6

OpenMP Runtime

Andrea Marongiu[1,2], Giuseppe Tagliavini[2] and Eduardo Quiñones[3]

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

This chapter introduces the design of the OpenMP runtime and its key components, the *offloading library* and the *tasking runtime library*. Starting from the execution model introduced in the previous chapters, we first abstractly describe the main interactions among the main actors involved in program execution. Then we focus on the optimized design of the offloading library and the tasking runtime library, followed by their performance characterization.

6.1 Introduction

The model assumed in the previous chapters considers the existence of multiple applications, starting execution on the *host* processor, and each one is composed of multiple real-time (RT) tasks which can be sent to the *accelerator* with the aim to speed up their execution. This paradigm, commonly referred to as *offloading*, has been widely adopted in many computing domains from embedded systems to HPC [1, 2]. In the context of the Kalray architecture (described in Chapter 2), IO cores take on the *host* role while the *clusters* are used as accelerators. Accordingly, an OpenMP-based software stack with offloading support must leverage both host and acceleration roles. On the *host* side, an OpenMP directive ($\#\text{pragma omp target}$) is used to specify a region of code which can be offloaded. Inside a cluster, a pool of threads is dedicated for the execution of the offloaded workload and the RTOS (introduced in Chapter 7) is in charge of scheduling the execution of the threads on the available cores.

The complete software stack to handle the described execution model is composed of an *offloading library* and a *tasking runtime library*. The offloading library executes on the *host* and is in charge of initiating offload sequences to the accelerator. On the accelerator side, the *request manager* (RM) is the component in charge to collect offload requests and create pools of threads (hereafter called *jobs* as in the OS terminology) to execute them. Depending on runtime design and hardware specific features, the RM can be implemented as an RT task (a software component) or may be mapped to a dedicated core (a hardware component). The tasking runtime library provides an optimized support for task parallelism on the accelerator and runs on top of the RTOS. It is further divided into a low-level library (or LL-RTE) [3], where all the tightly coupled interactions with the RTOS are implemented, plus a high-level library, where all the management of the tasking constructs resides.

6.2 Offloading Library Design

Figure 6.1 summarizes the timing diagram (time flows from top to bottom on the vertical axis) and the interactions between the software blocks providing the offload support. At the higher level of abstraction, the *host* sends request to the RM, which orchestrates the execution of the workload on the *processing elements* (PEs).

The *host* support is implemented as a user-level library that interfaces OpenMP offloads (expressed at the application level with a target directive) to the computing clusters. The key features of this library can be summarized as follows:

- **Low-cost offload:** As initializing the communication channels between the *host* and the offload manager and loading into the cluster shared memory the binary file containing the OpenMP library (high-level library + LL-RTE) are costly operations, the *host* offload library implements it as a one-time operation that happens at system startup (the $\mathrm{GOMP\_init}$ method). Every time the *host* program encounters a target directive, this is translated into a call to the $\mathrm{GOMP\_target}$ function, which sends a control packet to the offload manager of the target cluster and then triggers the copy of input data. This handshake procedure is streamlined to guarantee minimum overhead.
- **Asynchronous offload:** The offload procedure is asynchronous. After sending the offload request to the cluster, $\mathrm{GOMP\_target}$ immediately

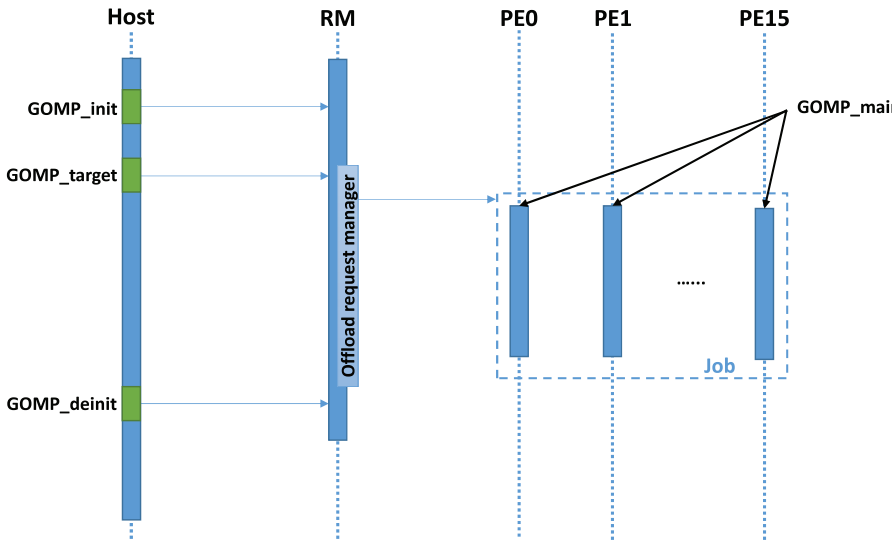


Figure 6.1 Timing diagram of an offloading procedure.

