3

Predictable Parallel Programming with OpenMP

Maria A. Serrano[1], Sara Royuela[1], Andrea Marongiu[2] and Eduardo Quiñones[1]

[1]Barcelona Supercomputing Center (BSC), Spain
[2]Swiss Federal Institute of Technology in Zürich (ETHZ), Switzerland; and University of Bologna, Italy

This chapter motivates the use of the OpenMP (Open Multi-Processing) parallel programming model to develop future critical real-time embedded systems, and analyzes the time-predictable properties of the OpenMP tasking model. Moreover, this chapter presents the set of compiler techniques needed to extract the timing information of an OpenMP program in the form of an *OpenMP Direct Acyclic Graph* or OpenMP-DAG.

3.1 Introduction

Parallel programming models are key to increase the productivity of parallel software from three different angles:

1. From a *programmability* angle, parallel programming models provide developers with the abstraction level required to program parallel applications while hiding processor complexities.
2. From a *portability* angle, platform-independent parallel programming models allow executing the same parallel source code in different parallel platforms.
3. From a *performance* angle, different levels of abstraction allow for a fine-tuned parallelism, i.e., users may either squeeze the capabilities of a specific architecture using the language capabilities, or rely on runtime mechanisms to dynamically exploit parallelism.

Hence, parallel programming models are of paramount importance to exploit the massive computation capabilities of state-of-the-art and future parallel and heterogeneous processor architectures. Several approaches coexist with such a goal, and these can be grouped as follows [1]:

- *Hardware-centric models* aim to replace the native platform programming with higher-level, user-friendly solutions, e.g., Intel® TBB [2] and NVIDIA® CUDA [3]. These models focus on tuning an application to match a chosen platform, which makes their use neither a scalable nor a portable solution.
- *Application-centric models* deal with the application parallelization from design to implementation, e.g., OpenCL [4]. Although portable, these models may require a full rewriting process to accomplish productivity.
- *Parallelism-centric models* allow users to express typical parallelism constructs in a simple and effective way, and at various levels of abstraction, e.g., POSIX threads [6] and OpenMP [7]. This approach allows flexibility and expressiveness, while decoupling design from implementation.

Considering the vast amount of parallel programming models available, there is a noticeable need to unify programming models to exploit the performance benefits of parallel and heterogeneous architectures [9]. In that sense, OpenMP has proved many advantages over its competitors to enhance productivity. The next sections introduce the main characteristics of the most relevant programming models, and conclude with an analysis of the main benefits of OpenMP.

3.1.1 Introduction to Parallel Programming Models

The multitude of parallel programming models currently existing makes it difficult to choose the language that better fits the needs of each particular case. Table 3.1 introduces the main characteristics of the most relevant programming models in critical embedded systems. The features considered are the following: performance (based on throughput, bandwidth, and other metrics), portability (based on how straight-forward it is to migrate to different environments), heterogeneity (based on the support for cross-platform applications), parallelism (based on the support provided for data-based and task-based parallelism), programmability (based on how easy it is for programmers to get the best results), and flexibility (based on the features for parallelizing offered in the language).

Table 3.1 Parallel programming models comparison

	Pthreads	OpenCL	CUDA	Cilk Plus	TBB	OpenMP
Performance	✓	✓	✓✓	✓✓	✓	✓
Portability	✓	✓	×	×	×	✓✓
Heterogeneity	×	✓	✓	×	✓	✓✓
Parallelism	data/task	data/task	data	data/task	task	data/task
Programmability	×	×	×	✓✓	✓	✓
Flexibility	✓	✓	×	×	✓	✓✓

3.1.1.1 POSIX threads

POSIX threads (Portable Operating System Interface for UNIX threads), usually referred to as Pthreads, is a standard C language programming interface for UNIX systems. The language provides efficient light-weight mechanisms for thread management and synchronization, including mutual exclusion and barriers.

In a context where hardware vendors used to implement their own proprietary versions of threads, Pthreads arose with the aim of enhancing the portability of threaded applications that reside on shared memory platforms. However, Pthreads results in very poor programmability, due to the low-level threading model provided by the standard, that leaves most of the implementation details to the programmer (e.g., work-load partitioning, worker management, communication, synchronization, and task mapping). Overall, the task of developing applications with Pthreads is very hard.

3.1.1.2 OpenCL^{TM}

OpenCL^{TM} (Open Computing Language) is an open low-level application programming interface (API) for cross-platform parallel computing that runs on heterogeneous systems including multicore and manycore CPUs, GPUs, DSPs, and FPGAs. There are two different actors in an OpenCL system: the host and the devices. The language specifies a programming language based on C99 used to control the host, and a standard interface for parallel computing, which exploits task-based and data-based parallelism, used to control the devices.

OpenCL can run in a large variety of devices, which makes portability its most valuable characteristic. However, the use of vendor-specific features may prevent this portability, and codes are not guaranteed to be optimal due to the important differences between devices. Furthermore, the language has an important drawback: it is significantly difficult to learn, affecting the programmability.

3.1.1.3 NVIDIA® CUDA

NVIDIA® CUDA is a parallel computing platform and API for exploiting CUDA-enabled GPUs for general-purpose processing. The platform provides a layer that gives direct access to the GPU's instruction set, and is accessible through CUDA-accelerated libraries, compiler directives (such as OpenACC [10]), and extensions to industry-standard programming languages (such as C and C++).

The language provides dramatic increases of performance when exploiting parallelism in GPGPUs. However, its use is limited to CUDA-enabled GPUs, which are produced only by NVIDIA®. Furthermore, tuning applications with CUDA may be hard because it requires rewriting all the offloaded kernels and knowing the specifics of each platform to get the best results.

3.1.1.4 Intel® Cilk™ Plus

Intel® Cilk Plus [11] is an extension to C/C++ based on Cilk++ [12] that has become popular because of its simplicity and high level of abstraction. The language provides support for both data and task parallelism, and provides a framework that optimizes load balance, implementing a work-stealing mechanism to execute tasks [13].

The language provides a simple yet efficient platform for implementing parallelism. Nonetheless, portability is very limited because only Intel® and GCC implement support for the language extensions defined by Cilk Plus. Furthermore, the possibilities available with this language are limited to tasks (*_cilk_spawn*, *_cilk_sync*), loops (*_cilk_for*), and reductions (*reducers*).

3.1.1.5 Intel® TBB

Intel® TBB is an object-oriented C++ template library for implementing task-based parallelism. The language offers constructs for parallel loops, reductions, scans, and pipeline parallelism. The framework provided has two key components: (1) compilers, which optimize the language templates enabling a low-overhead form of polymorphism, and (2) runtimes, which keep temporal locality by implementing a queue of tasks for each worker, and balance workload across available cores by implementing a work-stealing policy.

TBB offers a high level of abstraction in front of complicated low-level APIs. However, adapting the code to fit the library templates can be arduous. Furthermore, portability is limited, although the last releases support Visual C++, Intel® C++ compiler, and the GNU compiler collection.

3.1.1.6 OpenMP

OpenMP, the de-facto standard parallel programming model for shared memory architectures in the high-performance computing (HPC) domain, is increasingly adopted also in embedded systems. The language was originally focused on a thread-centric model to exploit massive data-parallelism and loop intensive applications. However, the latest specifications of OpenMP have evolved to a task-centric model that enables very sophisticated types of fine-grained and irregular parallelism, and also include a host-centric accelerator model that enables an efficient exploitation of heterogeneous systems. As a matter of fact, OpenMP is supported in the SDK of many of the state-of-the-art parallel and heterogeneous embedded processor architectures, e.g., Kalray MPPA [14], and TI Keystone II [16].

