to that region, and each team has access only to the tasks associated to that team. Subsequently, two problems arise: (1) threads encountering a TSP can only schedule tasks that belong to its own team, so highest priority tasks from other teams might be delayed, and (2) threads waiting in a barrier cannot see there is work to do from other teams, so a work-conserving policy cannot be guaranteed.

```
1 // Real-time functionality T1
2 #pragma omp parallel \
3             num_threads(2)
4 #pragma omp single
5 {
6   #pragma omp task priority(1)
7   { ... }
8   #pragma omp task priority(1)
9   { ... }
10  #pragma omp task priority(1)
11  { ... }
12 }
```

(a) Low priority tasks

```
1 // Real-time functionality T2
2 #pragma omp parallel \
3             num_threads(2)
4 #pragma omp single
5 {
6   #pragma omp task priority(2)
7   { ... }
8   #pragma omp task priority(2)
9   { ... }
10  #pragma omp task priority(2)
11  { ... }
12 }
```

(b) High priority tasks

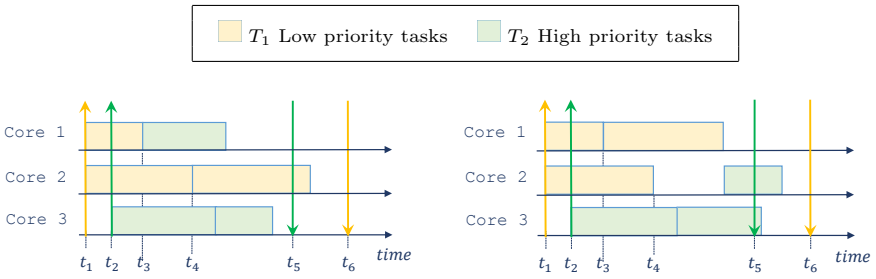
Fig. 3: Concurrent OpenMP parallel regions

As an example, Figure 3 shows the OpenMP code implementing two concurrent real-time tasks[1]. Figures 3a and 3b correspond to the low-priority and high-priority real-time tasks respectively, set by means of the `priority` clause. Moreover, both parallel regions consider two OpenMP threads and a one-to-one mapping to physical resources (cores). Figure 4a shows the time diagram of the expected parallel execution of the OpenMP tasks, as considered by the timing analysis. Low priority OpenMP tasks are created at time instant t_1, and high priority tasks, at t_2, and so low priority tasks start the execution first in cores 1 and 2. The timing analysis considers that, when a low priority task finishes, a high priority task starts the execution, e.g., at time instant t_3. As a result, the system is considered to be schedulable because all deadlines are meet, i.e., the high-priority real-time task completes before t_5 and the low-priority real-time task before t_6.

However, due to the black-box nature of the two concurrent parallel regions, the run-time behavior may be different to that computed at analysis time. Figure 4b shows the time diagram of a compliant OpenMP execution of the two parallel regions, but not consistent with the timing analysis shown in Figure 4a. The reason is that when the thread executing the low priority real-time task reaches the TSP at t_3, it is not aware of the pending high priority OpenMP tasks ready to execute in the other parallel region. As a result, the execution of the high-priority real-time task is delayed, missing its deadline at t_5. In this same scenario, a work-conserving strategy is not ensured, since at t_4, one of the OpenMP threads belonging to low-priority real-time task becomes idle and stays *busy-waiting* in the barrier while there is work to do in the other parallel region, instead of freeing the core to assign it to the other parallel team.

Next paragraphs present a real-time application where nested parallelism is useful, although its usage can cause the issues described in this section.

[1] The parallel region that encloses the two functionalities is not shown for simplicity.



(a) Expected behavior considered in the timing analysis.

(b) Possible behavior according to current OpenMP `parallel` implementations.

Fig. 4: Behavior of two real-time functionalities parallelized with OpenMP.

