Timing characterization of OpenMP4 tasking model

Timing Characterization of OpenMP4 Tasking Model

Maria A. Serrano
Barcelona Supercomputing Center and
Technical University of Catalonia,
Barcelona, Spain
maria.serranogracia@bsc.es

Alessandra Melani
Scuola Superiore Sant'Anna,
Pisa, Italy
alessandra.melani@sssup.it

Roberto Vargas
Barcelona Supercomputing Center and
Technical University of Catalonia,
Barcelona, Spain
roberto.vargas@bsc.es

Andrea Marongiu
Swiss Federal Institute of Technology,
Zurich, Switzerland
a.marongiu@iis.ee.ethz.ch

Marko Bertogna
University of Modena and Reggio
Emilia, Modena, Italy
marko.bertogna@unimore.it

Eduardo Quiñones
Barcelona Supercomputing Center,
Barcelona, Spain
eduardo.quinones@bsc.es

Abstract

OpenMP is increasingly being supported by the newest high-end embedded many-core processors. Despite the lack of any notion of real-time execution, the latest specification of OpenMP (v4.0) introduces a tasking model that resembles the way real-time embedded applications are modeled and designed, i.e., as a set of periodic task graphs. This makes OpenMP4 a convenient candidate to be adopted in future real-time systems. However, OpenMP4 incorporates as well features to guarantee backward compatibility with previous versions that limit its practical usability in real-time systems. The most notable example is the distinction between tied and untied tasks. Tied tasks force all parts of a task to be executed on the same thread that started the execution, whereas a suspended untied task is allowed to resume execution on a different thread. Moreover, tied tasks are forbidden to be scheduled in threads in which other non-descendant tied tasks are suspended. As a result, the execution model of tied tasks, which is the default model in OpenMP to simplify the coexistence with legacy constructs, clearly restricts the performance and has serious implications on the response time analysis of OpenMP4 applications, making difficult to adopt it in real-time environments.

In this paper, we revisit OpenMP design choices, introducing timing predictability as a new and key metric of interest. Our first results confirm that even if tied tasks can be timing analyzed, the quality of the analysis is much worse than with untied tasks. We thus reason about the benefits of using untied tasks, deriving a response time analysis for this model, and so allowing OpenMP4 untied model to be applied to real-time systems.

1. Introduction

Modern high-end embedded systems are increasingly concerned with providing higher performance in real-time, challenging the performance capabilities of current architectures. The advent of next-generation many-core embedded platforms has the chance of intercepting this converging need for predictable high-performance, but an evolution of programming paradigms is required to combine traditional requirements, i.e., ease of programmability and efficient exploitation of parallel resources, with timing analysis techniques.

OpenMP [2], the de-facto standard for shared memory parallel programming in high-performance computing (HPC), is increasingly being adopted also in embedded parallel and heterogeneous systems [11] [4] [6] [9] [17] [19]. Originally focused on a *thread-centric* model to exploit massively data-parallel and loop-intensive types of applications, the latest specification of OpenMP v4.0 (a.k.a. OpenMP4) has evolved to a *task-centric* model which enables very sophisticated types of fine-grained and irregular parallelism.

The current OpenMP4 tasking model allows the programmer to define explicit tasks and the data dependencies existing among them. At run-time, tasks are executed by a *team of threads*, which allows effectively utilizing many-core architectures while hiding its complexity to the programmer. Although the OpenMP specification leaves open the implementation of the task-to-thread scheduler, available implementations typically rely on breadth-first (BFS) [16] and work-first (WFS) [15] schedulers. The former creates all children tasks before executing them; the latter executes new tasks immediately after they are created (suspending the execution of the parent task, which potentially can be resumed on a different thread).

Several practical issues have been addressed by the OpenMP language committee when designing the tasking model specification [10], considering simplicity of use, compatibility with the existing specification and performance as the main metrics of interest. However, the requirements for the co-existence of a "legacy" thread-based execution model and a new task-based execution model led to conflicting needs for choosing in the default settings. Unfortunately, none of the considered design choices took time predictability into account, as this is traditionally not a relevant metric in the HPC domain. In this paper we revisit such design choices introducing *timing predictability* as a new key metric of interest.

Probably the most notable example of a "trade-off" design choice between the old (thread-centric) and the new (task-centric) specification is the distinction between *tied* and *untied* tasks. In state-of-the-art *tasking* programming models [15], there are points in the execution of a program where a thread can suspend the execution of the current task and switch to a new task. The suspended task can resume execution on a different thread, if available. This execution model implements a *work-conserving* policy, which ensures that no thread remains idle if there is work to be done. Ultimately, this behavior guarantees efficient exploitation of a multi-core processor and facilitates the timing characterization of parallel execution (see Section 3.2 for further details). Unfortunately, the thread-centric nature of many of the original OpenMP constructs exposes a number of issues if migration of a task from one thread to another is allowed. To give a few examples:

- thread id-based work partitioning among threads, at the core of older OpenMP programming practices (up to v2.5), would break the semantics of the program;
- mutually-exclusive code regions (e.g. `critical` construct) would result in deadlock scenarios, as critical section locks are owned by threads;
- private data to a thread (e.g., `threadprivate` variables) should also migrate with the task, which is not easy nor efficient to implement.

As a solution to the problem, the OpenMP4 specification states that tasks must be *tied* by default and that all *parts* of a *tied* task must execute on the same thread in which it started executing. Moreover, the OpenMP4 specification defines a set of *task scheduling constraints* in which tied tasks are not allowed to be scheduled in threads in which other non-descendant tied tasks are suspended. Overall, the *tied* tasking model results in a non work-conserving task scheduling approach. The knowledgeable programmer can specify a work-conserving approach by using *untied* tasks, which are allowed to resume execution on a different thread when suspended. As it often happens in OpenMP, the programmer takes responsibility for guaranteeing correct execution of the program.

The use of *tied* tasks clearly restricts the performance and has serious implications on the schedulability analysis of OpenMP4 applications. In this paper we explore the `tied` and `untied` clauses from a timing analyzability point of view, considering both scheduling strategies BFS and WFS and estimating the impact that such programming model features have on the capability of our analysis to provide precise and tight timing guarantees.

Our first experiences suggest that, despite OpenMP4 tasking model can be timing analyzed, the quality of the analysis is significantly worse when *tied* tasks are assumed, leading to a very pessimistic and conceptually complicated worst-case scheduling scenario. The use of *untied* tasks, instead, allows deriving an efficient schedulability test considering the scheduling constraints imposed by OpenMP tasking directives and clauses, due to the work-conserving nature of the *untied* tasking model. Overall, this paper demonstrates that OpenMP4 can be effectively applied in real-time systems if the untied tasking model is adopted.

2. OpenMP4 Tasking Model

This section summarizes the main characteristics of the OpenMP specification focusing on the tasking support provided by the latest versions [1][2].

2.1 From thread-centric to task-centric Model

Up to specification version 2.5, OpenMP assumed a thread-centric execution model, in which the programmer could determine the thread in which a code segment was executing (with the OpenMP call `omp_get_thread_num`). Following the single program, multiple data (SPMD) programming paradigm, the programmer was allowed to explicitly perform different work on various threads, based on their *id*. Moreover, the programmer could assign private storage to the thread (marking target variables with the `threadprivate` directive) that remained valid across executions of different parallel regions.

With the introduction of OpenMP 3.0 specification, the `task` construct was introduced, exposing a higher level of abstraction to programmers. A task is an independent parallel unit of work, specified by an instance of executable code and its data environment, and executed by a given available thread from a *team*. This new model, known as "tasking model", provides a very convenient abstraction of parallelism, being the run-time in charge of scheduling tasks to threads. With the latest 4.0 specification, OpenMP introduced advanced features to express dependencies among tasks, resembling sporadic DAG real-time scheduling models [19], as will be shown in Section 3.1. This makes OpenMP4 a firm candidate to be adopted in future real-time systems.