Different evaluations demonstrate that OpenMP delivers tantamount performance and efficiency compared to highly tunable models such as TBB [17], CUDA [18] and OpenCL [19]. Moreover, OpenMP has different advantages over low-level libraries such as Pthreads: on one hand, it offers robustness without sacrificing performance [21] and, on the other hand, OpenMP does not lock the software to a specific number of threads. Another important advantage is that the code can be compiled as a single-threaded application just disabling support for OpenMP, thus easing debugging and so programmability.

Overall, the use of OpenMP presents three main advantages. First, an expert community has constantly reviewed and augmented the language for the past 20 years. Second, OpenMP is widely implemented by several chip and compiler vendors from both high-performance and embedded computing domains (e.g., GNU, Intel®, ARM, Texas Instruments and IBM), increasing portability among multiple platforms from different vendors. Third, OpenMP provides greater expressiveness due to years of experience in its development; the language offers several directives for parallelization and fine-grained synchronization, along with a large number of clauses that allow it to contextualize concurrency and heterogeneity, providing fine control of the parallelism.

3.2 The OpenMP Parallel Programming Model

3.2.1 Introduction and Evolution of OpenMP

OpenMP represents the computing resources of a parallel processor architecture (i.e., cores) by means of high-level threads, named *OpenMP threads*, upon which programmers can assign units of code to be executed. During the execution of the program, the OpenMP runtime assigns these threads to low-level computing resources, i.e., the operating system (OS) threads, which are then assigned to physical cores by the OS scheduler, following the execution model defined by the OpenMP directives. Figure 3.1 shows a schematic view of the stack of components involved in the execution of an OpenMP program. OpenMP exposes some aspects of managing OpenMP threads to the user (e.g., defining the number of OpenMP threads assigned to a parallel execution by means of the num_threads clause). The rest of components are transparent to the user and efficiently managed by the OpenMP runtime and the OS.

Originally, up to OpenMP version 2.5 [22], OpenMP was traditionally focused on massively data-parallel, loop-intensive applications, following the *single-program-multiple-data* programming paradigm. In this model, known as *thread model*, OpenMP threads are visible to the programmer, which are controlled with work-sharing constructs that assign iterations of a loop or code segments to OpenMP threads.

The OpenMP 3.0 specification [23] introduced the concept of tasks by means of the task directive, which exposes a higher level of abstraction to programmers. A task is an independent parallel unit of work, which defines an instance of code and its data environment. This new model, known as *tasking model*, provides a very convenient abstraction of parallelism as it is the runtime (and not the programmer) the responsible for scheduling tasks to threads.

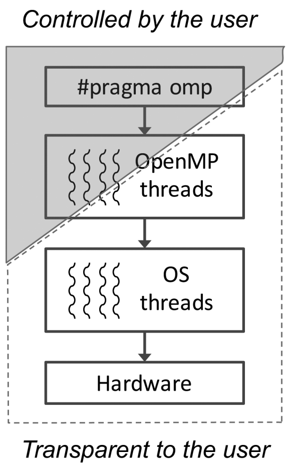


Figure 3.1 OpenMP components stack.

With version 4.0 of the specification [24], OpenMP evolved to consider very sophisticated types of fine-grained, irregular and highly unstructured parallelism, with mature support to express dependences among tasks. Moreover, it incorporated for the first time a new accelerator model including features for offloading computation and performing data transfers between the host and one or more accelerator devices. The latest version, OpenMP 4.5 [25], enhances the previous accelerator model by coupling it with the tasking model.

Figure 3.2 shows a time-line of all existent releases of OpenMP, since 1997, when the OpenMP Architecture Review Board (ARB) was formed. The next version, 5.0 [26–28], is planned for November 2018.

3.2.2 Parallel Model of OpenMP

This section provides a brief description of the OpenMP parallel programming model as defined in the latest specification, version 4.5.

3.2.2.1 Execution model

An OpenMP program begins as a single thread of execution, called the *initial thread*. Parallelism is achieved through the parallel construct, in which a new *team* of OpenMP threads is spawned. OpenMP allows programmers to define the amount of threads desired for a parallel region by means of the num_threads clause attached to the parallel construct. The spawned threads are joined at the implicit barrier encountered at the end of the parallel region. This is the so-called *fork-join model*. Within the parallel region, parallelism can be distributed in two ways that provide tantamount performance [29]:

1. The *thread-centric model* exploits structured parallelism distributing work by means of work-sharing constructs (e.g., for and sections constructs). It provides a fine-grained control of the mapping between

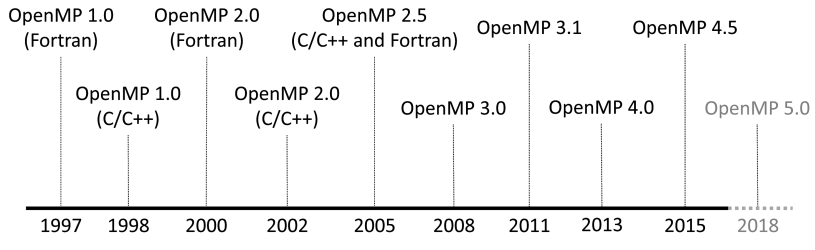


Figure 3.2 OpenMP releases time-line.

work and threads, as well as a coarse grain synchronization mechanism by means of the barrier construct.

2. The *task-centric model*, or simply *tasking model*, exploits both structured and unstructured parallelism distributing work by means of tasking constructs (e.g., task and taskloop constructs). It provides a higher level of abstraction in which threads are mainly controlled by the runtime, as well as fine-grained synchronization mechanisms by means of the taskwait construct and the depend clause that, attached to a task construct, allow the description of a list of *input* and/or *output* dependences. A task with an in, out or inout dependence is ready to execute when all previous tasks with an out or inout dependence on the same storage location complete.

Figure 3.3 shows the execution model of a parallel loop implemented with the for directive, where all spawned threads work in parallel from the beginning of the parallel region as long as there is work to do. Figure 3.4 shows the model of a parallel block with unstructured tasks. In this case, the single construct restricts the execution of the parallel region to only one thread until a task construct is found. Then, another thread (or the same, depending on the scheduling policy), concurrently executes the code of the task. In Figure 3.3, the colours represent the execution of differents iterations of the same parallel loop; in Figure 3.4, colours represent the parallel execution of the code included within a task construct.

3.2.2.2 Acceleration model

OpenMP also provides a *host-centric accelerator model* in which a host offloads data and code to the accelerator devices available in the same

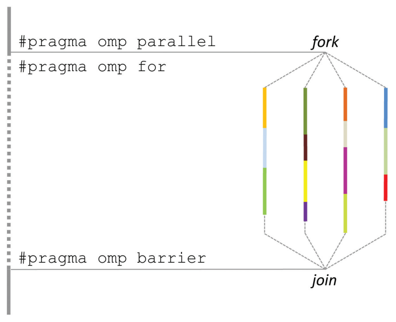


Figure 3.3 Structured parallelism.

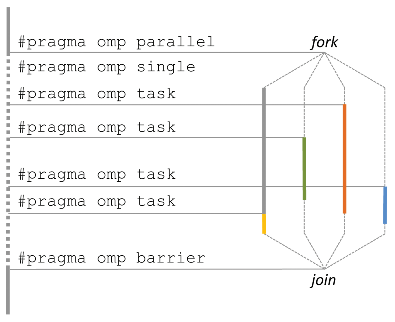


Figure 3.4 Unstructured parallelism.

processor architecture for execution by means of the target construct. When a target directive is encountered, a new *target task* enclosing the target region is generated. The target task is completed after the execution of the target region finishes. One of the most interesting characteristics of the accelerator model is its integration with the tasking model. Note that each accelerator device has its own team of threads that are distinct from threads that execute on another device, and these cannot migrate from one device to another.