returns to the caller (with the exception of the multi-offload case described below). The result of the offload computation can be retrieved by calling a blocking synchronization primitive (GOMP_target_wait).

- **Multi-cluster support:** An application can perform offloads on different clusters, from 1 up to 16. The initialization is required for each cluster that is used by the current application. The cluster is specified by the programmer using the OpenMP syntax (i.e., the device clause of a target directive).
- **Multi-offload support:** An application can perform multiple offloads on the same cluster. At the same time, multiple offloads can coexist on the same cluster at different priority levels[1]. The priority level is specified by the programmer using the OpenMP syntax (i.e., the priority clause), and it is propagated to the runtime using a parameter of GOMP_target.

The function calls to the offloading runtime are not invoked directly by the developer, as the OpenMP syntax is used to identify the code and data to be offloaded. The compiler transforms the offloading OpenMP directives as

[1]Focusing on the MPPA-256 platform, currently two levels are supported on RTEMS hosts and four on Linux hosts

defined in its accelerator model (i.e., target and declare target directives) to the corresponding offloading runtime calls, as described in Chapter 3.

In our design, the RM is implemented as a persistent RTOS task to be executed on the accelerator side (i.e., by one of the cluster cores). The RTOS leverages the notion of a *task scheduling point* (TSP) to check the availability of a new offload request and perform the requested actions. At each such scheduling point, the RTOS can (re)start the execution of the RM itself (if a new offload has arrived in the meantime and needs to be enqueued to the ready job list) or another job in the queue, depending on the synchronization policy adopted. TSPs are naturally identified as synchronization points in an OpenMP program (see Chapter 3 for more details). The OS provides synchronization primitives (described in Chapter 7) which can be used to block one (or more) thread(s) within a job on a certain wait condition, and the OpenMP runtime invokes these primitives to enforce synchronization.

To reduce the runtime overheads, the metadata for all the supported RTOS jobs on a cluster (one per priority level) are created and initialized upon the first call to the RM. The activated jobs execute the GOMP_main function of the runtime library to initialize the offload support on the cluster side.

6.3 Tasking Runtime

The OpenMP tasking model has been introduced in Chapter 3. Task-based parallelism offers a powerful conceptual framework to exploit irregular parallelism in target applications, and several works have demonstrated the effectiveness of tasking [4–7]. However, the sophisticated semantics of the OpenMP tasking execution model are translated into a complex control code that has to be executed in addition to the application code itself. This ultimately results in significant time overheads, if the application tasks are not large enough to hide such overheads. Thus, a performance-efficient design of a tasking runtime environment (RTE) targeting low-end embedded manycore accelerators is a challenging task, as embedded parallel applications typically exhibit very fine-grained parallelism [6, 8], and are thus very sensitive to time overheads. Moreover, memory overheads are also very relevant in this context, as embedded architectures feature very limited amounts of fast, on-chip memory. Allocating runtime support metadata in such memories reduces time overheads, as the control code executes faster, but reduces the space available for program data. As the metadata for a tasking runtime might consume a significant amount of memory, it is necessary to find a good tradeoff between the implied space and time overheads.

The applicability of the tasking approach to embedded applications and embedded manycore accelerators is often limited to coarse-grained parallel tasks, capable of tolerating the high overheads typically implied in a tasking runtime. State-of-the-art tasking runtimes for embedded manycores [6] succeed in achieving low overheads and enabling high speedups for very fine-grained tasks, but only for simple *flat* parallel patterns (i.e., where all the tasks are created from the same parent task). The main reason for this limitation lies in a key design choice: only *tied* tasks are supported by most RTEs, whereas *untied* tasks are not supported. If a *tied* task is suspended (due to synchronization, creation of another task, etc.), only the thread that initially owned it is allowed to resume its execution. This clearly significantly limits the available parallelism when more sophisticated (and realistic) parallel execution patterns are considered, like nested tasking (for instance, in programs that use recursion).