For backward compatibility reasons, both models need to coexist in the last OpenMP specification. As a result, the tasking model introduces certain characteristics that complicate the derivation of a tight timing analysis. The next section introduces the design choices in the default settings to allow both execution models to coexist.

2.2 OpenMP4 Tasking Model

An OpenMP program starts with an *implicit task*[1] surrounding the whole program. This implicit task is executed by a single thread, called the *master* or *initial* OpenMP thread that runs sequentially.

When the thread encounters a `parallel` construct, it creates a new *team* of threads, composed of itself and $n-1$ additional threads (n being specified with the `num_threads` clause).

When a thread encounters a `task` construct, a new *explicit task* is created and assigned to one of the threads in the current team for immediate or deferred execution, based on additional clauses: `depend`, `if`, `final` and `untied`.

The `depend` clause forces sibling tasks to be executed in a given order based on dependences defined among data items. The `if` clause makes the new task to be *undeferred* and executed by a thread of the team, suspending the current task region until the new task completes. Similarly, the `final` clause makes all descendants of the new task to be *included*, meaning that they must execute immediately by the encountering thread.

The `untied` clause (which is the focus of this paper) makes the new generated task not being tied to any thread and so, in case it is suspended, it can later be resumed by any thread in the team. By default, OpenMP tasks are *tied* to the thread that first starts their execution. Hence, if such tasks are suspended, they can later only be resumed by the same thread.

The completion of a subset or all explicit tasks bound to a given parallel region may be specified through the use of task synchronization constructs, i.e., `taskwait`, `taskgroup` and `barrier` constructs. The `taskwait` construct specifies a wait on completion of child tasks of the current task. The `taskgroup` construct specifies a wait on completion of child tasks of the current task and their descendant tasks. The `barrier` construct specifies an explicit barrier where all threads of the team must complete execution before any of them is allowed to continue execution beyond the barrier.

Figure 1 shows an OpenMP program example. The code enclosed in the `parallel` construct defines a team of N threads. The `single` construct at line 2 is used to specify that only one of the threads in the team has to execute the implicit task region T_0. When the thread executing T_0 encounters the `task` constructs at lines 4, 12 and 19, new tasks T_1, T_3 and T_5 are generated. T_3 will not start its execution until T_1 finishes because there exits a data dependency (T_1 produces item x and T_3 consumes it). The thread executing T_1 creates task T_2 and the thread executing T_3 creates task T_4. Similarly, the thread executing T_5 creates task T_6, T_6 creates tasks T_7 and T_9, and T_7 creates T_8.

All tasks are guaranteed to have completed at the *implicit barrier* at the end of the parallel region at line 39. Moreover, task T_1 will wait on the `taskwait` at line 8 until task T_2 has completed and similarly T_3 will wait T_4, T_5 will wait T_6, task T_7 will wait T_8 and T_0 will wait on the `taskwait` at line 37 until tasks T_1, T_3 and T_5 have completed before proceeding past the `taskwait`.

[1] An implicit task is not created by the programmer but by the run-time; tasks created by the programmer using the `task` construct are commonly referred to as *explicit tasks*.

```
#pragma omp parallel num_threads(N) {
#pragma omp single { // T0
  part00
  #pragma omp task depend(out:x) // T1
  { part10
    #pragma omp task  { part20 } // T2
    part11
    #pragma omp taskwait
    part12
  }
  part01
  #pragma omp task depend(in:x) // T3
  { part30
    #pragma omp task { part40 } // T4
    #pragma omp taskwait
    part31
  }
  part02
  #pragma omp task // T5
  { part50
    #pragma omp task { // T6
      part60
      #pragma omp task  // T7
      { part70
        #pragma omp task { part80 } // T8
        #pragma omp taskwait
        part71
      }
      part61
      #pragma omp task { part90 } // T9
      part62
    }
    part51
    #pragma omp taskwait
    part52
  }
  #pragma omp taskwait
  part03
}}
```

Figure 1: Example of an OpenMP program using tasking constructs.

A predecessor/descendant relationship exists among tasks. Predecessor tasks of T_i are T_i's parent task and parent's predecessor tasks. On the contrary, the descendant tasks of T_i are T_i's child tasks and child's descendant tasks. For example, predecessor tasks of T_4 are T_3 and T_0 and the descendant tasks of T_5 are T_6, T_7 , T_8 and T_9.

2.3 OpenMP Task-to-thread Scheduling

2.3.1 Task Scheduling Points (TSP) and Task Scheduling Constraints (TSC)

OpenMP [2] defines *task scheduling points* (*TSP*) as points in the program where the encountering task can be suspended and the hosting thread can be rescheduled to a different task. As a result, *TSPs* divide task regions into *task parts* (or simply *parts*) executed uninterrupted from start to end. The example shown in Figure 1 identifies the parts in which each task region is divided, e.g. T_0 is composed of part$_{00}$, part$_{01}$, part$_{02}$ and part$_{03}$.

TSPs occur upon (1) task creation and completion, (2) at task synchronization points such as taskwait directives and taskgroup directives, (3) at explicit and implicit barriers and (4) upon taskyield directives, in which the current task can be suspended in favor of the execution of a different task[2]. When a thread encounters a *TSP*, it can begin or resume the execution of a task, provided that a set of *task scheduling constraints* (*TSC*) are fulfilled:

1. An *included* task must be executed immediately after the task is created.
2. Scheduling of new *tied* tasks is constrained by the set of task regions that are currently tied to the thread, and that are not suspended in a barrier region. If this set is empty, any new *tied* task may be scheduled. Otherwise, a new *tied* task may be scheduled only if it is a descendant task of every task in the set.
3. A *dependent* task shall not be scheduled until its task data dependencies are fulfilled.
4. When a task contains an if clause and its associated condition evaluates to false, the task is executed immediately if the rest of the *TSC*s are met.

```
 1 for (i=0; i < N; i++) {
 2   #pragma omp task // T1
 3   {
 4     foo();
 5     #pragma omp critical
 6     {
 7       bar();
 8       #pragma omp task // T2
 9       foobar();
10     }
11   }
12 }
```

Figure 2: Example of an OpenMP program using synchronization constructs.

A program relying on any other *TSC* or performing a different action when a *TSP* is encountered is non OpenMP-conforming.

TSC 2 may considerably reduce the number of threads available to tied tasks, impacting on both performance and timing predictability. Next section explains the reason of such a design choice.

2.3.2 Understanding TSC 2

TSC 2 prevents tied task from being scheduled in threads in which other non-descendant tied tasks are suspended. This inhibits the run-time from incurring in a deadlock situation when the critical synchronization construct is used within a task [10]. The critical construct is a synchronization mechanism inherited from the *thread-centric* model that defines a region that can be exclusively executed by a single thread at a time [2]. The reason of the deadlock situation is because the owner of the lock is a thread and not a task.

Figure 2 shows an example in which the critical construct is used within a task. The example will create as many T_1 and T_2 task instances as for-loops iterations. When the thread executing the first instance of T_1 enters the critical section, the thread obtains the lock so that no other thread can access it. However, the execution of this task instance T_1 can be suspended when reaching the *TSP* at line 8 (T_2 task construct) and so the same thread may execute a different task. If the thread started executing another instance of T_1, it would eventually reach the critical section again, but this time would not be able to enter it as this thread already has the lock. This leads to a deadlock situation in which the thread has the lock due to the first T_1 instance and, at the same time, is blocked in the critical section due to the second T_1 instance. Notice that the critical construct does not imply a *TSP*, so that the thread is stalled in the second T_1 task instance.

The *TSC 2* prevents the same thread from executing any tied task that is not descendant of T_1. Note that T_2 is a descendant task of T_1 and so it is allowed to execute it.

When untied tasks are used, the responsibility of the utilization of critical sections or thread-specific information lies on the programmer.