In case the accelerator device is not available or even does not exist (this may occur when the code is ported from one architecture to another) the target region is executed in the host. The map clause associated with the target construct specifies the data items that will be mapped to/from the target device. Further parallelism can be exploited within the target device.

3.2.2.3 Memory model

OpenMP is based on a relaxed-consistency, shared-memory model. This means there is a memory space shared for all threads, called *memory*. Additionally, each thread has a temporary view of the memory. The temporary view is not always required to be consistent with the memory. Instead, each private view synchronizes with the main memory by means of the *flush* operation, which can be implicit (due to operations causing a memory fence) or explicit (using the flush operation). Data cannot be directly synchronized between two different threads temporary view.

The view of each thread has of a given variable is defined using data-sharing clauses, which can determine the following sharing scopes:

- private: a new fresh variable is created within the scope.
- firstprivate: a new variable is created in the scope and initialized with the value of the original variable.
- lastprivate: a new variable is created within the scope and the original variable is updated at the end of the execution of the region (only for tasks).
- shared: the original variable is used in the scope, thus opening the possibility of data race conditions.

The use of data-sharing clauses is particularly powerful to avoid unnecessary synchronizations as well as race conditions. All variables appearing within a construct have a default data-sharing defined by the OpenMP specification ([25] Section 2.15.1). These rules are not based on the use of the variables, but on their storage. Thus, users are duty-bound to explicitly scope many variables, changing the default data-sharing values, in order to fulfill correctness (e.g., avoiding data races) and enhance performance (e.g., avoiding unnecessary privatizations).

3.2.3 An OpenMP Example

Listing 3.1 illustrates an OpenMP program that uses both the tasking and the accelerator models. The code enclosed in the parallel construct (line 4) defines a team of four OpenMP threads on the host device. The single construct (line 6) specifies that only one thread starts executing the associated block of code, while the rest of threads in the team remain waiting. When the task regions are created (lines 9 and 11), each one is assigned to one thread in the team (may be the same thread), and the corresponding output dependences on variables x and y are stored. When the target task (lines 13:14) is created, its dependences on x and y are checked. If the tasks producing these variables are finished, then the target task can be scheduled. Otherwise, it must be deferred until the tasks from which it depends have finished. When the target task is scheduled, the code contained in the target region and the variables in the map(to:) clause (x and y) are copied to the accelerator device. After its execution, the res variable is copied back to the host memory as defined by the map(from:) clause. The presence of a nowait clause in the target task allows the execution on the host to continue after the target task is created.

Listing 3.1 OpenMP example of the tasking and the accelerator models combined

```
 1 int foo(int a, int b)
 2 {
 3    int res;
 4    #pragma omp parallel num_threads(4) shared(res) firstprivate(a, b)
 5    {
 6    #pragma omp single shared(res) firstprivate(a, b)
 7    {
 8      int x, y;
 9      #pragma omp task shared(x) firstprivate(a) depend(out:x)
10      x = a*a;
11      #pragma omp task shared(y) firstprivate(b) depend(out:y)
12      y = b*b;
13      #pragma omp target map(to:x,y) map(from:res) nowait \
14                         shared(res) firstprivate(x, y) depend(in:x,y)
15      res = x + y;
16    }
17    return res;
18    }
19 }
```

All OpenMP threads are guaranteed to be synchronized at the implicit barrier included at the end of the parallel and single constructs (lines 16 and 19 respectively). A nowait clause could be added to the single construct to avoid unnecessary synchronizations.

3.3 Timing Properties of the OpenMP Tasking Model

The tasking model of OpenMP not only provides a very convenient abstraction layer upon which programmers can efficiently develop parallel applications, but also has certain similarities with the *sporadic direct acyclic graph (DAG) scheduling model* used to derive a (worst-case) response time analysis of parallel applications. Chapter 4 presents in detail the response time analyses that can be applied to the OpenMP tasking model. This section derives the OpenMP-DAG upon which these analyses are applied.

3.3.1 Sporadic DAG Scheduling Model of Parallel Applications

Real-time embedded systems are often composed of a collection of periodic processing stages applied on different input data streaming coming from sensors. Such a structure makes the system amenable to timing analysis methods [30].

The *task model* [31], either *sporadic or periodic*, is a well-known model in scheduling theory to represent real-time systems. In this model, real-time applications are typically represented as a set of n *recurrent* tasks $\tau = \{\tau_1, \tau_2, .., \tau_n\}$, each characterized by three parameters: worst-case execution time ($WCET$), period (T) and relative deadline (D). Tasks repeatedly emit an infinite sequence of *jobs*. In case of periodic tasks, jobs arrive strictly periodically separated by the fixed interval time T. In case of sporadic tasks, jobs do not have a strict arrival time, but it is assumed that a new job released at time t must finish before $t + D$. Moreover, a minimum interval of time T must occur between two consecutive jobs from the same task.

With the introduction of multi-core processors, new scheduling models have been proposed to better express the parallelism that these architectures offer. This is the case of the *sporadic DAG task model* [32–36], which allows the exploitation of parallelism *within* tasks. In the sporadic DAG task model each task (called *DAG-task*) is represented with a *directed acyclic graph* (DAG) $G = (V, E)$, T and D. Each node $v \in V$ denotes a sequential operation characterized by a WCET estimation. Edges represent dependences between nodes: if $e = (v_1, v_2) : e \in E$, then the node v_1 must complete its execution before node v_2 can start executing. In other words, the DAG captures scheduling constraints imposed by dependences among nodes and it is annotated with the WCET estimation of each individual node.

Overall, the DAG represents the main formalism to capture the properties of a real-time application. In that context, although the current specification of OpenMP lacks any notion of real-time scheduling semantics, such as deadline, period, or WCET, the structure and syntax of an OpenMP program have certain similarities with the DAG model. The task and taskwait constructs, together with the depend clause, are very convenient for describing a DAG. Intuitively, a task describes a node in V in the DAG model, while taskwait constructs and depend clauses describe the *edges* in E. Unfortunately, such a DAG would not convey proper information to derive a real-time schedule that complies with the semantics of the OpenMP specification.

In order to understand where the difficulties of mapping an OpenMP program onto an expressive task graph stem from, and how to overcome them, the next section further delves into the details of the OpenMP execution model.

3.3.2 Understanding the OpenMP Tasking Model

When a task construct is encountered, the execution of the new task region can be assigned to one of the threads in the current team for immediate or deferred execution, with the corresponding impact on the overall timing behaviour. Different clauses allow defining how a task, its parent task and its child tasks will behave at runtime:

- The depend clause allows describing a list of input (in), output (out), or input-output (inout) dependences on data items. Dependences can only be defined among *sibling* tasks, i.e., first-level descendants of the same *parent* task.
- An if clause whose associated expression evaluates to false forces the encountering thread to suspend the current task region. Its execution cannot be resumed until the newly generated task, defined to be an *undeferred task*, is completed.
- A final clause whose associated expression evaluates to true forces all its child tasks to be *undeferred* and *included* tasks, meaning that the encountering thread itself sequentially executes all the new descendants.
- By default, OpenMP tasks are *tied* to the thread that first starts their execution. If such tasks are suspended, they can only be resumed by the same thread. An untied clause forces the task not to be tied to any thread; hence, in case it is suspended, it can later be resumed by any thread in the current team.