GPS-aided SINU

Global Positioning System (GPS)-aided Strapdown Inertial Navigation Unit (SINU) system is a low cost motion measurement device commonly used in real-time navigation systems. The system, depicted in Figure 5, is composed of two functionalities: (1) obtain information from accelerometers, gyroscopes and magnetometers to generate outputs in terms of position, velocity and orientation, and (2) combine this information with that obtained from a Global Positioning System (GPS) to minimize errors by implementing a Kalman filter [20].

The Kalman filter is a common recursive application that estimates the internal state of a linear dynamic system from a series of noisy measurements. As depicted in Figure 6, it is separated into two distinct phases: the prediction phase and the measurement phase. Both utilize the Cholesky decomposition to capture the mean and covariance of the system state.

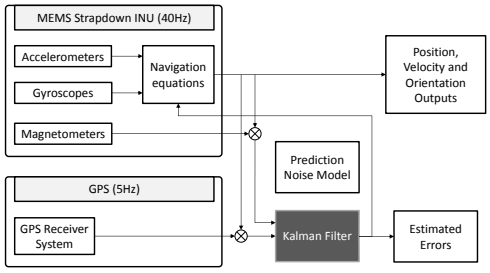


Fig. 5: Block diagram of the GPS-aided SINU system.

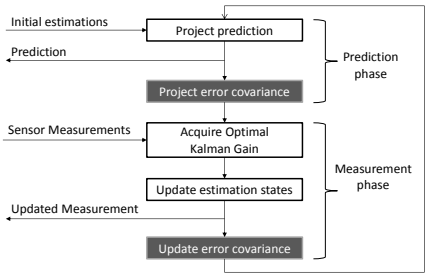


Fig. 6: Block diagram of the Kalman filtering algorithm.

Overall, the GPA-aided SINU is a real-time application that can exploit two levels of parallelism: in the outer level, the computation of the two functionalities (i.e., computing position, velocity and orientation, and estimating errors) can

be performed in parallel; in the inner level, the computation of the Cholesky decomposition used in the Kalman Filter [39] can be further parallelized. The use of nested parallel regions can however prevent the scheduler from fulfilling priorities or ensuring work-conserving executions.

3 Current Implementations

The OpenMP [2] specification defines an *OpenMP thread* as an execution entity with a stack and associated static memory (so called threadprivate memory) that is managed by the OpenMP implementation. Then, this high-level concept may be implemented using different libraries, e.g., pthreads [4] and Windows threads [32]. Hence, when the specification states that a `parallel` construct causes the creation of a team of threads, and that the number of threads remains constant for the duration of that parallel region, it refers to the high-level concept of thread, and not the actual computing resources.

In that context, runtimes must consider the overhead introduced by multi-threading libraries [27] when using computing resources. This includes: (1) thread library startup overhead, that is one-time overhead occurring when the library starts; (2) thread startup overhead, that is time to create threads; (3) per-thread overhead, that is work scheduling overhead; and (4) lock management overhead, that is time spent managing locks. Two of them are particularly interesting when it comes to share resources among teams: the thread startup and the per-thread overheads. On the other hand, thread library startup overhead is usually negligible, and several works tackle lock management overhead [31, 9].

Current OpenMP runtimes (e.g., LLVM [28], libgomp [19]) try to reduce the impact of thread startup overhead by using a *pool of threads* [13], and so avoid the costly creation and destruction of threads. For example, libgomp safely reuses idle threads, considering the processor binding and the thread affinity. As an illustration, for the code shown in Figure 7, LLVM consistently creates X*Y threads, while libgomp creates a number equal or (a bit) bigger than X*Y. Both results prove that LLVM and libgomp use pools of threads.

```
1 #pragma omp parallel num_threads(X)
2    for (int i=0; i<1000; i++) {
3       #pragma omp parallel num_threads(Y)
4        { ... }
```

Fig. 7: OpenMP example with nested parallelism.