2.3.3 Scheduling Algorithms

When a task encounters a *TSP*, the program execution branches into the OpenMP runtime system, where task-to-thread schedulers

[2] An additional *TSP* is implied at OpenMP construct target but we do not consider in this paper for the sake of simplicity.

can: 1) begin the execution of a task region bound to the current team or 2) resume any previously suspended task region bound to the current team. The order in which these two actions are applied is not specified by the standard. An ideal task scheduler will schedule tasks for execution in a way that maximizes concurrency while accounting for load imbalance and locality to facilitate better performance. Current implementations of OpenMP run-times are based on two main task scheduling policies:

Breadth-First scheduling (BFS). When a task is created, it is placed into a pool of tasks and the encountering thread continues the execution of the parent task. Tasks placed in that pool can then be executed by any available thread from the team. Due to *TSC 2*, when a tied task is suspended in a *TSP*, it is placed into the private pool of tasks associated to its execution thread. Untied tasks instead are queued into a pool of tasks accessible by all threads in the team. Access to these pools can be LIFO (i.e., last queued tasks will be executed first) or FIFO (i.e., oldest queued tasks will be executed first). Threads will always try to schedule first a task from their local pool. If it is empty then they will try to get tasks from the team pool. An example of BFS is shown in [16].

Work-first scheduling (WFS). New tasks are executed immediately after they are created by the parent's thread, suspending the execution of the parent task. When a task is suspended in a *TSP*, it is placed in a per thread local pool which can be accessed in a LIFO or FIFO manner. When looking for tasks to execute, threads will look into their local pool. If it is empty, they will try to steal work from other threads. When stealing from another thread pool, to comply with OpenMP restrictions, tied task cannot be stolen from its associated thread. The Cilk scheduler [15] pertains to this family. In particular, it is a WFS where access to the local pool is LIFO, tries to steal the parent task first and otherwise steals from another thread pool in a FIFO manner.

WFS tends to obtain better performance results than BFS due to two reasons [3]: (1) the WFS strategy tries to follow the serial execution path hoping that if the sequential algorithm was well designed, it will lead to better data locality; and (2) it also has the property of minimizing space: in a BFS strategy all tasks coexist simultaneously, because all child tasks are created before executing them. On the contrary, WFS creates the same number of tasks, but fewer tasks have to exist at the same time because they are executed immediately after they are created. However, OpenMP implementations typically use BFS due to the tied tasks default restriction: if WFS is implemented, when a tied task T_i creates a child tied task T_{i+1}, this one starts its execution in T_i's thread. Then, T_i is suspended and it cannot resume its execution until T_{i+1} finishes or suspends in a TSP because it is tied to a thread. Therefore, WFS turns a parallel program with tied tasks into a sequential execution, as will be shown in Section 5.1.

Overall, TSC 2 and the semantics of *tied* tasks prevent the implementation of work-conserving schedulers. We will discuss in the next section how this limits the analyzability of the tied task execution model.

3. Timing characterization of OpenMP4

The sporadic DAG scheduling model [21] [20] [5] [7] [13] generalizes the *fork-join* execution model to allow exploitation of parallelism within tasks. This section explains how to derive an OpenMP-DAG and the implications that the OpenMP4 tasking model has on the scheduling.

3.1 OpenMP4 Tasking Model and Sporadic DAG Scheduling Model

Despite the current OpenMP specification lacks any notion of real-time scheduling semantics, the structure and syntax of an OpenMP program have certain similarities with DAG-based models presented in the real-time community, as shown in [19].

In the sporadic DAG model, each task (called *DAG-task*) is represented by a *directed acyclic graph* (DAG) $G = (V, E)$, a period (T) and a deadline (D). Each node $\upsilon_i \in V$ denotes a sequential operation or *job*, characterized by a *worst-case execution time* (WCET) estimate c_i. The edges represent the dependencies between jobs: if $(\upsilon_1, \upsilon_2) \in E$, then job υ_1 must complete its execution before job υ_2 can start executing. In other words, the DAG captures scheduling constraints imposed by dependencies among jobs and it is annotated with a WCET estimate c_i of each individual job. When a DAG-task is released at time t, all jobs in V are ready to execute if precedence constraints are fulfilled, and all jobs must finish before time $t + D$.

Moreover, the sporadic DAG model defines a *chain* as a sequence of jobs $\lambda = \upsilon_1, \upsilon_2, \ldots, \upsilon_k$ such that $(\upsilon_i, \upsilon_{i+1})$ is an edge in G, $1 \leq i < k$. The length of this chain is the sum of the WCETs of all its nodes, i.e., len(λ) = $\sum_{i=1}^{k} c_i$. The *critical path* of G is the longest chain in G and its length is denoted by len(G). Finally, the *volume* of a DAG-task is defined as the sum of all WCETs of its jobs, i.e., vol(G) = $\sum_{\upsilon_i \in V} c_i$.

The execution of an OpenMP program has certain similarities with the execution of a DAG-task: (1) the execution of a task *part* in the OpenMP program resembles the execution of a job in V for which WCET estimation can be derived [19]; (2) the edges E in the DAG model can be used to model the `depend` clause, which forces tasks not to be scheduled until all precedence constraints are fulfilled; the `if` and `final` clauses, which make the task to be suspended until the new task completes execution; and the synchronization directives.

Figure 3 shows the OpenMP-DAG obtained by the example program presented in Figure 1. Tasks *parts* are the nodes in V and the *TSPs* encountered at the end of a task *part* (task creation or completion, task synchronization) are the edges in E, The figure distinguishes three different types of edges: control flow dependencies (dotted arrows) that force *parts* to be scheduled in the same order as they are executed within the task, *TSP* task creation dependencies (dashed arrows) that force tasks to start/resume execution after the corresponding *TSP*, and *TSP* synchronization dependencies (solid arrows) that force the sequential execution of tasks as defined by the `if` clause, the `depend` clause and the `taskwait` synchronization construct. All edges express a precedence constraint.

3.2 Schedulability Problem for OpenMP4

Once the OpenMP-DAG of an OpenMP application is derived, the problem of schedulability reduces to the problem of determining whether the DAG can be scheduled on the available threads to complete within a specified relative deadline D, i.e., within D time units from the release of the DAG.

The OpenMP4 specification is agnostic of the task-to-thread scheduling implemented by the run-time. It is therefore the responsibility of the run-time developer to implement the most suitable scheduler for the OpenMP system, guaranteeing that the *TSCs* defined in Section 2.3.1 are fulfilled.

In high-performance systems, the main goal of task-to-thread schedulers is to maximize the occupancy of threads. In real-time systems, the main goal is not only maximizing the use of resources but also to provide timing guarantees. The use of *work-conserving* schedulers facilitates the timing characterization of parallel execution.

Definition 1. *A scheduling algorithm is said to be* work-conserving *if and only if it never idles threads whenever there exists at least one ready job awaiting execution in the system.*

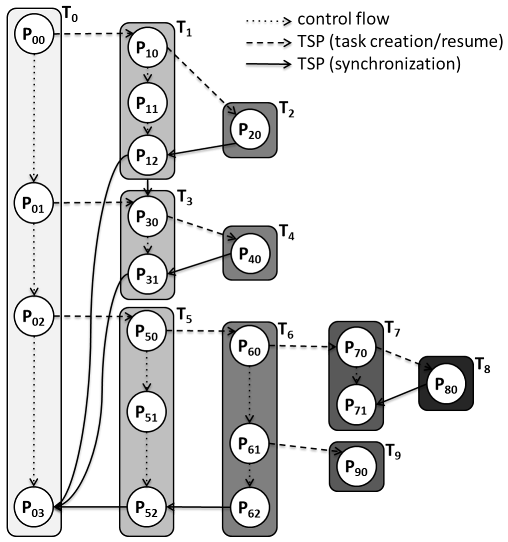


Figure 3: DAG corresponding to the program in Figure 1.

For work-conserving schedulers, the problem of determining the schedulability of an OpenMP-DAG has a strong correspondence with the makespan[3] minimization problem of a set of precedence constrained jobs (task *parts* in our case) on identical processors (threads in a team in our case), which is known to be strongly NP-hard by a result of Lenstra and Rinnooy Kan [12]. However, the Graham's *List Scheduling* algorithm [18], which can be implemented in polynomial run-time complexity, provides an approximation of $2 - \frac{1}{m}$ for this problem, being m the total number of threads in a team. This means that this algorithm is able to produce for any input task graph a value of the makespan that is at most $2 - \frac{1}{m}$ times the optimal one. The *List Scheduling* algorithm simply maps tasks to available threads in a team without introducing idle times if not needed, i.e., it implements a *work-conserving* scheduling algorithm.