Listing 3.2 OpenMP example of task scheduling clauses

```
 1 #pragma omp parallel
 2 {
 3 #pragma omp single nowait                                  // T_0
 4 {
 5     ...        // tp_00
 6     #pragma omp task depend(out:x) untied final(true)      // T_1
 7     {
 8         ...    // tp_10
 9         #pragma omp task                                   // T_4
10         { ... } // tp_4
11         ...    // tp_11
12     }
13     ...        // tp_01
14     #pragma omp task depend(in:x)                          // T_2
15     { ... }    // tp_2
16     ...        // tp_02
17     #pragma omp taskwait
18     ...        // tp_03
19     #pragma omp task                                       // T_3
20     { ... }    // tp_3
21     ...        // tp_04
22 }
23 }
```

Listing 3.2 shows an example of an OpenMP program using different tasking features. The parallel construct creates a new team of threads (since num_threads clause is not provided, the number of threads associated is implementation defined). The single construct (line 3) generates a new task region T_0, and its execution is assigned to just one thread in the team. When the thread executing T_0 encounters its child task constructs (lines 6, 14, and 19), new tasks T_1, T_2, and T_3 are generated. Similarly, the thread executing T_1 creates task T_4 (line 9).

Tasks T_1 and T_2 include a depend clause both defining a dependence on the memory reference x, so T_2 cannot start executing until T_1 finishes. T_4 is defined as an *included task* because its parent T_1 contains a final clause that evaluates to *true*, so T_1 is suspended until the execution of T_4 finishes. All tasks are guaranteed to have completed at the *implicit barrier* at the end of the parallel region (line 23). Moreover, task T_0 will wait on the taskwait (line 17) until tasks T_1 and T_2 have completed before proceeding.

OpenMP defines *task scheduling points* (TSPs) as points in the program where the encountering task can be suspended, and the hosting thread can be rescheduled to a different task. TSPs occur upon task creation and completion and at task synchronization points such as taskwait directives or explicit and implicit barriers[1].

Task scheduling points divide task regions into *task parts* executed uninterruptedly from start to end. Different parts of the same task region are executed in the order in which they are encountered. In the absence of task synchronization constructs, the order in which a thread executes parts of different tasks is unspecified. The example shown in Figure 3.2 identifies the parts in which each task region is divided: T_0 is composed of task parts $tp_{00}, tp_{01}, tp_{02}, tp_{03}$, and tp_{04}; T_1 is composed of task parts tp_{10}, and tp_{11}; and T_2, T_3, and T_4 are composed of task part tp_2, tp_3, and tp_4, respectively.

When a task encounters a TSP, the OpenMP runtime system may either begin the execution of a task region bound to the current team, or resume any previously suspended task region also bound to it. The order in which these actions are applied is not specified by the standard, but it is subject to the following *task scheduling constraints* (TSCs):

TSC 1: An *included* task must be executed immediately after the task is created.

TSC 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 all tasks in the set belong to the same task region and the new *tied* task is a *child task* of the task region.

TSC 3: A dependent task shall not be scheduled until its task data dependences are fulfilled.

TSC 4: When a task is generated by a construct containing an if clause for which the conditional expression evaluates to false, and the previous constraints are already met, the task is executed immediately after generation of the task.

3.3.3 OpenMP and Timing Predictability

The execution model of OpenMP tasks differs from the DAG model in a fundamental aspect: a node in the DAG model is a sequential operation that

[1]Additional TSPs are implied at different OpenMP constructs (target, taskyield, taskgroup). See Section 2.9.5 of the OpenMP specification [25] for a complete list of task scheduling points.

cannot be interrupted[2]. Instead, an OpenMP task can legally contain multiple TSPs at which the task can be suspended or resumed following the TSCs.

Moreover, in order to correctly capture scheduling constraints of each task as defined by the OpenMP specification, a DAG-based real-time scheduling model requires to know: (1) the dependences among tasks, (2) the point in time of each TSP, and (3) the scheduling clauses associated to the task.

This section analyses the extraction of a DAG that represents the parallel execution of an OpenMP application upon which timing analysis can be then applied. It focuses on three key elements:

1. How to reconstruct an OpenMP task graph from the analysis of the code that resembles the DAG-task structure based on TSPs.
2. To which elements of an OpenMP program WCET analysis must be applied.
3. How to schedule OpenMP tasks based on DAG-task methodologies so that TSCs are met.

3.3.3.1 Extracting the DAG of an OpenMP program

The execution of a task part resembles the execution of a node in V, i.e., it is executed uninterrupted. To that end, OpenMP task parts, instead of tasks, can be considered as nodes in V.

Figure 3.5 shows the DAG (named *OpenMP-DAG*) corresponding to the example presented in Listing 3.2, in which task parts form the nodes in V. T_0 is decomposed into task parts tp_{00}, tp_{01}, tp_{02}, tp_{03}, and tp_{04}, with a TSP at the end of each part caused by the task constructs T_1, T_2, and T_3 for tp_{00}, tp_{01}, and tp_{03}, and the taskwait construct for tp_{02}. Similarly, T_1 is decomposed into tp_{10} and tp_{11} with the TSP corresponding to the creation of task T_4 at the end of tp_{10}.

Depending on the origin of the TSP encountered at the end of each task part (i.e., task creation or completion, or task synchronization) three different types of dependences are identified: (a) control-flow dependences (dotted arrows), which force parts to be scheduled in the same order as they are executed within the task; (b) TSP dependences (dashed arrows), which force tasks to start/resume execution after the corresponding TSP, and (c) full synchronizations (solid arrows), which force the sequential execution of tasks as defined by the depend clause and task synchronization constructs. Note that all dependence types have the same purpose, which is to express

[2]This assumes the execution of a single DAG program, where a node cannot be interrupted to execute other nodes of the same graph. In a multi-DAG execution model, nodes can be preempted by nodes from different DAG programs if allowed by the scheduling approach.

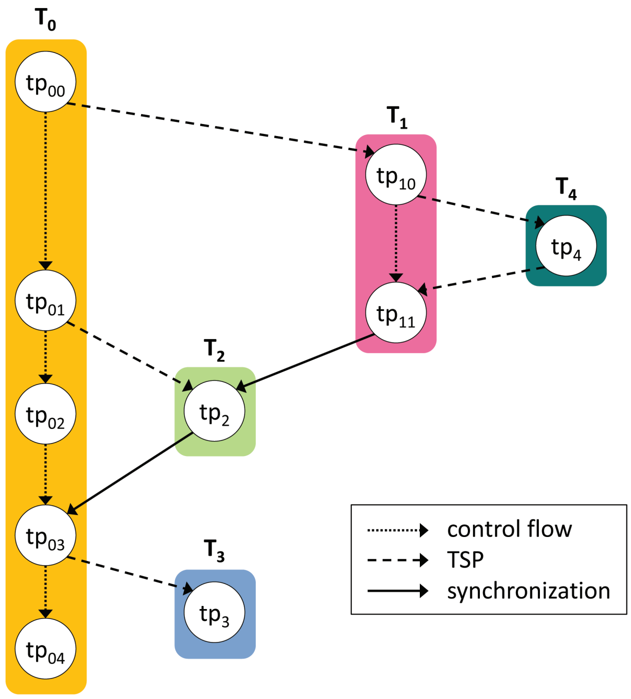


Figure 3.5 OpenMP-DAG composed of *task parts* based on the code in Listing 3.2.

a scheduling precedence constraint. As a result, the OpenMP-DAG does not require to differentiate them.