Although OS-threads are reused, the overhead associated with these resources is still quite high in architectures with a large amount of cores (e.g., the Intel® Xeon Phi™ Coprocessor [12]), because more threads are potentially created. In this context, Intel® introduced the concept of *hot teams* [30]. This idea, implemented in the LLVM runtime for OpenMP, exploits the fact that OpenMP programs may execute many parallel regions with the same set of parameters (i.e., number of threads, internal control variables and information associated

with the barrier). So, the runtime maintains one structure per team configuration. Intel also supports *nested hot teams*, that keep a pool of threads alive (but idle) during the execution of the non-nested parallel code [21]. This is very useful in cases such as the code presented in Figure 7, where the use of hot teams allows to create the X inner teams once and not destroy them. Without this, the runtime would create and destroy them a thousand times.

These techniques, and the behavior they model, are not controllable at an specification level (and sometimes not even at a runtime level). This is because OpenMP takes the responsibility of scheduling parallel work out of the hands of the programmer. Just the scheduling of loop iterations can be tuned by means of the `schedule` clause, and the *run-sched-var* and *def-sched-var* internal control variables (as determined in Section 2.9.2.1 of the specification [2]). The scheduling of tasks is completely managed by the runtime following the Task Scheduling Constraints defined in the specification (Section 2.10.6).

Some runtime implementations, such as Nanos++ [11], allow a finer control of the scheduler by means of execution modifiers: *throttling policies* (i.e., define whether a new task is created and pushed into the scheduler system, or just a minimal description of the task is created and it is executed right away in the current context), *barrier algorithms* (i.e., how threads waiting at barriers execute remaining work), *traversal order* (i.e., how tasks are traversed, e.g., work-first and breadth-first), and *thread managers* (i.e., control the amount of resources needed for a specific amount of workload). Regarding the latter, there are specific libraries, e.g., Dynamic Load Balancing (DLB) [16], that, attached to the runtime system, allow dynamically managing threads to exploit work-conserving policies.

Overall, a constant behavior of current runtimes is that they tend to apply work-conserving scheduling policies because: (1) they are proven to be optimal for multi-threaded scheduling of Directed Acyclic Graphs [7] (as the ones generated by OpenMP tasks and their dependencies) because it helps load balance, and (2) they are used in the timing analysis performed for real-time systems in order to get not too pessimistic results. This policy defines a work queue for each thread; then, whenever a thread becomes idle, it may steal work from other busy threads. Both the Intel and the GNU OpenMP runtimes (i.e., KMP and libgomp) implement work-stealing for tasks (this aspect can be tuned in Intel by means of the environment variable KMP_TASKING). However, the time spent in busy-waiting is particular to each implementation.

4 Run-time Scheduling based on Cooperative Parallels

As introduced in Section 3, there exist two different kinds of *threads* involved in the execution of an OpenMP code. On one hand, the OpenMP threads are high-level abstractions associated to each team that remain fixed until the team completes. On the other hand, OS-level threads (e.g., pthreads, as used hereinafter) upon which OpenMP threads execute may exist along the execution of the whole application and be reused among different OpenMP teams (using thread pooling), even when the teams execute concurrently. For instance, the

pthreads can be shared among two concurrent parallel regions, and so two (or more) OpenMP threads from different teams (or even the same team) could be mapped to the same pthread. The use of this technique can lead to incorrect executions, considered in Section 4.2.

It is OpenMP-compliant to have several OpenMP threads concurrently mapped to the same pthread. However, in current implementations, the OpenMP thread scheduler is not aware about the internal execution status of each of the parallel regions. As a result, different issues relevant for the HPC and real-time domains may arise, i.e., load imbalance, work non-conserving executions or the impossibility of honoring priorities across teams (see Section 2 for further details).