Therefore, implementing OpenMP4 run-time incorporating work-conserving schedulers seems to be the best option. Current OpenMP4 run-time implementations already incorporate work-conserving schedulers, i.e. BFS and WFS (see Section 2.3.3).

Unfortunately, TSC 2 and the execution semantics of tied tasks force these schedulers not to be work-conserving. On the one hand, *TSC 2* forbids a new `tied` task to be scheduled to a thread where it is not a descendant of all the other suspended *tied* tasks already assigned to this thread. This may potentially reduce the number of threads in the team that can be assigned to new tied tasks. On the other hand, *tied* task parts cannot migrate when the task is resumed and its corresponding thread is being used by another descendant *tied* task or an untied task. These constraints impose extra conditions on the schedulability analysis of OpenMP4 programs.

This is not the case for untied tasks, which are not subject to *TSC 2*, allowing *parts* of the same task to execute on different threads; so, when a task is suspended, the next *part* to be executed can be resumed on a different thread. Hence, the execution model of untied tasks allows BFS and WFS to be work-conserving.

Overall, the additional requirements imposed by the use of tied tasks suggest devising distinct timing characterizations for the two types of OpenMP4 tasks, i.e., tied and untied. Hence, in the rest of this paper we analyze both types of tasks to characterize their timing behavior, outlining the major challenges posed by the use of tied tasks in a real-time domain.

4. Schedulability Analysis of Untied Tasks

The `untied` clause allows a task to be executed in any thread and, in case it is suspended, to be resumed by any thread in the team. In other words, the task can be freely migrated across threads during its execution. This flexibility in the task allocation is exploited at the analytical level in order to derive a direct solution to the schedulability problem.

Given the OpenMP-DAG derived in Section 3.1, we build upon the result in [18] to derive response-time bounds for untied tasks, by considering that each task *part* represents a sequence of operations that can be executed in one of the available threads as soon as all its three types of dependencies have been fulfilled (control flow, *TSP* creation/resume and *TSP* synchronization). Whenever more parts than available threads are ready to be executed, we assume any possible allocation order is possible, provided that the scheduling strategy remains work-conserving. This is the case of BFS and WFS strategies.

We now derive an upper-bound on the response-time of an OpenMP program composed of untied tasks and represented as an OpenMP-DAG G. Such a bound can be computed starting from the proof of the $2 - \frac{1}{m}$ approximation bound in [18], in conjunction with some additional considerations. Here, we first establish two lower-bounds on the minimum makespan R^{opt} of an OpenMP program, which will be useful to derive an upper-bound on its response-time.

Proposition 1.

$$R^{opt} \geq \frac{1}{m} \sum_{\upsilon_i \in V} c_i = \frac{1}{m} \mathrm{vol}(G). \tag{1}$$

Proposition 2.

$$R^{opt} \geq \max_{\lambda \in G} \sum_{\upsilon_i \in \lambda} c_i = \mathrm{len}(G). \tag{2}$$

Equation (1) trivially follows from the fact that the total amount of work should be executed on m threads, while Equation (2) is obtained by noticing that parts belonging to a chain must be executed sequentially. This is true for any chain of the OpenMP-DAG, and in particular for its longest one, i.e., its critical path.

We now review the proof in [18] to derive the approximation bound of *List Scheduling* on the minimum makespan of a generic set of precedence-constrained jobs (parts), which applies to OpenMP-DAGs with untied tasks as well.

Theorem 1. *Graham's List Scheduling algorithm gives a* $2 - \frac{1}{m}$ *approximation for the makespan minimization problem of a set of precedence-constrained jobs (or parts) expressed by means of a task graph G, scheduled on* m *identical processors (or threads).*

Proof. Let υ_z be the job in G that completes last, and t_z its starting time. Let υ_{z-1} be the predecessor of υ_z that completes last. By the precedence relation between the two jobs, we have that $t_z \geq t_{z-1} + c_{z-1}$. Proceeding in this way until a job without predecessors is reached, we construct a particular chain of jobs $\lambda^* = (\upsilon_1, \ldots, \upsilon_z)$. The fundamental observation that must be made is that, between the completion time $t_i + c_i$ of each job of λ^* and the starting time of the next job, all threads must be busy, otherwise job υ_{i+1} would have started earlier. The same applies to the time interval between 0 and t_1. Note also that some job belonging to λ^* is executing at every time instant when not all the threads are busy.

The response-time R of the OpenMP-DAG is given by the sum of the time instants when some of the threads are idle and the

[3] The *makespan* of a set of precedence constrained jobs is defined as the total length of the schedule (i.e., response-time) of the collection of jobs.

time instants when all the threads are busy. The former contribution cannot exceed len(λ^*), while the latter cannot exceed $\frac{1}{m}(\mathrm{vol}(G) - \mathrm{len}(\lambda^*))$, since the total amount of workload executed in such time slots is no more than $\mathrm{vol}(G) - \mathrm{len}(\lambda^*)$. Hence,

$$R \leq \mathrm{len}(\lambda^*) + \frac{1}{m}\left(\mathrm{vol}(G) - \mathrm{len}(\lambda^*)\right). \tag{3}$$

Now, by combining Equations (1), (2) and (3) and rephrasing the terms, we obtain:

$$\begin{aligned} R &\leq \mathrm{len}(\lambda^*) + \frac{1}{m}\left(\mathrm{vol}(G) - \mathrm{len}(\lambda^*)\right) = \\ &= \mathrm{len}(\lambda^*) + \frac{1}{m}\mathrm{vol}(G) - \frac{1}{m}\mathrm{len}(\lambda^*) \leq \\ &\leq R^{opt} + R^{opt} - \frac{1}{m}R^{opt} = \\ &= \left(1 - \frac{1}{m} + 1\right) R^{opt} = \\ &= \left(2 - \frac{1}{m}\right) R^{opt}. \end{aligned}$$

□

Equation (3) cannot be directly used as an upper-bound to the response-time of the OpenMP-DAG, because the chain λ^* is not known a priori. However, a simple upper-bound can be found from Equation (3) by upper-bounding the length of the chain λ^* with the critical path length of the task graph, as it is longer than any possible chain in the OpenMP-DAG. The following lemma formalizes this result.

Lemma 1. *An upper-bound on the response-time of an OpenMP-DAG composed of untied tasks is given by R^{ub}:*

$$R^{ub} = \mathrm{len}(G) + \frac{1}{m}\left(\mathrm{vol}(G) - \mathrm{len}(G)\right). \tag{4}$$

Proof. The upper-bound R^{ub} simply follows from Equation (3) by definition of critical path and by considering that $1 \geq \frac{1}{m}$. More explicitly:

$$\begin{aligned} R &\leq \mathrm{len}(\lambda^*) + \frac{1}{m}\left(\mathrm{vol}(G) - \mathrm{len}(\lambda^*)\right) = \\ &= \left(1 - \frac{1}{m}\right)\mathrm{len}(\lambda^*) + \frac{1}{m}\mathrm{vol}(G) \leq \\ &\leq \mathrm{len}(G) + \frac{1}{m}\left(\mathrm{vol}(G) - \mathrm{len}(G)\right). \end{aligned}$$

□

The result of Lemma 1 suggests that, whenever an OpenMP4 program is composed of untied tasks, a timing analysis can be easily performed by checking Equation (4) against the relative deadline D of the OpenMP-DAG.

5. Impact of Tied Tasks on Scheduling

When the OpenMP-DAG comprises tied tasks, the timing analysis presents some conceptual difficulties that significantly affect the complexity of the schedulability problem.

Tied tasks are constrained by *TSC 2*, which reduces the number of available threads for the execution of new tied tasks, and by the fact that tied tasks must always resume on the same thread where they started executing. Overall, these two constraints impact both performance and timing predictability.