Besides the depend clause, the if and final clauses also affect the order in which task parts are executed. In both cases the encountering task is suspended until the newly generated task completes execution. In order to model the undeferred and included tasks behaviour, a new edge is introduced in E. In Figure 3.5, a new dependence between tp_{40} and tp_{11} is inserted, so the task region T_1 does not resume its execution until the included task T_4 finishes.

3.3.3.2 WCET analysis is applied to *tasks* and *task parts*

In order to comply with the DAG-model, nodes in the OpenMP-DAG must be further annotated with the WCET estimation of the corresponding task parts. By constructing the OpenMP-DAG based on the knowledge of TSPs (i.e., by considering as nodes in V only those code portions that are executed uninterruptedly from start to end) the timing analysis of each node has a WCET which is independent of any dynamic instance of the OpenMP program (i.e., how threads may be scheduled to tasks and parts therein). As a result, the timing behaviour of task parts will only be affected by concurrent accesses to shared resources [37]. It is important to remark that the WCET estimation is applied to a task when it is composed of a single task part. This is the case of T_2, T_3, and T_4 from Figure 3.5.

3.3.3.3 DAG-based scheduling must not violate the TSCs

When real-time scheduling techniques are applied to guarantee the timing behaviour of OpenMP applications, the semantics specified by the OpenMP TSCs must not be violated.

The clauses associated to a task construct not only define precedence constraints, as shown in Section 3.3.3.1, but they also define the way in which tasks, and task parts therein, are scheduled according to the TSCs defined in Section 3.3.2. This is the case of the if, final and untied clauses, as well as the default behaviour of *tied* tasks. These clauses influence the order in which tasks execute and also how task parts are scheduled to threads. Regarding the latter, the restrictions imposed by TSCs are the following:

- *TSC 1* imposes *included* tasks to be executed immediately by the encountering thread. In this case, the scheduling of the OpenMP-DAG has to consider both the task part that encounters it and the complete *included* task region as a unique unit of scheduling. In Figure 3.5, the former case would give tp_4 the highest priority, and the latter case would consider tp_{10} and tp_4 as a unique unit of scheduling.
- *TSC 2* does not allow scheduling new *tied* tasks if there are other suspended *tied* tasks already assigned to the same thread, and the suspended tasks are not descendants of the new task. Listing 3.3 shows a fragment of code in which this situation can occur. Let's assume that T_1, which is not a descendent of T_3, is executed by *thread 1*. When T_1 encounters

Listing 3.3 Example of an OpenMP fragment of code with *tied tasks*

```
...
#pragma omp task // T1
{
    #pragma omp task if(false) // T2
    {...}
}
#pragma omp task // T3
{...}
```

the TSP of the creation of T_2, it is suspended because of TSC 4, and it cannot resume until T_2 finishes. Let's consider that T_2 is being executed by a different thread, e.g., *thread 2*. If T_2 has not finished when the TSP of the creation of T_3 is reached, then T_3 cannot be scheduled on *thread 1* because of TSC 2, even if *thread 1* is idle. As a result, *tied* tasks constrain the scheduling opportunities of the OpenMP-DAG.

- *TSC 3* imposes tasks to be scheduled respecting their dependences. This information is already contained in the OpenMP-DAG.
- *TSC 4* states that *undeferred* tasks execute immediately if TSCs 1, 2, and 3 are met. Differently, *untied* tasks are not subject to any TSC, 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. Therefore, one possible scheduling strategy for *untied* tasks that satisfies TSC 4 is not to schedule *undeferred* and *untied* task parts until *tied* and *included* tasks are assigned to a given thread. This guarantees that TSCs 1 and 2 are met. This is because task parts of *tied* and *included* tasks are bound to the thread that first started their execution, which reduces significantly their scheduling opportunities. Instead, *untied* and *undeferred* task parts have a higher degree of freedom as they can be scheduled to any thread of the team. Therefore, for the OpenMP-DAG to convey enough information to devise a TSC-compliant scheduling, each node in V must be augmented with the *type of task* as well (untied, *tied*, *undeferred* and *included*) as shown in Figure 3.5.

Figure 3.6 shows a possible schedule of task parts in Listing 3.2, assuming a work-conserving scheduling. T_0 is a tied task, so all its task parts are scheduled to the same thread (*thread 1*). T_1 is an *untied* task so tp_{10} and tp_{10} can execute in different threads (*thread 1* and *2* in the example). Note that tp_{11} does not start executing until tp_4 completes due to the TSP constraint. Moreover, the execution of tp_4 starts immediately after the creation of T_4 on

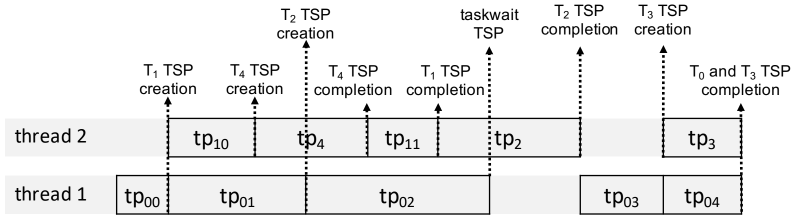


Figure 3.6 DAG composed of task parts.

the same thread that encounters it (*thread 2*). Finally, tp_2 and tp_3 are scheduled to idle threads (*thread 4* and *5*, respectively) once all their dependences are fulfilled.

3.4 Extracting the Timing Information of an OpenMP Program

The extraction of an OpenMP-DAG representing the parallel execution of an OpenMP program in such a way that timing analysis can be performed, requires analyzing the OpenMP constructs included in the source code, so the nodes and edges that form the DAG can be identified. This information can be obtained by means of compiler analysis techniques. Concretely, there exists two different analysis stages needed to build the OpenMP-DAG $G = (V, E)$:

1. A *parallel structure stage*, in which the nodes in V, i.e., tasks parts, and edges in E, are identified based on TSPs, TSCs, and data- and control-flow information.

Listing 3.4 OpenMP program using the tasking model

```
#pragma omp parallel num_threads(8)
{
#pragma omp single nowait                              //T0
{
   for(i=0; i<=2; i++)
      for(int j=0; j<=2; j++) {
         if(i==0 && j==0) {            // Initial block
            #pragma omp task depend(inout:m[i][j])
            compute_block(i, j);                     // T1
         } else if (i == 0) {          // Blocks in upper edge
            #pragma omp task depend(in:m[i][j-1], inout:m[i][j])
            compute_block(i, j);                     // T2
         } else if (j == 0) {          // Blocks in left edge
            #pragma omp task depend(in:m[i-1][j], inout:m[i][j])
            compute_block(i, j);                     // T3
         } else {                      // Internal blocks
            #pragma omp task depend(in:m[i-1][j],in:m[i][j-1], \\
                                    in:m[i-1][j-1],inout:m[i][j])
            compute_block(i, j);                     // T4
         }
      }
}
}
}
```

2. A *task expansion stage*, in which the tasks (and task parts) that will be actually instantiated at runtime are identified by expanding the control flow information extracted in the previous stage.