To address these issues and force a given implementation to provide the run-time behavior required by HPC or critical real-time systems, we define the *cooperative parallels*, in which concurrent parallel regions communicate to exchange information about their execution status. Concretely, the run-time thread scheduler will act as follows:

- Whenever there is an idle OpenMP thread waiting in a `barrier` or a `taskwait`, `barrier`), it will communicate with other concurrent parallel regions to check if there is pending work to do. If this is the case, the idle OpenMP thread will be suspended and the pthread will map to the parallel region with pending work to do. This will allow to provide better load balancing execution for HPC and real-time systems, as well as guaranteeing a work-conserving scheduling execution in case of real-time systems.
- Whenever an OpenMP thread arrives to a TSP, it will check the work pending in its team and will communicate with the other concurrent parallel regions to check the priority of the pending ready tasks. If the most priority ready task belongs to other team, the OpenMP thread will be suspended and the pthread will map to the parallel region in which the highest priority OpenMP task belongs to. OpenMP thread (and then the most priority task). This will allow to accomplish OpenMP tasks priorities as required by real-time systems.

Moreover, we propose to extend the `requires` directive with a new implementation defined requirement called `ext_cooperative_parallel`. This directive forces the implementation of OpenMP run-time to handle teams in such a way that the thread scheduler will take into account the work pending in all teams executing concurrently as described in this section.

Overall, the implementation of the cooperative parallels requires to have a global overview of the *running* OpenMP threads and the pending work of each team, while maintaining the compliance with the OpenMP execution model. Section 4.2 describes the properties that could be affected when implementing *cooperative parallels*, and must remain valid in the OpenMP specification. Before, section 4.1 describes an example of the desired behavior of the *cooperative parallels*.

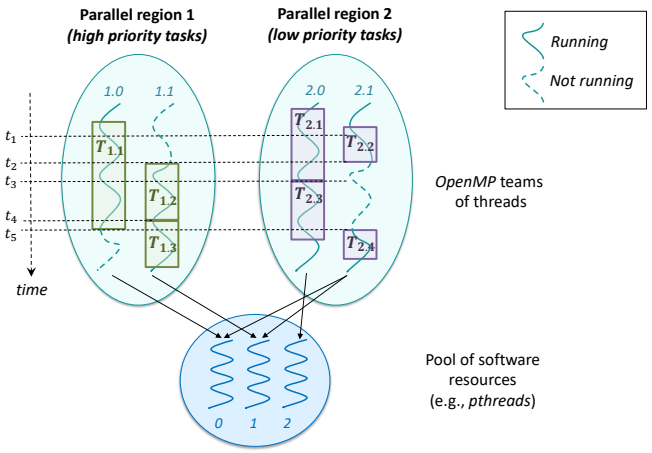


Fig. 8: Example of two *Cooperative parallels*.

4.1 Example

Figure 8 shows an example of two concurrent cooperative parallels, each with two OpenMP threads, that execute OpenMP tasks with a given priority. For simplicity, there are two priority levels for the OpenMP tasks, high and low, executed within the parallel region 1 and 2, respectively. There are three pthreads with IDs 0, 1, and 2. OpenMP threads have IDs 1.0 and 1.1 (parallel region 1), and 2.0 and 2.1 (parallel region 2).

Runtime behavior of the proposed cooperative parallels.

1. Initially, at time instance t_1, we consider that all the OpenMP tasks of both teams are ready to be executed and the OpenMP threads 1.0, 2.0 and 2.1 are being executed in the available pthreads, with the following mapping: 1.0 mapped to 0, 2.0 mapped to 2 and 2.1 mapped to 1.
2. At time instant t_2, the OpenMP thread 2.1 reaches a TSP. Since task $T_{1,2}$ of parallel region 1 has a priority higher than any other ready tasks of the parallel region 2, the OpenMP thread 2.1 is suspended and the pthread 1 is mapped to the OpenMP thread 1.1, and so task $T_{1,2}$ can start executing.
3. At time instant t_3, the OpenMP thread 2.0 reaches a TSP. At this point, task $T_{1,3}$ has a priority higher than pending tasks $T_{2,3}$ and $T_{2,4}$. However, the two OpenMP threads of team 1 are already executing, and so the OpenMP thread 2.0 starts the execution of the task $T_{2,3}$.
4. At time instant t_4, the OpenMP thread 1.1 reaches a TSP but, since all the tasks in parallel region 2 have lower priority than $T_{1,3}$, OpenMP thread 1.1 starts the execution of $T_{1,3}$.
5. Finally, let's assume that at time instant t_5 the OpenMP thread 1.0 reaches a `taskwait`), and so it becomes idle. Therefore, since there is still a ready task pending to be executed in the parallel region 2, OpenMP thread 1.0 is suspended and the pthread 0 is mapped to the OpenMP thread 2.1 to start the execution of task $T_{2,4}$.