5.1 Reduction of available threads

This section analyzes the implications of using tied tasks from a schedulability point of view. In particular, we compute the number of threads available to a new task due to *TSC 2* (Section 5.1.1) and the number of tasks that can prevent another task from resuming its execution in its thread (Section 5.1.2). In this way, we demonstrate that *tied* task execution model results in a non-conserving policy and explain why analyzing tied tasks without introducing unacceptable pessimism is prohibitive, or at least conceptually very difficult to achieve.

The following sections analyze these two scenarios assuming a generic scheduler (GenS) in which no concrete scheduling policy has been considered, and the BFS and WFS strategies with FIFO policies (see Section 2.3.3). Notice that the possible scheduling solutions derived by BFS and WFS strategies are included in GenS.

5.1.1 New tied tasks

The number of available threads for a new tied task may be reduced because other *tied* tasks suspended in a *TSP* may prevent the new tied task from being scheduled in the same thread. According to *TSC 2*, the new tied task can be scheduled to a thread in which other tied tasks are suspended only if it is a descendant of all the tasks tied to this thread. In the extreme case, a new tied task could even not start its execution despite existing available threads in the team.

We consider basic notions of set theory to derive the number of tasks affecting the effective number of threads available to new tied tasks. Concretely, we define $BlockCT_i(GenS)$ as the set of potential tasks that may prevent task T_i from executing on the same threads in which they are suspended considering a GenS strategy:

$$\begin{aligned} BlockCT_i(GenS) =& (T \setminus DesT_i \setminus PreT_i \setminus DDepT_i \setminus \{T_i\}) \\ & \cap TSPT, \end{aligned} \tag{5}$$

where T is the set of all tasks, $DesT_i$ is the set of descendant tasks of T_i, $PreT_i$ is the set of predecessor tasks of T_i, $DDepT_i$ is the set of tasks having a data dependency relationship with T_i and $TSPT$ is the set of tasks with at least one *TSP* that can suspend their execution (e.g. contain a `task` or a `taskwait` construct). The data dependency relationship in $DDepT_i$ considers tasks with data dependencies through `depend` clauses and also their child tasks if a synchronization dependency exits (e.g. a `taskwait`).

In other words, $BlockCT_i$ contains the sibling tasks of T_i and their descendant tasks that do not depend on T_i and that can be suspended in a *TSP*. It is important to remark that the descendant tasks of T_i have not been created yet at the point T_i is created, hence we can neglect them. Similarly, the dependent tasks of T_i and their descendant tasks are not considered because they have to wait until T_i has finished in order to start executing. Also, the predecessor tasks of T_i can be neglected because, according to *TSC 2*, T_i can be scheduled in the threads of all its predecessor tasks.

In the case of BFS strategy, $BlockCT_i(BFS)$ if defined as $BlockCT_i(GenS)$ removing the tasks that start executing after T_i (due to the FIFO policy):

$$\begin{aligned} BlockCT_i(BFS) = & BlockCT_i(GenS) \setminus SAftT_i = \\ & =((T \setminus DesT_i \setminus PreT_i \setminus DDepT_i \setminus \{T_i\}) \\ & \quad \cap TSPT_{bfs}) \setminus SAftT_i, \end{aligned} \tag{6}$$

where $BlockCT_i(GenS)$ is the set defined in Equation (5) and $SAftT_i$ is the set of sibling (and their descendant) tasks starting their execution after T_i according to the FIFO policy. This set includes the tasks for which the execution order can be defined. For example, in Figure 3 we can ensure that T_5 starts executing

Table 1: $DesT_i$, $PreT_i$, $DDepT_i$ and $SAftCT_i$ sets for each task T_i in Figure 1.

T_i	$DesT_i$	$PreT_i$	$DDepT_i$	$SAftCT_i$
T_0	$\{T_1..., T_9\}$	$\emptyset$	$\emptyset$	$\emptyset$
T_1	$\{T_2\}$	$\{T_0\}$	$\{T_3, T_4\}$	$\{T_3..., T_9\}$
T_2	$\emptyset$	$\{T_0, T_1\}$	$\{T_3, T_4\}$	$\emptyset$
T_3	$\{T_4\}$	$\{T_0\}$	$\{T_1, T_2\}$	$\emptyset$
T_4	$\emptyset$	$\{T_0, T_3\}$	$\{T_1, T_2\}$	$\emptyset$
T_5	$\{T_6, T_7, T_8, T_9\}$	$\{T_0\}$	$\emptyset$	$\emptyset$
T_6	$\{T_7, T_8, T_9\}$	$\{T_0, T_5\}$	$\emptyset$	$\emptyset$
T_7	$\{T_8\}$	$\{T_0, T_5, T_6\}$	$\emptyset$	$\{T_9\}$
T_8	$\emptyset$	$\{T_0, T_5, T_6, T_7\}$	$\emptyset$	$\emptyset$
T_9	$\emptyset$	$\{T_0, T_5, T_6\}$	$\emptyset$	$\emptyset$

after T_1, but we do not know whether T_2 will start or not after T_5. However, we cannot ensure that T_3 is executed before T_5 despite the BFS FIFO policy, because T_3 depends on T_1 and so T_5 may start executing before T_1 finishes.

It is important to notice that the set $TSPT$ contains different elements depending on the task scheduling policy. In the case of BFS ($TSPT_{bfs}$), task creation *TSPs* are not considered in this set because the parent task is not suspended when it creates a child task, but rather it continues its execution in the same thread.

Finally, in case of the WFS strategy, the set $BlockCT_i(WFS)$ is empty, because all tasks T_i start executing immediately after their creation in the parent task thread:

$$BlockCT_i(WFS) = \emptyset. \tag{7}$$

Table 1 shows, for each task T_i in Figure 1, the sets $DesT_i$, $PreT_i$, $DDepT_i$ and $SAftT_i$, required to calculate $BlockCT_i$ for GenS, BFS and WFS, and shown in Table 2. Moreover, T is equal to $\{T_0, T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8, T_9\}$, $TSPT_{gens}$ is equal to $\{T_0, T_1, T_3, T_5, T_6, T_7\}$, $TSPT_{bfs}$ is equal to $\{T_0, T_1, T_3, T_5, T_7\}$, and $TSPT_{wfs}$ is equal to $TSPT_{gens}$.

As an example, given T_1, $DesT_1$ is equal to $\{T_2\}$ because T_1 creates T_2, $PreT_1$ is equal to $\{T_0\}$ because T_0 creates T_1, and $DDepT_1$ is equal to $\{T_3, T_4\}$ because T_3 and its descendant task (due to the `taskwait` in T_3) have a data dependency relationship with T_1. As a result, $BlockCT_1(GenS)$ is equal to $\{T_5, T_6, T_7\}$ and so these tasks can suspend their execution and block a thread that T_1 could not use. However, $BlockCT_1(BFS)$ is equal to $\emptyset$ because, according to the BFS policy, T_5, T_6 and T_7 are created after T_1 ($\{T_5, T_6, T_7\} \in SAftCT_i$) and T_6 never suspends its execution ($T_6 \notin TSPT_{bfs}$). Finally, $BlockCT_1(WFS)$ is equal to $\emptyset$ by definition.

Within the set $BlockCT_i$ only tasks that can be executed in parallel have to be considered at the same time. That is, if $BlockCT_i = \{T_j, T_{j+1}\}$ but T_{j+1} depends on T_j, and so T_{j+1} and T_j will never execute in parallel, then $BlockCT_i = \{T_j\}$ or $BlockCT_i = \{T_{j+1}\}$. This is the case of task T_5. Given the situation in which T_3 is suspended in the `taskwait`, waiting for the task T_4 to finish, T_5 could not use T_3's thread because it is not a descendant of it. Similarly, T_1 could block a thread of T_5 if it is suspended waiting for T_2. However, T_1 and T_3 could not subtract a thread to T_5 at the same time because they are executed sequentially and therefore, in the case of $GenS$ or BFS, $BlockCT_5 = \{T_1\}$ or $BlockCT_5 = \{T_3\}$.