The next subsections further describe these stages. With the objective of facilitating the explanation of the compiler analysis techniques, Listing 3.4 introduces an OpenMP program that will be used for illustration purposes. The code processes the elements of a blocked 2D matrix using a *wave-front* parallelization strategy [38]. The parallel construct (line 1) defines a team of 8 threads. The single construct (line 3) specifies that only one thread will execute the associated code. The algorithm divides the matrix in 3×3 blocks, assigning each one to a different task. Each block $[i, j]$ consumes the previous adjacent blocks and itself. Hence, all tasks (lines 8, 11, 14, and 17:18) have an inout dependence on the computed block $[i, j]$. T_2 and T_3 (lines 11 and 14) compute the upper and left edges, so additionally they consume the left $[i, j-1]$ and upper $[i-1, j]$ blocks, respectively. Finally, T_4 (lines 17:18) computes the internal blocks, hence additionally it consumes the left $[i-1, j]$, upper $[i, j-1]$, and left-upper diagonal $[i-1, j-1]$ blocks. All tasks are guaranteed to complete at the implicit barrier at the end of the parallel region (line 24).

3.4.1 Parallel Structure Stage

This stage identifies the TSPs surrounding tasks parts, and the corresponding TSCs associated with each task part in order to: (1) generate a parallel control-flow graph (PCFG) that holds all this information as well as parallel semantics [39], and (2) analyze this graph so that the necessary information to expand a complete DAG is obtained. With such purpose in mind the analysis performs the following calculations:

- Generate the PCFG of the source code taking into account: (a) the dependences introduced by any kind of TSPs (i.e., task creation, task completion and task synchronization), as introduced in Section 3.3.3.1, (b) the data dependences introduced by the depend clause, and (c) the if and final clauses, hence the behaviour of undeferred and included tasks.
- On top of that, analyze the control-flow statements, i.e., selection statements (if-else and switch) and loops that identify whether a task is instantiated or not at runtime. To do so, three analyses are required: induction variables [40], reaching definitions [41], and range analysis [42]. Additionally, determine the conditions that must be fulfilled for two instantiated tasks to depend on one another [3].

3.4.1.1 Parallel control flow analysis

The *abstract syntax tree* (AST) used in the compiler to represent the source code is used to generate the PCFG of an OpenMP program. This enriches the classic control-flow graph (CFG) with information about parallel execution. This process performs a conservative analysis of the synchronizations among tasks, because the compiler may not be able to assert when two depend clauses designate the same memory location, e.g., array accesses or pointers. Hence, synchronization edges are augmented with predicates defining the condition to be fulfilled for an edge to exist. In the example shown in Listing 3.4, the dependences that matrix m originates among tasks depend on the values of i and j.

3.4.1.2 Induction variables analysis

On top of the PCFG, the compiler evaluates the loop statements to discover the induction variables (IVs) and their evolution over the iterations using the common tuple representation $\langle lb, ub, str\rangle$, where lb is the lower bound, ub is the upper bound, and str is the stride. This analysis is essential for the later expansion of the graph, since the induction variables will determine the shape of the iteration space for each loop statement.

3.4.1.3 Reaching definitions and range analysis

Finally, the compiler computes the values of all variables involved in the execution of any task. With such a purpose, it analyzes reaching definitions and also extends range analysis with support for OpenMP. The former computes the definitions reaching any point in the program. The later computes the values of the variables at any point of the program in four steps: (1) generate a set $\mathcal{C}$ of equations that constrain the values of each variable (equations are built for each assignment and control flow statement); (2) build a *constraint graph* that represents the relations among the constraints; (3) split the graph into *strongly connected components* (SCCs) to avoid cycles; (4) propagate the ranges over the SCCs in topological order. Both analyses are needed to propagate the values of the relevant variables across the expanded code.

3.4.1.4 Putting all together: The wave-front example

The previously mentioned analyses provide the information needed to generate an initial version of the DAG, named *augmented DAG* (*aDAG*), with data and control flow knowledge. The aDAG is defined by the tuple

$$aDAG = \langle N, E, C\rangle \tag{3.1}$$

where:

- $N = \{V \times T_N\}$ is the set of nodes with their corresponding type $T_N = \{Task, Taskwait, Barrier\}$.
- $E = \{N \times N \times P\}$ is the set of possible synchronization edges with the predicate P that must fulfill for the edge to exist.
- $C = N \times \{F\}$ is the set of *control flow statements* involved in the instantiation of any task $n \in N$, where $F = S \times \{T_F\}$, being S the condition to instantiate the tasks and $T_F = \{Loop, IfElse, Switch\}$, the type of the structure.

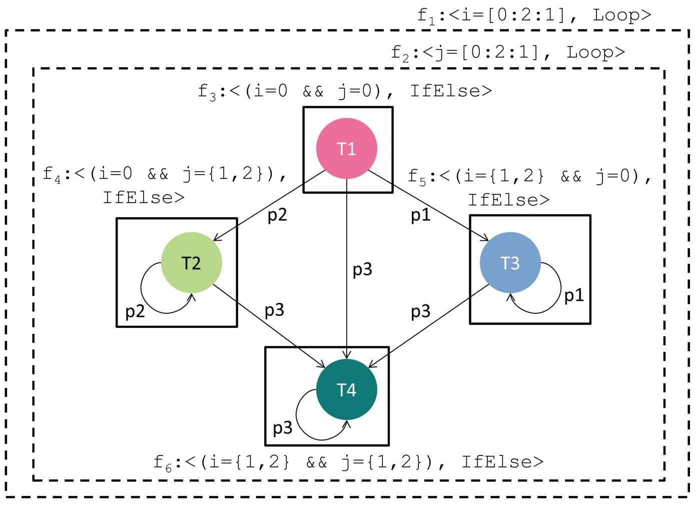
Figure 3.7 shows the aDAG of the OpenMP program in Listing 3.4. The set of nodes N includes all task constructs $N = T_1, T_2, T_3, T_4$ (lines 8, 11, 14, and 17:18), all with type $T_N = Task$. The control flow statements for each node N, $f_i \in F$ are the for (lines 5 and 6) and if (lines 7, 10, 13, and 16) statements, and include information about: (a) the IVs of each loop i, j, both with $lb = 0$, $ub = 2$ and $str = 1$ (dashed-line boxes); (b) the conditions of the selection statements enclosing each task (solid-line boxes), and (c) the ranges of the variables in those conditions. In the figure, T_3 is instantiated if $i = 1$ or 2 and $j = 0$. In the predicates $p \in P$ associated to the synchronization edges in E, the left hand side of the equality corresponds to the value of the variable at the point in time the source task is instantiated, while the right side corresponds to the value when the target task is instantiated. For example, the predicate of the edge between T_1 and T_3 with $p1((i_S == i_T || i_S == i_T - 1) \&\& j_S == j_T)$ evaluates to *true*, meaning that the edge exists when the values of i and j in the source task T_1 are $i_S = 0$ and $j_S = 0$, and the values of i and j in the target task T_3 are $i_T = 1$ and $j_T = 0$.

For simplicity, Figure 3.7 only includes the dependences that are actually expanded in the next stage (Section 3.4.2). The actual aDAG has edges between any possible pair of tasks because they all have inout dependences on the element $m[i][j]$. Moreover, the task-parts that form the task T_0 with the corresponding task creation dependences are not included.