Scheduling policies: Another limitation that follows from supporting only *tied* tasks is the restricted set of scheduling policies available. Breadth-first scheduling (BFS) and work-first scheduling (WFS) are the two most widely used policies for distributing tasks among available threads. Upon encountering a task creation point: (i) BFS will push the new task in a queue and continue execution of the parent task and (ii) WFS will suspend the parent task and start execution of the new task. BFS tends to be more demanding in terms of memory, as it creates all tasks before starting their execution (and thus all tasks coexist simultaneously). This is an undesirable property in general and in particular for resource-constrained embedded systems, which would make WFS a better candidate. WFS also has the nice property of following the execution path of the original sequential program, which tends to result in better data locality [5]. However, when *tied* tasks are used, BFS is the only choice in practice, as WFS leads to a complete serialization of task executions when nested parallelism is adopted. Moreover, it has been shown that the use of *untied* tasks significantly reduces the worst case response time analysis [9].

Task queue: The most widespread design solution to support the OpenMP tasking execution model is to rely on a centralized task queue. This minimizes memory footprint for runtime support metadata, which is a must in the context of embedded platforms. The basic building block of the proposed design focuses on lightweight support for *push* and *pop* operations on such a centralized queue (upon task creation and extraction, respectively), relying on

fine-grained locking mechanisms. TSPs are implemented using lightweight events, which avoids the massive contention implied by active polling (idle threads on the TSP are put into sleep mode). When a task is created (i.e., pushed in the queue), the creator thread sends a signal which wakes up a single thread (selected using round-robin). After completing the task execution, the thread returns into sleeping mode. The described queue is implemented with a doubly linked list. This data structure allows to *push* and *pop* tasks from the queue and also remove a task in any position of the queue. This is key for low overhead, as tasks are not constrained to execute in-order (except when dependencies are specified), so their completion and removal from the queue is independent of their position. Note that a simple linked list does not allow this operation.

Untied tasks: The described support is sufficient to show excellent performance in the presence of simple flat parallel patterns, where all the tasks are created from within a single level (i.e., a single parent task), but lacks the capability of supporting more sophisticated forms of parallelism, like nested parallel patterns found in programs that use recursion, and for which the tasking model was originally proposed. Consequently, *untied* tasks are not supported by using this basic implementation. Due to the limitations of *tied* tasks described previously, the scheduling policy relies on BFS, and WFS is not supported. In the following, we describe how we extend this baseline implementation to fully support nested parallel patterns and *untied* tasks, while keeping the implementation lightweight and not too memory-hungry. These both are the key requirements for any implementation suitable for embedded manycore accelerators. Our main goal is to achieve a comparable efficiency in terms of task granularity (the finer the better) for which near-ideal speedups are achieved.

Figure 6.2 shows how task suspension works in most implementations supporting *tied* tasks (WFS is assumed). The thread on which the code shown in the figure is executing has an associated stack (depicted on the left). When a `task` directive is encountered, the thread jumps to a runtime function that manages the creation of a new task from the enclosed code region. Because WFS is considered, the thread encountering the new task executes the code encapsulated within the task region, and the parent task is suspended (as it is a tied task and so cannot migrate to a different thread). A new stack frame is activated for this task, like in every regular function call. The same thing happens at every nested `task` directive. When a task is completed, the stack