The cardinality[4] of each set $BlockCT_i$ determines the maximum number of tasks that may block threads which T_i could not use at its creation time due to *TSC 2*.

Table 2: Given the example in Figure 1, tasks that may block threads for a new task at creation time.

			$BlockCT_i$
T_i	$BlockCT_i(GenS)$	$BlockCT_i(BFS)$	(WFS)
T_0	$\emptyset$	$\emptyset$	$\emptyset$
T_1	$\{T_5, T_6, T_7\}$	$\emptyset$	$\emptyset$
T_2	$\{T_5, T_6, T_7\}$	$\{T_5, T_7\}$	$\emptyset$
T_3	$\{T_5, T_6, T_7\}$	$\{T_5, T_7\}$	$\emptyset$
T_4	$\{T_5, T_6, T_7\}$	$\{T_5, T_7\}$	$\emptyset$
T_5	$\{T_1\}$ *or* $\{T_3\}$	$\{T_1\}$ *or* $\{T_3\}$	$\emptyset$
T_6	$\{T_1\}$ *or* $\{T_3\}$	$\{T_1\}$ *or* $\{T_3\}$	$\emptyset$
T_7	$\{T_1\}$ *or* $\{T_3\}$	$\{T_1\}$ *or* $\{T_3\}$	$\emptyset$
T_8	$\{T_1\}$ *or* $\{T_3\}$	$\{T_1\}$ *or* $\{T_3\}$	$\emptyset$
T_9	$\{T_1, T_7\}$ *or* $\{T_3, T_7\}$	$\{T_1, T_7\}$ *or* $\{T_3, T_7\}$	$\emptyset$

5.1.2 At resumption time

When a suspended *tied* task wants to resume, it may not restart its execution even if there are idle available threads, because the thread the task is tied to is executing another task (it is important to remark that a task can only be suspended if it contains a *TSP*). This situation occurs when a task has been suspended in a *TSP* and, at resumption time, another (predecessor or descendant) task or an *untied* task is executing in the thread. There may be other idle threads but the task cannot resume its execution because it is tied to its thread.

We define $BlockRT_i(GenS)$ as the set of potential tasks that may prevent (block) task $T_i \in TSPT$ from resuming its execution in the thread to which T_i is tied, assuming GenS strategy.

$$BlockRT_i(GenS) = DesT_i \cup PreT_i \cup uT, \tag{8}$$

where $DesT_i$ is the set of descendant tasks of T_i, $PreT_i$ is the set of predecessor tasks of T_i and uT is the set of untied tasks.

This set contains predecessor and descendant tasks of T_i and all the untied tasks because, due to *TSC 2*, they are the only ones that can be scheduled in T_i's thread, and therefore can prevent T_i to be resumed.

In the case of BFS strategy, $BlockRT_i(BFS)$ is defined as:

$$BlockRT_i(BFS) = (DesT_i \setminus TSPDepT_i) \cup PreT_i \cup uT, \tag{9}$$

where $TSPDepT_i$ is the set of tasks having a synchronization dependency with T_i through any of its *TSP*. With respect to $PreT_i$, which is the set of predecessor tasks, T_i's parent task is included in it only if it contains a `taskyield` *TSP*, as in this case T_i's parent task could block T_i's thread. Otherwise, if T_i's parent task is suspended in any other *TSP*, e.g. a `taskwait`, it cannot resume its execution as it is blocked until T_i finishes and so, it cannot block T_i's thread.

In the case of WFS strategy, $BlockRT_i(WFS)$ is equal to $BlockRT_i(GenS)$:

$$BlockRT_i(WFS) = BlockRT_i(GenS) = DesT_i \cup PreT_i \cup uT. \tag{10}$$

As previously noted, WFS is particularly affected when *tied* tasks are implemented, because the parallel execution turns into a sequential execution. When any T_i is created, it starts its execution in the parent's thread. The parent task is suspended and it cannot resume its execution in another thread because it is tied to T_i's thread. On the contrary, if T_i is suspended in a *TSP* (not a task creation) and T_i's parent task resumes its execution, the thread T_i is blocked because of its parent.

Table 3 shows the sets $TSPDepT_i$ for each $T_i \in TSPT$ in Figure 1. Based on this and on $PreT_i$ and $DesT_i$ shown in Table 1, the sets $BlockRT_i$ considering GenS, BFS and WFS strategies have been calculated and are shown in Table 4. Similar to the $BlockCT_i$, the set $TSPT$ is different for BFS because the task

[4] The cardinality of a set A, expressed as $|A|$, is a measure of the number of elements of the set.

Table 3: $TSPDepT_i$ set for each task $T_i \in TSPT$ in Figure 1.

T_i	$TSPDepT_i$
T_0	$\{T_1, T_2, T_3, T_4, T_5, T_6\}$
T_1	$\{T_2\}$
T_3	$\{T_4\}$
T_5	$\{T_6\}$
T_6	–
T_7	$\{T_8\}$

Table 4: Given the example in Figure 1, tasks that may block threads for each task $T_i \in TSPT$ at resumption time.

T_i	$BlockRT_i(GenS)$	$BlockRT_i(BFS)$	$BlockRT_i(WFS)$
T_0	$\{T_1, ...T_9\}$	$\{T_7, T_8, T_9\}$	$\{T_1, ...T_9\}$
T_1	$\{T_0, T_2\}$	–	$\{T_0, T_2\}$
T_3	$\{T_0, T_4\}$	–	$\{T_0, T_4\}$
T_5	$\{T_0, T_6, T_7, T_8, T_9\}$	$\{T_0, T_7, T_8, T_9\}$	$\{T_0, T_6, T_7, T_8, T_9\}$
T_6	$\{T_0, T_5, T_7, T_8, T_9\}$	–	$\{T_0, T_5, T_7, T_8, T_9\}$
T_7	$\{T_0, T_5, T_6, T_8\}$	$\{T_0, T_5\}$	$\{T_0, T_5, T_6, T_8\}$

creation is not considered as a scheduling point for the task creating the new task.

Hence, given T_0, $BlockRT_0(GenS)$ contains all its descendant tasks because all of them can be scheduled in the same thread and prevent T_0 from resuming its execution. In case of $BlockRT_0(BFS)$, we analyze independently each *TSP* in which T_0 can be suspended, that is, at the `taskwait` at line 26 in Figure 1. Then, from $DesT_0$ we remove all the descendant tasks that have a synchronization dependency with this `taskwait`, i.e., tasks in $TSPDepT_0$: task T_1 (and recursively its child T_2 because it has another synchronization dependency with T_1), task T_3 (and similarly T_4) and task T_5 (and similarly T_6 but not its descendants T_7, T_8 and T_9 because there is no synchronization dependency with T_6). As a result, tasks T_7, T_8 and T_9 compose the set $BlockRT_0(BFS)$. Finally, for $BlockRT_0(WFS)$, T_0's thread will be blocked by its descendants $BlockRT_0(WFS) = \{T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8, T_9\}$.

5.2 Issues on the timing characterization of tied tasks

The reasoning about the computation of $BlockCT_i$ and $BlockRT_i$ suggests that deriving schedulability results when tied tasks are involved is extremely challenging, unless very pessimistic assumptions are made. More specifically, in Section 4 we have leveraged the work-conserving policy implied by the use of *untied* tasks to derive a timing analysis simply based on two quantities: (i) the critical path length of the entire task graph and (ii) the remaining interfering workload over m threads.