3.4.2 Task Expansion Stage

3.4.2.1 Control flow expansion and synchronization predicate resolution

Based on the aDAG, this stage generates an *expanded DAG* (or simply DAG) representing the complete execution of the program in two phases: (1) expand control flow structures (i.e., decide which branches are taken for the selection


f1:<i=[0:2:1], Loop>
f2:<j=[0:2:1], Loop>
f3:<(i=0 && j=0), IfElse>
f4:<(i=0 && j={1,2}), IfElse>
f5:<(i={1,2} && j=0), IfElse>
f6:<(i={1,2} && j={1,2}), IfElse>
T1
T2
T3
T4
p1
p2
p3

p1: $(i_S == i_T \ || \ i_S == i_T - 1) \ \&\& \ j_S == j_T$

p2: $i_S == i_T \ \&\& \ (j_S == j_T \ || \ j_S == j_T - 1)$

p3: $(i_S == i_T \ || \ i_S == i_T - 1) \ \&\& \ (j_S == j_T \ || \ j_S == j_T - 1)$

Figure 3.7 aDAG of the OpenMP program in Listing 3.4.

statements and how many iterations are executed for the loop statements) to determine which tasks (and so task-parts) are actually instantiated; and (2) resolve the synchronization predicates to conclude which tasks have actual dependences.

Control flow structures are expanded from outer to inner levels. In the aDAG in Figure 3.7, the outer loop f_1 is expanded first, and then the inner loop f_2. Finally, the if-else structures f_3, f_4, f_5, and f_6 are resolved. Each expansion requires the evaluation of the associated expressions to determine the values of each variable. For example, when the outer loop f_1 is expanded, each iteration is associated with the corresponding value of i.

This expansion process creates two identifiers: (1) an identifier of the loops involved in the creation of a task (l_i), labeling each loop expansion step, and (2) a unique *static task construct identifier* (sid_t), labeling each task construct.

The process results in a temporary DAG in which all tasks instantiated at runtime are defined, but synchronization predicates are not solved. To do so, the value of the variables propagated in the control flow expansion is used to evaluate predicates and decide which edges actually exist.

Likewise, loop identifiers l_i are used to eliminate backwards dependences, i.e., tasks instantiated in previous iterations cannot depend on tasks instantiated in later iterations.

Figure 3.8 shows the final DAG of the program in Listing 3.4. It contains all task instances with a unique numerial identifier (explained in the next section) and all dependences that can potentially exist at runtime. Transitive dependences (dashed arrows) are included as well, although they can be removed because they are redundant.

3.4.2.2 t_{id}: A unique task instance identifier

A key property of the expanded task instances is that they must include a *unique task instance identifier* t_{id} required to match the instantiated tasks expanded at compile-time (and included in the DAG) with those instantiated at runtime. Equation 3.2 computes t_{id} as follows:

$$t_{id} = sid_t + T \times \sum_{i=1}^{L_t} l_i \cdot M^i \tag{3.2}$$

where sid_t is a unique task construct identifier (computed during the control flow expansion stage), T is equal to the number of task, $\mathrm{taskwait}$, and $\mathrm{barrier}$ constructs in the source code, L_t is the total number of nested loops involved in the execution of the task t, i refers to the the nesting level, l_i is the loop unique identifier at nesting level i (computed during the control

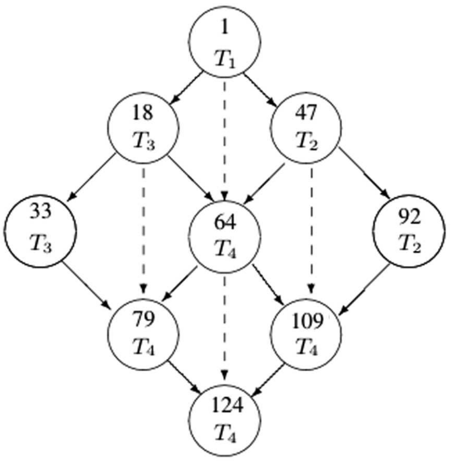


Figure 3.8 The DAG of the OpenMP program in Listing 3.4.

flow expansion stage), and M the maximum number of iterations of any considered loop.

The use of loop properties in Equation 3.2 (i.e., L_t, l_i, i, and M), guarantees that a unique task identifier for each task instance is generated, even if they come from the same task construct. Hence, task instances from different loop iterations result in different t_{id} because every nesting level l_i is multiplied by the maximum number of iterations M.

Consider task T_4, with identifier 79, in Figure 3.8. This task instance corresponds to the computation of the matrix block $m[2,1]$. Its identifier is computed as follows: (1) $sid_{T4} = 4$, because T_4 is the fourth task found in sequential order while traversing the source code; (2) $T = 5$ because there are four task constructs and one (implicit) barrier in the source code; (3) $L_{T_4} = 2$, the two nested loops enclosing T_4; (4) $M = 3$, the maximum number of iterations in any of the two considered loops; and (5) $l_1 = 2$ and $l_2 = 1$ are the values of the loop identifiers at the corresponding iteration. Putting them all together: $T4_{id} = 4 + 5(2 * 3^1 + 1 * 3^2) = 79$.

It is important to remark that t_{id} must be computed at both compile-time and run-time, and so all information needed to compute Equation 3.2 must be available in both places. Chapter 6 presents the combined compiler and run-time mechanisms needed to reproduce all the required information (including sid_t and l_i identifiers) at run-time.

3.4.2.3 Missing information when deriving the DAG

In case the framework cannot derive some information (mostly when control-flow statements and dependences contain pointers that may alias or arrays with unresolved subscripts, or the values are not known at compile-time), it still generates a DAG that correctly represents the execution of the program. Next, each possible case is argued:

- When an if-else statement cannot be evaluated, all its related tasks in C are considered for instantiation, hence included in the DAG. In this case, the DAG will include a task instance that will never exist. Chapters 4 and 6 present the mechanisms required to take this into consideration for response time analysis and parallel run-time execution.
- If a loop cannot be expanded because its boundaries are unknown, parallelism across iterations is disabled by inserting a taskwait at the end of the loop. By doing so, all tasks instantiated within an iteration must complete before the next iteration starts.

- Lastly, dependences whose predicate cannot be evaluated are always kept in the DAG, making the involved tasks serialized.

The situations described above will result in a bigger DAG (when if–else conditions cannot be evaluated) or in a performance loss (when loop bounds or synchronization predicates cannot be determined), although a correct DAG is guaranteed. In the worst-case scenario, where no information can be derived at compile-time, the resultant DAG corresponds to the sequential execution of the program, i.e., all tasks are assumed to be instantiated, and their execution is to be sequentialized. It is important to remark that embedded applications can often provide all the required information to complete the DAG expansion, as it is required for timing analysis [43].

3.4.3 Compiler Complexity

The complexity of the compiler is determined by the complexity of the two stages presented in Sections 3.4.1 and 3.4.2.

The complexity of the control/data flow analysis stage is dominated by the PCFG analysis and range analysis phases. The complexity of the former is related to the number of split constructs present in the source code, in which the Cyclomatic Complexity [44] metric is usually used. The latter, has been proved to have an asymptotic linear complexity [42].

The complexity of the task expansion stage is dominated by the computation of the dependences among tasks, which is performed using a Cartesian product: the input dependence of a task can be generated by any of the previously created task instances. As a result, the complexity is quadratic on the number of instantiated tasks.

3.5 Summary

This chapter provided the rationale and the model for the use of fine-grained parallelism in general, and the OpenMP parallel programming model in particular, to support applications that require predictable performance, to develop future critical real-time embedded systems, and analyze the time predictable properties of the OpenMP tasking model. Based on this model, the chapter then described the advances in compiler techniques to extract timing information of OpenMP parallel programs, and build the *OpenMP* DAG required to enable predictable scheduling (described in the next chapter) and the needed timing analysis (in Chapter 5). This OpenMP-DAG also provides the building block for the execution of the OpenMP runtime (Chapter 6) and Operating System (Chapter 7).