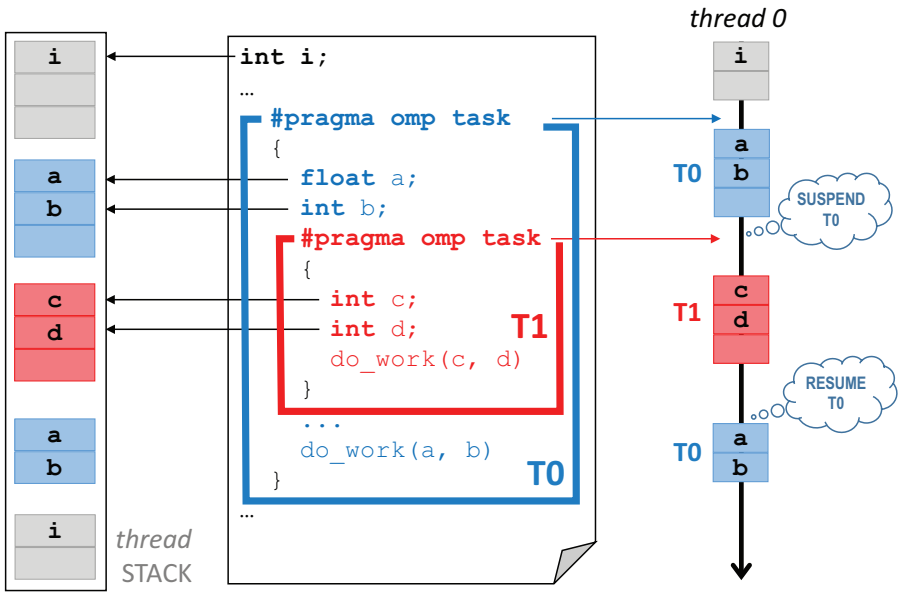


Figure 6.2 Task suspension in the baseline implementation (considering tied tasks and WFS).

pointer is reset to the top of the previous active frame. Since the semantics of *tied* task scheduling ensure that suspension/resumption can happen only on the same thread, no explicit bookkeeping to save/restore the context of a task is required.

The key extension required to support *untied* tasks is the capability of allowing to resume a suspended task on a different thread than the one that started and suspended it. To achieve this goal, we rely on lightweight *co-routines* [10]. Co-routines rely on cooperative tasks that publicly expose their code and memory state (register file, stack), so that different threads can take control of the execution after restoring the memory state. Every time that a thread suspends or resumes a suspended cooperative task, a context switch is performed. We place the required metadata to support task contexts (TCs) in the shared multi-bank memory and we use inline assembly to minimize the cost of the routines to save and restore the architectural state.

Figure 6.3 shows how task suspension works in our approach for *untied* tasks (WFS is assumed). Initially, the thread on which the code shown in the figure is executing uses its own private stack (in gray). When the outermost task region (T0) is encountered, the context of the current task is saved in the

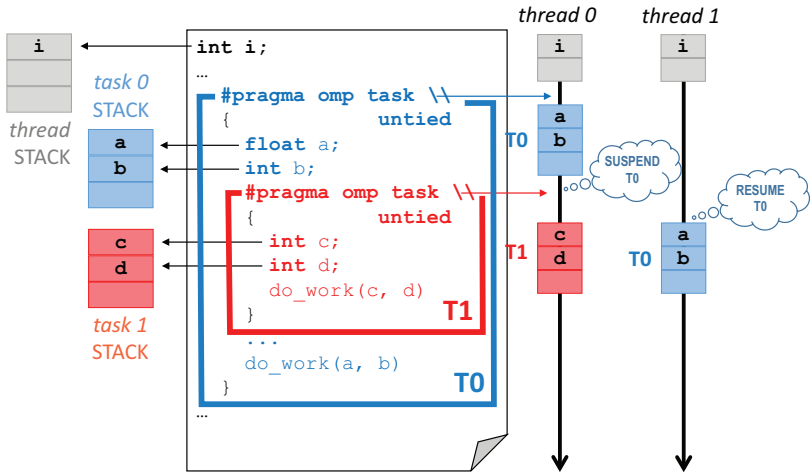


Figure 6.3 Untied task suspension with task contexts and per-task stacks.

TC (including the current SP, that is, the task pointer register), then the thread is rescheduled to execute the new task T0. The SP of the thread is updated to the stack of T0 (in blue) and the new task is started. When the creation point of the innermost task T1 is reached, an identical procedure is followed. The context of T0 is saved in its TC, which is pushed back in the queue, then thread 0 is pointed to the stack of T1 (in red). Now the suspended T0 can be pulled out of and restarted by thread 1. On top of this basic mechanism, a number of other design choices were made to minimize the cost of our runtime support, which we describe in the following.

Task hierarchy: Supporting nested tasks requires to keep in the runtime a data structure (a *tree*) that represents the hierarchy of multiple task regions. A parent task has a link to its children and vice versa, to facilitate exchange of information about execution status. For example, a parent task needs to be informed about the execution completion of its children to support the semantics of the `taskwait` directive. When a parent task completes its execution, its children become orphans and should not care to inform the parent. The fastest solution to handle parent task termination in terms of bookkeeping would be not to delete the descriptor, but just to maintain the task in a *zombie* status until all children have completed. This operation would require a simple update to the descriptor, which can be executed in a very short time. However, this solution brings to a memory occupation that

is not acceptable for our constrained platform. Thus, we opt for a costlier removal of the descriptor from the *tree*. As a consequence, all child tasks must receive an update from the parent to avoid dangling pointers to a deallocated descriptor.