4.2 OpenMP compliance

Possible implementations of the *cooperative parallels* concept must take into account some of the features defined in the OpenMP specification in order to be OpenMP compliant. This section analyses these features.

Thread affinity policy. The thread affinity policy (managed in OpenMP by the OMP_PROC_BIND environment variable, the *bind-var* ICV and the `proc_bind` clause) establishes how OpenMP threads are assigned to OpenMP places. If the thread affinity is enabled, the OpenMP implementation should not move OpenMP threads between OpenMP places once a thread in the team is assigned to a place. However, an OpenMP place is defined as "*an unordered set of processors on a device*", i.e., physical resources (hardware threads, cores, etc.), as described in section 6.5 of the OpenMP API v5.0 [2]. Therefore, the OpenMP thread affinity, although compatible with the *cooperative parallels*, may break the desired behavior if a given computing resource is idle to execute work of an OpenMP thread that it is not assigned to it. In any case, the programmer is responsible of defining a thread affinity that does not break the properties brought by the cooperative parallels.

Deadlocks. The use of the same OS-level thread to execute different OpenMP regions associated with different OpenMP threads may generate deadlocks. We recognize two cases: one regarding barriers, and the other regarding locking routines. In the former case, if some OpenMP threads are blocked executing the implicit barrier of one parallel region, and some others are executing the implicit barrier of another parallel region, the OS-level threads may end up having in their call stack the execution of an implicit barrier that they are not going to be able to execute until they do not finish the execution of the current one. In order to solve this issue, our proposal should require an implementation that does not block the different contexts in the call stack of the OS-level, for instance implementing the OpenMP threads as user-level threads. In the latter case, when locking routines are used, compiler analysis [24] can be used to determine the possibility of a deadlock and hence inform the runtime not to safely share threads among OpenMP teams.

Threadprivate variables. OpenMP provides Thread Local Storage mechanisms by means of the `threadprivate` directive, which allows to specify a list of variables that must be replicated for each OpenMP thread. Typically, current implementations use either mechanisms provided by the base language (e.g., the C/C++ __thread attribute), or mechanisms provided by POSIX (i.e., pthread_getspecific(), pthread_setspecific()), because it is the simpler way to go. However, this mechanisms are not valid if different OpenMP threads are mapped to the same OS-level thread, because the latter may end up having incoherent information coming from the different OpenMP threads. For that reason, when OS-level threads are to be reused among different cooperative parallel regions,

the runtime must provide the mechanisms to determine to which parallel region a OS-level thread is assigned at a given point in time, so the proper threadprivate data is accessed.

5 Conclusions

Nested parallelism is a well-known strategy used in the HPC and the critical real-time domains to exploit irregular parallelism in systems exposing parallelism at different levels. However, due to the black-box nature of the parallel regions, nested parallelism may also result in an inefficient parallel execution because of load imbalance in the concurrent parallel regions. Moreover, in case of critical real-time systems, the computation may result incorrect from a timing perspective because of a work non-conserving execution, and the impossibility of fulfilling priorities among different parallel regions.

To address these problems, this paper introduces the concept of *cooperative parallels*, in which the information about the internal execution status of concurrent teams can be shared among them. Moreover, the possible scheduling solutions that can take benefit of this information are analyzed. From that discussion we conclude that a deeper control of the mapping between OpenMP threads and the underlying OS-level threads (e.g., pthreads) is needed to fulfill the work-conserving and priority driven strategies required in both HPC and critical safety systems to achieve better performance and meet timing constraints. An implementation of the cooperative parallel remains as a future work. Nonetheless, this paper discusses the compliance of the cooperative parallel concept with the current OpenMP specification, and provides tips to inspire future implementations.