However, when considering the non-work-conserving scenario induced by tied tasks, deriving such an accurate analysis is not as easy, due to multiple reasons:

1. It is not correct to compute the critical path of the task graph as a whole, but rather a critical path reaching the end of each task in the OpenMP-DAG, since it is important to compute the different time offsets after which each task can start executing. In fact, since each task has its own descendant and precedence relationships, the corresponding $BlockCT_i$ and $BlockRT_i$ sets will be different, suggesting to carry out a *per-task* timing analysis.
2. The interference contribution for a tied task cannot be considered as evenly distributed. Specifically, it is necessary to differentiate the interference contribution before the task starts, which can be accounted for as evenly distributed on the threads being blocked due to $BlockCT_i$, and the interference suffered by the task at each of its *TSPs*, which includes the full contribution of the set of tasks $BlockRT_i$.
3. The critical path reaching the end of a task may comprehend parts of other tasks that can have different descendant relationships with respect to T_i, which makes really hard to identify which tasks may actually interfere T_i without introducing unacceptable pessimism in the analysis. In order to have an intuitive feeling of the problem, please consider again the example given in Figure 3, where all task parts have unitary WCETs. Here, task T_3 has a data dependency with T_1, hence it cannot start executing until T_1 has finished. When computing the critical path reaching the end of T_3, we immediately observe that it is not simply composed of tasks that are predecessors of T_3, but also by parts of T_1 and T_2 p_{10}, p_{11} and p_{20} (that are not predecessors of T_3). Hence, the interference imposed on critical task parts of T_3 cannot simply be estimated based on the descendant relationships of T_3 (i.e., by the knowledge of $BlockRT_3$), but should take into account those of all the tasks involved, which hugely complicates the analysis.
4. From the analytical point of view, computing an upper-bound on the response-time of a tied task T_i would require to assume the worst-case scenario in which all the tasks that can be suspended simultaneously at the creation point of T_i are indeed suspended, inhibiting T_i to execute on the corresponding threads tied to these tasks. Therefore, beside knowing the maximum number of tasks that could be suspended at the time of T_i's creation due to TSC 2 (i.e., the set $BlockCT_i$), we should provide an upper-bound on the maximum time the suspended tasks would take before being resumed.

Overall, the above considerations confirm that a timing analysis for tied tasks, besides being conceptually very difficult to achieve, would require to address sources of inherent complexity that would lead to unacceptably pessimistic response-time bounds. As a result, the makespan of the task graph may undergo large variations depending on the allocation of newly generated tasks, leading in few cases to resource under-utilization and undesirable idleness of some threads as shown in next section.

5.3 Platform Under-utilization

As previously observed, the use of tied tasks encompasses their suspension and resumption only by the same thread that first started their execution. This may lead to platform under-utilization reducing the number of threads working even if there are tasks ready to execute. We refer as m_i^* to the minimum number of threads available to task T_i at the time of its creation. Since not all threads may be available to a task when it is created, it follows that the interference suffered from other tasks cannot be considered to be evenly distributed across all threads, but only on $m_i^* \leq m$ threads.

Theorem 2. *The minimum value of m_i^* is 2, for any task graph comprising tied tasks.*

Proof. The statement can be demonstrated by the two following points: (i) providing a configuration where $m_i^* = 2$, and (ii) showing that no configuration can be produced with $0 \leq m_i^* < 2$.

(i) There exists a scenario where $m_i^* = 2$. Consider the OpenMP program illustrated in Figure 4. Suppose the program must be executed on $m = 4$ threads and that the allocation on the available threads is as shown in Figure 5(a). Tasks T_1, T_2 and T_3 must wait for their first-level descendants before terminating, due to the `taskwait` directives. Then, if task parts p_{04} and p_{40} have a very long WCET, there is a long time interval where T_5, T_6 and T_7 cannot execute on threads 2 and 3, although they are idle, due

```
#pragma omp parallel num_threads(N) {
#pragma omp single { // T0
  part00
  #pragma omp task { // T1
    part10
    #pragma omp task {  // T2
      part20
      #pragma omp task {  // T3
        part30
        #pragma omp task { part40 } // T4
        #pragma omp taskwait
        part31
      }
      #pragma omp taskwait
      part21
    }
    #pragma omp taskwait
    part11
  }
  part01
  #pragma omp task { part50 } // T5
  part02
  #pragma omp task { part60 } // T6
  part03
  #pragma omp task { part70 } // T7
  part04
}}
```

Figure 4: Example of an OpenMP program, pessimistic scheduling of tied tasks.

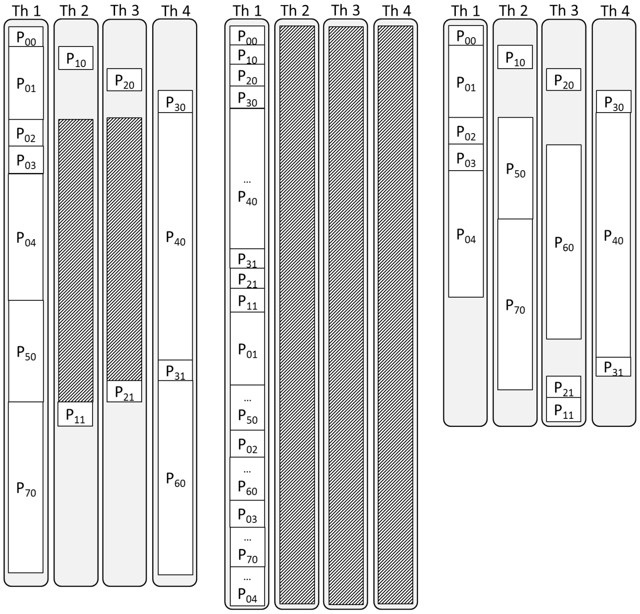
to *TSC 2*. T_5, T_6 and T_7 can only be scheduled in threads 1 and 4 which are used by tasks T_4 and T_0, respectively. Therefore, T_5, T_6 and T_7 cannot start their execution until they finish and such a time interval can be arbitrarily long depending on the WCET of task parts p_{04} and p_{40}.

(ii) There is no configuration such that $m_i^* = 0$ or 1. It cannot be $m_i^* = 0$ because this would mean that all m threads contain tasks simultaneously suspended in a *TSP*, but then none of them would make progress (i.e., a deadlock occurs). In this case, no new task can be created, hence the blocking due to *TSC 2* cannot be experienced.

Analogously, it cannot be $m_i^* = 1$. By contradiction, assume $m_i^* = 1$. This means that when task T_i is released, $m-1$ threads are not available to it due to *TSC 2*, i.e., $m-1$ threads are blocked by tasks that are not predecessors of T_i. Such $m-1$ tasks must be suspended in a *TSP*, and cannot continue executing because some of their synchronization constraints are not fulfilled. This can only happen when some task must wait for its first-level descendants, due to a `taskwait` or an if-false clause. The semantics of the latter constructs implies that there cannot be any synchronization arrow that traverses multiple levels: indeed, synchronization arrows can either connect siblings (belonging to the same level) in the case of data-dependency, or first-level descendants to their father, in the case of `taskwait` or if-false. From this reasoning, it follows that the $m-1$ tasks must belong to $m-1$ contiguous descendant levels $[l_x, l_{x+m-2}]$. Therefore, the task that generates T_i must belong to l_i, being either $i \leq x-1$ or $i \geq x+m-1$. In the case $i \leq x-1$, a contradiction is reached, because each of the m threads executes at least one task, but the task belonging to $x+m-2$ has no descendant, hence there is no reason why it should be suspended in a *TSP*. If instead $i \geq x+m-1$, then the task that generates T_i is descendant of all the other $m-1$ tasks, and the same holds for T_i. This facts also imply a contradiction because *TSC 2* comes into play only when the generated task is not descendant of the other ones. We conclude that there is no situation such that $m_i^* = 1$, proving the theorem. □

Therefore, we define m_i^* as:

$$m_i^* = max(2, m - |BlockCT_i|), \tag{11}$$



(a) BFS tied **(b) WFS tied** **(c) BFS untied**

Figure 5: (a) Pessimistic breadth-first scheduling example with tied tasks, (b) work-first scheduling (LIFO) with tied tasks and (c) breadth-first scheduling with untied tasks. Program in Figure 4.

where $|BlockCT_i|$ is the maximum number of tasks that may block threads which T_i could not use at its creation time due to *TSC 2*. As we consider all potential cases, this number of tasks can be greater than the total number of threads, m. Therefore, $m - |BlockCT_i|$ may be negative, but it is proven by Theorem 2 that the minimum value of m_i^* is 2. Hence, in this case, an accurate timing analysis should identify which tasks compose this subset in the worst-case, since only a subset of the tasks composing $BlockCT_i$ will subtract threads to the considered task. However, when tied tasks are involved, it is absolutely non-trivial to identify the scenario that maximizes the interference imposed on T_i. This is another subtle reason (in addition to those listed in Section 5.2) that explains why devising a timing analysis for tied tasks is so difficult.