Taskwait construct: Task-level synchronization is widely used in recursive-based parallel patters. Here typically a fixed number of tasks are created at every recursion level, and their execution is synchronized with a taskwait directive. When a parent task encounters a taskwait, it should wait until all the children (first-level descendants) have completed, but typically for performance the thread hosting the parent task is allowed to switch to executing one of the children tasks. In the baseline implementation, this feature is supported by just traversing the list of children tasks in the *tree* data structure and inspecting their status to verify that it is set to WAITING. We changed this mechanism to rely on two queues per task, to directly reference children in the WAITING and RUNNING states, respectively. Upon creation, a task is inserted in the WAITING queue. Every time that a task starts to execute, the runtime moves this task from the WAITING queue to the RUNNING queue, and vice versa in case of suspension. Decoupling waiting and running tasks require a costlier bookkeeping upon task insertion and extraction, but allow faster support for taskwait as it is no longer required to search the tree for WAITING tasks. While the benefit brought by this implementation is not evident in the presence of flat parallel patterns, as the taskwait is virtually useless in this case, in recursive parallel patterns, it is extensively used and this design choice pays off.

Task dependencies: In the presence of recursive parallel patterns, it is important to distinguish between suspended tasks that could be resumed at any time and tasks that are suspended due to a scheduling constraint that needs to be unblocked. A typical example is, again, tasks suspended upon a taskwait or due to a data dependence. As already mentioned, recursive parallelism extensively relies on such a form of synchronization, thus hosting this type of suspended tasks in the same queue that also hosts ready-to-execute tasks used to lead to a situation where we would repeatedly pop from there a task just to realize that the scheduling constraint was still unsatisfied. We would then have to push back the task in the queue and retry. Checking the status of the task before extracting it does not entirely solve the problem, as it requires time-consuming search operations. To deal with this problem, we changed the implementation to avoid re-inserting in the queue suspended tasks with

unresolved dependencies. Such tasks are kept floating instead, and it is up to the task that will eventually resolve the dependence to push them back into the queue. This modification requires some additional checks to deal with the above-mentioned case, but greatly improves the performance of recursive parallel programs.

Allocation of runtime metadata: To minimize the overhead for dynamic resource allocation (memory, locks, task descriptors, etc.), we have extensively used pools of pre-allocated resources. This is significantly faster than *malloc*-like primitives and does not require lock-protected operations, as we adopt thread-private resources. The downside is memory occupation. Since the targeted architecture relies on a shared cluster memory with a limited size, we have to wisely use the available space. A reasonable design solution would be to dedicate roughly 5–10% of this memory to hosting tasking support data structures. The original task descriptor has a size of 174 bytes, while the extensions that we introduced require another 98 bytes for the contexts, plus the stacks. Private thread stacks are configured to be 1 KB (a common choice for embedded systems), while task stacks are by default 1/4 of that size. Clearly, all those values are parameters in our design, and can be changed depending on specific application requirements.

Despite the increment of runtime memory requirements, the use of pre-allocated resources enables to exploit finer grained parallelism, which is paramount in current and future embedded systems. Next, we describe solutions to reduce memory pressure and runtime overhead.

Cutoff mechanisms: With 10% of the cluster's shared memory allocated to task descriptors, the runtime can host simultaneously 750 pre-allocated *tied* tasks or 400 *untied* tasks. If the queue of available task descriptors is depleted during the program execution, a mechanism (known in the literature as *cutoff* [11]) is triggered. When this condition is met, the creation of new task descriptors must be suspended to avoid that runtime resources saturate when the task production rate is greater than the execution rate. Our runtime supports two different cutoff variants: *yield* and *work-first*. In the first case, the producer task is stopped and pushed at the end of the READY queue, with the aim to re-schedule the core to executing pending tasks instead of generating new ones. Using the second variant, the producer task starts working in work-first mode by executing the new tasks in-place via a standard function call: in this case, task descriptors are not required, as the synchronization is enforced by serializing tasks on the same thread.