Acknowledgments. This project has received funding from the European Unions Horizon 2020 research and innovation programme under grant agreement No 780622.

References

1. ARB: Openmp 3.0 specification (2008), https://www.openmp.org/wp-content/uploads/spec30.pdf
2. ARB: Openmp 5.0 specification (2018), https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf
3. Ayguadé, E., Duran, A., Hoeflinger, J., Massaioli, F., Teruel, X.: An experimental evaluation of the new OpenMP tasking model. In: International Workshop on Languages and Compilers for Parallel Computing. pp. 63–77. Springer (2007)
4. Barney, B.L.L.N.L.: Posix threads programming (2017), https://computing.llnl.gov/tutorials/pthreads/
5. Bertogna, M., Xhani, O., Marinoni, M., Esposito, F., Buttazzo, G.: Optimal selection of preemption points to minimize preemption overhead. In: Procedings of the 23rd Euromicro Conference on Real-Time Systems (ECRTS) (2011)

6. Blikberg, R., Sørevik, T.: Load balancing and OpenMP implementation of nested parallelism. Parallel Computing **31**(10-12), 984–998 (2005)
7. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. Journal of the ACM (JACM) **46**(5), 720–748 (1999)
8. Briggs, J.P., Pennycook, S.J., Fergusson, J.R., Jäykkä, J., Shellard, E.P.: Chapter 10 - Cosmic Microwave Background Analysis: Nested Parallelism in Practice. In: High Performance Parallelism Pearls, vol. 2, pp. 171–190 (2015)
9. Caballero, D., Duran, A., Martorell, X.: An OpenMP* barrier using SIMD instructions for Intel® Xeon Phi™ Coprocessor. In: International Workshop on OpenMP. pp. 99–113. Springer (2013)
10. Cajas, J., Houzeaux, G., Vázquez, M., García, M., Casoni, E., Calmet, H., Artigues, A., Borrell, R., Lehmkuhl, O., Pastrana, D., Yánez, D., Pons, R., Martorell, J.: Fluid-Structure Interaction Based on HPC Multicode Coupling. SIAM Journal on Scientific Computing **40**(6), C677–C703 (2018)
11. Center, B.S.: Ompss user guide (2019), https://pm.bsc.es/ftp/ompss/doc/user-guide/index.html
12. Chrysos, G.: Intel® Xeon Phi™ Coprocessor - The architecture. Intel Whitepaper **176** (2014)
13. Dimakopoulos, V.V., Hadjidoukas, P.E., Philos, G.C.: A microbenchmark study of OpenMP overheads under nested parallelism. In: International Workshop on OpenMP. pp. 1–12. Springer (2008)
14. Duran, A., Gonzalez, M., Corbalán, J.: Automatic thread distribution for nested parallelism in OpenMP. In: Proceedings of the 19th annual international conference on Supercomputing. pp. 121–130. ACM (2005)
15. Ferry, D., Li, J., Mahadevan, M., Agrawal, K., Gill, C., Lu, C.: A real-time scheduling service for parallel tasks. In: 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS). pp. 261–272. IEEE (2013)
16. Garcia, M., Corbalan, J., Labarta, J.: LeWI: A Runtime Balancing Algorithm for Nested Parallelism. In: International Conference on Parallel Processing. pp. 526–533 (2009)
17. Garcia Gasulla, M.: Dynamic load balancing for hybrid applications (2017)
18. Garcia-Gasulla, M., Mantovani, F., Josep-Fabrego, M., Eguzkitza, B., Houzeaux, G.: Runtime mechanisms to survive new hpc architectures: A use case in human respiratory simulations. The International Journal of High Performance Computing Applications (2019)
19. GNU: libgomp (2019), https://gcc.gnu.org/onlinedocs/libgomp/
20. Hun, L.C., Yeng, O.L., Sze, L.T., Chet, K.V.: Kalman Filtering and Its Real-Time Applications. In: Real-time Systems (2016)
21. Jeffers, J., Reinders, J., Sodani, A.: Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition. Morgan Kaufmann (2016)
22. Kim, J., Kim, H., Lakshmanan, K., Rajkumar, R.R.: Parallel scheduling for cyber-physical systems: Analysis and case study on a self-driving car. In: Proceedings of the ACM/IEEE 4th international conference on cyber-physical systems. pp. 31–40. ACM (2013)
23. Knafla, B., Leopold, C.: Parallelizing a real-time steering simulation for computer games with OpenMP. Parallel Computing: Architectures, Algorithms, and Applications **15**, 219 (2008)
24. Kroening, D., Poetzl, D., Schrammel, P., Wachter, B.: Sound static deadlock analysis for C/Pthreads. In: 31st International Conference on Automated Software Engineering. pp. 379–390. IEEE (September 2016)