Figure 5 illustrates a case of resource under-utilization implied by the use of tied tasks, as opposed to the untied case. In particular, Figure 5a shows a possible scheduling of the OpenMP program in Figure 4, considering BFS: if all the nested tasks are scheduled in different threads before T_5, T_6 and T_7, and being $part_{04}$ and $part_{40}$ very time-consuming, then the execution of tasks T_5, T_6 and T_7 is postponed even if threads 2 and 3 are idle (striped areas) but tied to tasks T_1 and T_2. Figure 5b shows the scheduling considering WFS (LIFO): as already noted, WFS turns into a sequential execution when implementing tied tasks. Notice that in this figure task parts $p_{40}, p_{50}, p_{60}, p_{70}$ and p_{04} are less time-consuming only for the sake of space-saving. If the clause `untied` is added to all the tasks in the program of Figure 4, we observe that the breadth-first scheduling of these untied tasks, illustrated in Figure 5c, determines no platform under-utilization beyond program limitations. WFS will result in a similar scheduling for untied tasks.

6. Related work

The OpenMP language committee presented in [10] a comparison between the thread-centric and the task-centric models, exposing the design choices done in the new tasking model due to conflicts with the thread-centric model. These decisions include the definition of *tied* and *untied* tasks and some others related to the data-sharing and the scheduling. However, the paper does not take time predictability into account.

In [3], authors performed an evaluation of different scheduling policies using their run-time system Nanos++ [22] and analyzed the differences existing between *tied* and *untied* tasks for an average performance point of view.

The first attempt to apply OpenMP4 has been recently introduced in [19], where the authors studied how to construct an OpenMP task graph which contains enough information to allow the application of real-time DAG scheduling models, from which timing guarantees can be derived, considering the tasking semantics of OpenMP4.

OpenMP has been already considered as a convenient interface to describe real-time applications to deal with parallel task models in multiprocessor systems. The earliest parallel task model to be proposed is the *fork-join* model [14], where each task is represented as an alternating sequence of sequential and parallel segments and no nested parallelism is allowed. Later, this model has been generalized by the *synchronous parallel* model [5] [8], which allows consecutive parallel segments with arbitrary degree of parallelism. Still, synchronization is enforced at every segment's boundary. As a further generalization of the previously mentioned task models, the *sporadic DAG model* represents a task as a directed acyclic graph, where each node is a sequential job, and edges represent precedence constraints between jobs [21] [20] [5] [7] [13].

Unfortunately, besides the increasing expressiveness provided by existing real-time parallel task models, all of them neglect the real semantics of the OpenMP execution model and bypass the functionality of the runtime system. Moreover, the focus has typically been on the thread-centric model from OpenMP specification v2.5, which is limited to a standard *fork-join* type of parallelism and does not take advantage of the way more expressive *tasking* interface. The purpose of this work, instead, is to demonstrate that OpenMP tasking model can be applied to real-time systems if work-conserving schedulers, such as BFS and WFS, are used.

Hence, to our knowledge, this work represents the first attempt to provide an accurate timing characterization of a real and prominent parallel programming model such as OpenMP.

7. Conclusions

This paper analyses from a timing perspective the two tasking execution models existing in OpenMP4, *tied* and *untied*. The existence of these two models results from the coexistence of the thread-centric and task-centric models for backward compatibility reasons.

The considerations drawn in this paper suggest that using tied tasks inside time-critical applications is not recommendable, because of the inherent pessimism that underlies the timing analysis of such tasks and the conceptual difficulties behind the construction of an accurate schedulability test.

On the other hand, we have shown that a simple schedulability analysis of OpenMP programs is possible whenever untied tasks are involved. This definitely suggests that the use of untied tasks would be preferable for parallel applications in the real-time context, since it would permit to exploit a parallel execution model in a predictable way. Overall, this paper demonstrates that OpenMP4 can be applied to real-time systems if untied tasking model is adopted.

8. Acknowledgments

This work was supported by EU project P-SOCRATES (FP7-ICT-2013-10) and by Spanish Ministry of Science and Innovation grant TIN2012-34557.

References

[1] OpenMP Application Program Interface, Version 3.1. July 2011.

[2] OpenMP Application Program Interface, Version 4.0. October 2013.

[3] A. Duran, et. al. Evaluation of OpenMP Task Scheduling Strategies. In *4th International Workshop on OpenMP*, 2008.

[4] A. Marongiu, et. al. Improving the programmability of STHORM-based heterogeneous systems with offload-enabled OpenMP. In *First International Workshop on Many-core Embedded Systems (MES)*, 2013.

[5] A. Saifullah, et. al. Multi-core real-time scheduling for generalized parallel task models. *Real-Time Systems*, 49(4):404–435, 2013.

[6] B. Chapman, et. al. Implementing OpenMP on a high performance embedded multicore MPSoC. In *IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, 2009.

[7] S. Baruah. Improved multiprocessor global schedulability analysis of sporadic dag task systems. In *26th Euromicro Conference on Real-Time Systems (ECRTS)*, 2014.

[8] C. Maia, et. al. Response-time analysis of synchronous parallel tasks in multiprocessor systems. In *22nd International Conference on Real-Time Networks and Systems (RTNS)*, 2014.

[9] C. Wang, et. al. libEOMP: A Portable OpenMP Runtime Library Based on MCA APIs for Embedded Systems. In *International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM)*, 2013.

[10] E. Ayguade, et. al. The Design of OpenMP Tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418, March 2009.

[11] E. Stotzer, et. al. OpenMP on the Low-Power TI Keystone II ARM/DSP System-on-Chip. In *OpenMP in the Era of Low Power Devices and Accelerators*, volume 8122, pages 114–127. Springer Berlin Heidelberg, 2013.

[12] J. K. Lenstra and A. H. G. Rinnooy Kan. Complexity of scheduling under precedence constraints. *Operations Research*, 26(1):22–35, 1978.

[13] Jing Li, et. al. Outstanding Paper Award: Analysis of Global EDF for Parallel Tasks. In *25th Euromicro Conference on Real-Time Systems (ECRTS)*, 2013.

[14] K. Lakshmanan, et. al. Scheduling parallel real-time tasks on multi-core processors. In *31st IEEE Real-Time Systems Symposium (RTSS)*, 2010.

[15] M. Frigo, et. al. The implementation of the Cilk-5 multithreaded language. In *ACM SIGPLAN Notices*, volume 33, pages 212–223. ACM, 1998.

[16] G. J. Narlikar. Scheduling threads for low space requirement and good locality. *Theory of Computing Systems*, 35(2):151–187, 2002.

[17] P. Burgio, et. al. Enabling fine-grained OpenMP tasking on tightly-coupled shared memory clusters. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2013.

[18] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM journal on Applied Mathematics*, 17(2):416–429, 1969.

[19] R. Vargas, et. al. OpenMP and Timing Predictability: A Possible Union? In *18th Design, Automation and Test in Europe Conference (DATE)*, 2015.

[20] S. Baruah, et. al. A Generalized Parallel Task Model for Recurrent Real-time Processes. In *IEEE Real-Time Systems Symposium (RTSS)*, 2012.

[21] V. Bonifaci, et. al. Feasibility Analysis in the Sporadic DAG Task Model. In *25th Euromicro Conference on Real-Time Systems (ECRTS)*, 2013.

[22] X. Teruel, et. al. Support for OpenMP tasks in Nanos v4. In *Proceedings of the 2007 conference of the center for advanced studies on Collaborative research*. IBM Corp., 2007.