Cutoff mechanisms are introduced to avoid an unbounded consumption of runtime resources, but recursive applications can cause additional problems. Using *untied* tasks, task stacks typically end up to be over-sized to fit the worst case (i.e., the maximum recursion level reached in the cutoff state) to the detriment of runtime memory footprint. To avoid this case, we introduced a specific optimization for *untied* tasks using *work-first* cutoff, which forces the producer task to swap its current stack with a special one that is the only one dimensioned for worst case recursive execution.

Support for scheduling policies: The OpenMP runtime provides specific features to support the scheduling policies that have been defined in Chapter 4. Two alternative implementations are selectable for task queues: *global* and *private* queues. The global implementation defines a single task queue for the application, and it is used to support *global scheduling*. The local implementation instantiates an independent queue per thread, and it is used to support *partitioned scheduling*, in which tasks are statically allocated to threads at design time.

Adopting a *limited preemption* scheduler, each TSP in the runtime is considered as a potential preemption point. This is implemented by calling a function designed and implemented for tight integration with the RTOS. The exact behavior depends on the current scheduling policy (*global* or *partitioned*) selected for the application, which is totally transparent to the runtime.

6.3.1 Task Dependency Management

The OpenMP tasking model includes a very mature support for highly unstructured task parallelism with features to express data dependencies (on specific data elements) between tasks. To do so, OpenMP introduces the depend clause, which imposes an ordering relation between sibling tasks (tasks that are child tasks of the same task region). OpenMP defines three types of dependencies: in, out, and inout. A task with an in clause cannot start until the set of tasks with an out or an inout clause on the same data elements complete. This feature is in fact very relevant for embedded systems, often running real-time applications modeled as *direct acyclic graphs* (DAGs)[2] (see Chapter 4 for further information).

[2]The terms TDG and DAG are equivalent; the former is typically used when referring to runtime methodologies; the latter is used when referring to real-time analysis.

Current implementations of the OpenMP tasking model targeting the high-performance domain (e.g., libgomp, nanos++) track data dependencies among tasks by building a *task dependency graph* (TDG) at runtime. When a new task is created, its in and out dependencies are matched against those of the existing tasks. To do so, each task region maintains a *hash table* that stores the memory address of each data element contained within the out and inout clauses, and the list of tasks associated to it. The hash table is further augmented with links to those tasks depending on it, i.e., including the same data element within the in and inout clauses. In this way, when the task completes, the runtime can quickly identify its successors, which may be ready to execute.

Building the TDG at runtime requires storing the hash tables in memory until a taskwait directive is encountered. Since dependencies can be defined only between sibling tasks, when such directives are encountered, all tasks in their binding region are guaranteed to finish. Moreover, removing the information of a single task at completion would result too costly, because dependent tasks are tracked in multiple linked lists in the hash table. As a result, the memory consumption may significantly increase as the number of instantiated tasks increases.

Such a memory consumption is clearly not a problem in high-performance systems, in which large amounts of memory are available. However, this is not in general the case for parallel embedded architectures. The MPPA processor features only 2 MB of on-chip private memory per cluster. Therefore, it is paramount to devise data structures that reduce to the bare minimum the memory requirements needed to implement the TDG.

To this aim, we maintain the complete OpenMP-DAG generated by the compiler as presented in Chapter 3. *Although this idea may seem counter-intuitive, the data structures needed to store a statically generated TDG are much lighter than those necessary to dynamically build the TDG*. This strategy results in a huge reduction of the memory used at runtime.

TDG Data Structure: A Sparse Matrix – A *sparse matrix* is an optimal solution to store the TDG with minimal footprint. Figure 6.4b shows the sparse matrix implementation of the DAG presented in Figure 6.4a. There, each entry contains a unique task instance identifier t_{id}, and stores in separate arrays the t_{id} and the number of tasks it depends on (labeled *Inputs* and *#in* respectively in the figure), and the t_{id} and the number of tasks depending on it (labeled *outputs* and *#out* respectively in the figure). Moreover, the sparse matrix is sorted using the t_{id}, so a dichotomic search can be applied.