25. Kurzak, J., Dongarra, J.: Implementing linear algebra routines on multi-core processors with pipelining and a look ahead. In: International Workshop on Applied Parallel Computing. pp. 147–156. Springer (2006)
26. LaGrone, J., Aribuki, A., Chapman, B.: A set of microbenchmarks for measuring OpenMP task overheads. In: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA). p. 1. Citeseer (2011)
27. Lindberg, P.: Performance obstacles for threading: How do they affect OpenMP code. Intel Software Developer Zone (2009), https://software.intel.com/en-us/articles/performance-obstacles-for-threading-how-do-they-affect-openmp-code
28. LLVM: OpenMP*: Support for the OpenMP language (2019), https://openmp.llvm.org
29. Meadows, L., Pennycook, S.J., Duran, A., Wilmarth, T., Cownie, J.: Workstealing and nested parallelism in SMP systems. In: International Workshop on OpenMP. pp. 47–60. Springer (2016)
30. Meadows, L., Kim, J.: Chapter 18 - Exploiting Multilevel Parallelism in Quantum Simulations. In: High Performance Parallelism Pearls, pp. 335–354. Volume 2: Multicore and Many-core Programming Approaches (2015)
31. Nanjegowda, R., Hernandez, O., Chapman, B., Jin, H.H.: Scalability evaluation of barrier algorithms for OpenMP. In: International Workshop on OpenMP. pp. 42–52. Springer (2009)
32. Russinovich, M.E., Solomon, D.A., Ionescu, A.: Windows internals. Pearson Education (2012)
33. Serrano, M.A., Melani, A., Bertogna, M., Quiñones, E.: Response-time analysis of DAG tasks under fixed priority scheduling with limited preemptions. In: Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE) (2016)
34. Serrano, M.A., Melani, A., Kehr, S., Bertogna, M., Quiñones, E.: An Analysis of Lazy and Eager Limited Preemption Approaches under DAG-based Global Fixed Priority Scheduling. In: Proceedings of the 20th IEEE International Symposium on Real-Time Distributed Computing (ISORC) (2017)
35. Serrano, M.A., Melani, A., Vargas, R., Marongiu, A., Bertogna, M., Quiñones, E.: Timing characterization of OpenMP4 tasking model. In: International Conference on Compilers, Architecture and Synthesis for Embedded Systems. pp. 157–166. IEEE (2015)
36. Serrano, M.A., Royuela, S., Quiñones, E.: Towards an OpenMP Specification for Critical Real-Time Systems. In: International Workshop of OpenMP. pp. 143–159 (2018)
37. Sun, J., Guan, N., Wang, Y., He, Q., Yi, W.: Scheduling and analysis of real-time Openmp task systems with tied tasks. In: Proceedings of Real-Time Systems Symposium (2017)
38. Vargas, R., Quiñones, E., Marongiu, A.: OpenMP and timing predictability: a possible union? In: Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition. pp. 617–620 (2015)
39. YarKhan, A., Kurzak, J., Luszczek, P., Dongarra, J.: Porting the PLASMA numerical library to the OpenMP standard. International Journal of Parallel Programming **45**(3), 612–633 (2017)