References

[1] Pllana, S., and Xhafa, F., *Programming Multicore and Many-core Computing Systems*, volume 86. John Wiley and Sons, 2017.

[2] Reinders, J., *Intel Threading Building Blocks*. O'Reilly and Associates, Inc., 2007.

[3] *NVIDIA CUDA C Programming Guide*. https://docs.nvidia.com/cuda /cuda-c-programming-guide/index.html, 2016.

[4] Stone, J. E., Gohara, D., and Shi, G., OpenCL: A parallel programming standard for heterogeneous computing systems. *CSE*, 12, 66–73, 2010.

[5] Snir, M., *MPI–the Complete Reference: The MPI core*, volume 1. MIT press, 1998.

[6] Butenhof, D. R., *Programming with POSIX Threads*. Addison-Wesley, 1997.

[7] Chapman, B., Jost, G., and Van Der Pas., *Using OpenMP: Portable Shared Memory Parallel Programming*, volume 10. MIT press, 2008.

[8] Duran, A., Ayguadé, E., Badia, R. M., Labarta, J., Martinell, L., Martorell, X., and Planas, J., Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Process. Lett.* 21, 173–193, 2011.

[9] Varbanescu, A. L., Hijma, P., Van Nieuwpoort, R., and Bal, H., "Towards an effective unified programming model for many-cores." In *IPDPS*, pp. 681–692. IEEE, 2011.

[10] OpenACC. *Directives for Accelerators*. http://www.openacc-standard.org, 2017.

[11] Robison, A. D., Cilk plus: Language support for thread and vector parallelism. *Talk at HP-CAST*, 18:25, 2012.

[12] Leiserson, C. E., "The cilk++ concurrency platform." In *Design Automation Conference, 2009. DAC'09. 46th ACM/IEEE*, pp. 522–527. IEEE, 2009.

[13] Saule, E., and Çatalyürek, Ü. V., "An early evaluation of the scalability of graph algorithms on the intel mic architecture." In *Parallel and Distributed Processing Symposium Workshops and PhD Forum (IPDPSW), 2012 IEEE 26th International*, pp. 1629–1639. IEEE, 2012.

[14] De Dinechin, B. D., Van Amstel, D., Poulhiés, M., and Lager, G., "Time-critical computing on a single-chip massively parallel processor." In *DATE*, 2014.

[15] CEA STMicroelectronics. Platform 2012: A many-core programmable accelerator for ultra-efficient embedded computing in nanometer technology. *Whitepaper,* 2010.

[16] Texas Instruments. *SPRS866: 66AK2H12/06 Multicore DSP+ARM KeyStone II System-on-Chip (SoC).*

[17] Kegel, P., Schellmann, M., and Gorlatch, S., "*Using OpenMP vs. Threading Building Blocks for Medical Imaging on Multi-Cores*." In *Europar*. Springer, 2009.

[18] Lee, S., Min, S-J., and Eigenmann, R., OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization. *SIGPLAN Not.* 44, 101–110, 2009.

[19] Shen, J., Fang, J., Sips, H., and Varbanescu, A. L., "Performance gaps between OpenMP and OpenCL for multi-core CPUs." In *ICPPW*, pp. 116–125. IEEE, 2012.

[20] Krawezik, G., and Cappello, F., "Performance comparison of MPI and three OpenMP programming styles on shared memory multiprocessors." In *SPAA*. ACM, 2003.

[21] Kuhn, B., Petersen, P., and O'Toole, E., OpenMP versus threading in C/C++. *Concurr. Pract. Exp.* 12, 1165–1176, 2000.

[22] *OpenMP 2.5 Application Programming Interface*. http://www.openmp.org/wp-content/uploads/spec25.pdf, 2005.

[23] *OpenMP 3.0 Application Programming Interface*. http://www.openmp.org/wp-content/uploads/spec30.pdf, 2008.

[24] *OpenMP 4.0 Application Programming Interface*. http://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf, 2013.

[25] *OpenMP 4.5 Application Programming Interface.* http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf, 2015.

[26] *OpenMP Technical Report 2 on the OMPT Interface.* http://www.openmp.org/wp-content/uploads/ompt-tr2.pdf, 2014.

[27] *OpenMP Technical Report 4: Version 5.0 Preview 1.* http://www.openmp.org/wp-content/uploads/openmp-tr4.pdf, 2016.

[28] *OpenMP Technical Report 5: Memory Management Support for OpenMP 5.0.* http://www.openmp.org/wp-content/uploads/openmp-TR5-final.pdf, 2017.

[29] Podobas, A., and Karlsson, S., "Towards Unifying OpenMP Under the Task-Parallel Paradigm." In *IWOMP*, 2016.

[30] Buttazzo, G. C., *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, volume 24. Springer Science and Business Media, 2011.

[31] Davis, R. I., and Burns, A., A survey of hard real-time scheduling for multiprocessor systems. *ACM computing surveys (CSUR)*, 43, 35, 2011.

[32] Bonifaci, V., Marchetti-Spaccamela, A., Stiller, S., and Wiese, A., "Feasibility analysis in the sporadic dag task model." In *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, pp. 225–233. IEEE, 2013.

[33] Baruah, S., Bonifaci, V., Marchetti-Spaccamela, A., Stougie, L., and Wiese, A., "A generalized parallel task model for recurrent real-time processes." In *Real-Time Systems Symposium (RTSS), 2012 IEEE 33rd*, pp. 63–72. IEEE, 2012.

[34] Saifullah, A., Li, J., Agrawal, K., Lu, C., and Gill, C., Multi-core real-time scheduling for generalized parallel task models. *Real-Time Sys.* 49, 404–435, 2013.

[35] Baruah, S., "Improved multiprocessor global schedulability analysis of sporadic dag task systems." In *Real-Time Systems (ECRTS), 2014 26th Euromicro Conference on*, pp. 97–105. IEEE, 2014.

[36] Li, J., Agrawal, K., Lu, C., and Gill, C., "Outstanding paper award: Analysis of global edf for parallel tasks." In *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, pp. 3–13. IEEE, 2013.

[37] Radojković, P., Girbal, S., Grasset, A., Quiones, E., Yehia, S., and Cazorla, F. J., On the evaluation of the impact of shared resources in multithreaded cots processors in time-critical environments. *ACM Trans. Architec. Code Opt. (TACO)*, 8:34, 2012.

[38] Rochange, C., Bonenfant, A., Sainrat, P., Gerdes, M., Wolf, J., Ungerer, T., et al. "Wcet analysis of a parallel 3d multigrid solver executed on the merasa multi-core." In *OASIcs-OpenAccess Series in Informatics*, volume 15. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010.

[39] Royuela, S., Ferrer, R., Caballero, D., and Martorell, X., "Compiler analysis for openmp tasks correctness." In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, p. 7. ACM, 2015.

[40] Muchnick, S. S., *Advanced Compiler Design Implementation.* Morgan Kaufmann, 1997.

[41] Aho, A. V., Sethi, R., and Ullman, J. D., *Compilers: Principles, Techniques, and Tools*, volume 2. Addison-wesley Reading, 2007.

[42] Pereira, F. M. Q., Rodrigues, R. E., and Campos, V. H. S., "A fast and low-overhead technique to secure programs against integer overflows."

In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp 1–11. IEEE Computer Society, 2013.

[43] Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., et al. The worst-case execution-time problem?"overview of methods and survey of tools. *ACM Trans. Embed. Comput. Sys. (TECS)*, 7:36, 2008.

[44] McCabe, T. J., "A complexity measure." *IEEE Transactions on software Engineering*, 4, pp. 308–320, 1976.