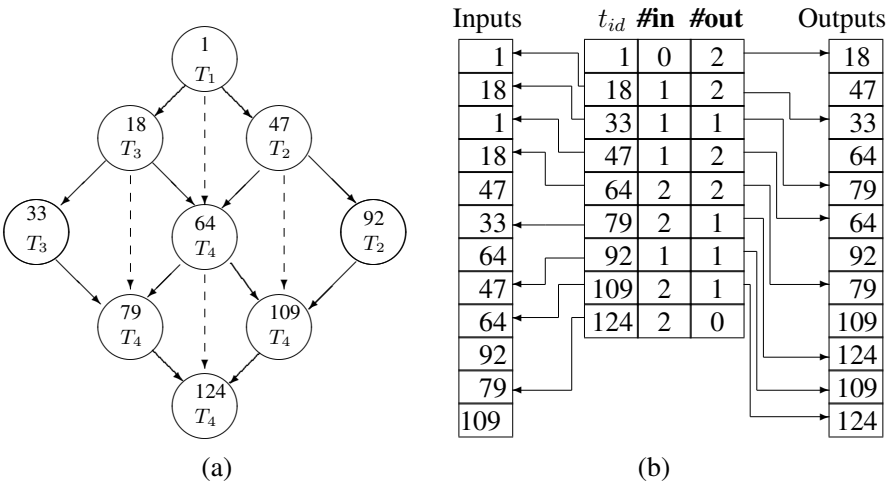


Figure 6.4 On the left (a), the DAG of an OpenMP program. On the right (b), the sparse matrix data structure implementing DAG shown on the left.

t_{id}, computed with Equation 6.1 (also presented in Chapter 3), is a key mechanism used to identify the tasks actually instantiated at runtime with those included in the DAG. Therefore, the same value of t_{id} must be generated at compile time (so each node in the DAG has a unique identifier) and at runtime (so tasks can identify its input and output data dependencies).

$$t_{id} = sid_t + T \times \sum_{i=1}^{L_t} l_i \cdot M^i \tag{6.1}$$

where sid_t is a unique task construct identifier, T is equal to the number of task, taskwait, and 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, and M is maximum number of iterations of any considered loop.

All the information required to compute Equation 6.1 must therefore be available at compile time. sid_t is inserted by the compiler as a new parameter in the function call of the tasking runtime in charge of creating a new OpenMP task (named GOMP_task). In order to obtain the same l_i at compile-time and at runtime, the compiler introduces a *loop stack* per loop statement, and *push* and *pop* operations before the loop begins and after it ends, respectively. At every loop iteration, the top of the stack is increased by 1. The overhead

associated to the stack is very little because it is inserted only in those loops where tasks are created and the overhead due to the task creation dominates. The rest of parameters, i.e., T, L_t, and M are encapsulated in the TDG data structure.

Consider task T_4, with identifier 79, in Figure 6.4a. 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 all together: $T_{4_{id}} = 4 + 5(2 * 3^1 + 1 * 3^2) = 79$.

Finally, with the objective of monitoring the execution state of task instances, each entry in the sparse matrix has an associated counter (not shown in the figure) describing its state. The counter is:

- -1 if the task has not been instantiated (created) or it has finished;
- 0 if the task is ready to run; and
- > 0 if the task is waiting its input tasks to finish. The value indicates the number of tasks created and not completed it still depends on.

The runtime task scheduler works as follows:

- When a new task is created, the runtime checks the state of its *input* tasks. If all their counters are -1, the task is ready to execute; otherwise, the state of the counter of the new task is initialized with the number of input tasks with a state ≥ 0.
- When a task finishes, it decrements by 1 the counters of all its output tasks whose counter is > 0.

It is important to remark that, when the TDG contains tasks whose related if-else statement condition has not been determined at compile time and it evaluates to $false$ at runtime, the value of the counter is the same as the tasks would have already finished, i.e., -1 (see Chapter 3 for further information).

6.4 Experimental Results

In the following, we present results aimed at characterizing the overheads of the proposed OpenMP runtime design and demonstrating the reduced impact on the overall application performance, compared to different solutions.

6.4.1 Offloading Library

Synchronization on the Kalray MPPA architecture has a significant impact on the offloading cost. The preliminary implementation of the **BLOCK_OS** policy, which has the most complex semantics among all, required 75,500 cycles to initialize the runtime metadata. It is possible to halve the initialization cost by (i) replacing dynamic memory allocation of runtime data structures with a static memory mapping and (ii) distributing between the available cores the initialization of data structures.

As a further optimization, we implemented a lightweight runtime check of the presence of a pending offload request to prevent the RTOS from executing the RM when no new offload requests to process are present. This further reduced the initialization cost to 33,250 cycles. Figure 6.5 reports the offload cost on the cluster side for different synchronization policies. We report minimum and maximum observed execution cycles (blue and orange bars, respectively). The leftmost groups of bars represent the original Kalray software infrastructure, while the three rightmost groups of bars represent the three policies of our software infrastructure. The results for Kalray show a very large variance between minimum and maximum observed offload cost. Anyhow, since the analysis tools rely on worst case execution time,

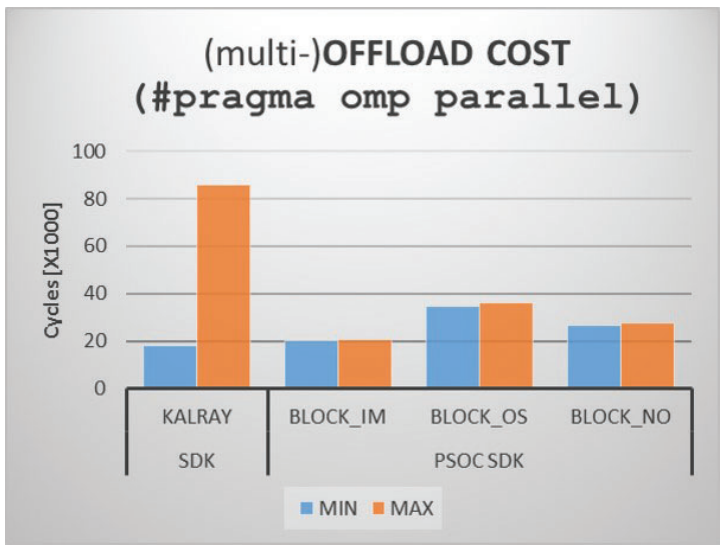


Figure 6.5 Costs of offload initialization.

to all practical purposes, we must consider the maximum time, which is around 82,000 cycles. All the three synchronization policies that we provide exhibit a very small variance, and their cost is in all cases much smaller than the worst case for the original Kalray SDK (roughly in line with the best case).

This notwithstanding, the observed costs for our runtime software are still relevant. Compared to state-of-the-art solution, we identified the main reason of this inefficiency in the management of non-coherent caches. A flush-and-invalidate operation on the data caches is performed at every synchronization point (the *unlock* primitive). This makes each access to runtime data structures very expensive in terms of execution cycles. Replacing data caches with L1 scratchpad memories and using these memories to store runtime data structures allow reducing the offload cost by 20x.

6.4.2 Tasking Runtime

As already pointed out, supporting the tasking execution model is usually subject to large overheads. While such overheads can be tolerated by large applications exploiting coarse-grained tasks, this is usually not the case for embedded applications, which rely on fine-grained workloads. To study this effect, our plots show speedup (parallel execution on 16 cluster cores versus sequential execution on a single cluster core) on the y-axis, comparing the original Kalray runtime to our runtime support for *tied* and *untied* tasks. For all the experiments except the one in Section 6.4.2.5, we use a set of microbenchmarks in which tasks only consist of ALU operations (e.g., add on local registers) and no load/store operations, which allows exploring the maximum achievable speedups. The number of ALU operations within the tasks can be controlled via a parameter, which allows studying the achievable speedup for various task granularities, which we report on the x-axis of each plot (task granularity is expressed as duration in clock cycles, roughly equivalent to the number of ALU operations that each task contains).

We consider three variants for the synthetic benchmark: LINEAR, RECURSIVE, and MIXED. These are representative of different task creation patterns found in real applications, and will be described in the following subsections.

6.4.2.1 Applications with a linear generation pattern

The LINEAR benchmark consists of $N = 512$ identical tasks, each with a workload of W ALU instructions. The main task creates all the remaining

N... 1 tasks from a simple loop (one task created per loop iteration) and then performs a taskwait to ensure that all tasks have completed their execution.

```
for (i=0; i<N; i++)
{
  #pragma omp task    // A task consisting
  synth (W);          // of W ALU instructions
}
#pragma omp taskwait
```

Figure 6.6 shows the results for the LINEAR benchmark. Focusing on the results for the original Kalray SDK ("noCO KALRAY" line), ideal speedups can be achieved only for tasks larger than 100 KCycles. For smaller tasks, the maximum achievable speedup is 3×. In this fine-grain task area, *our* tasks can consistently achieve a four times higher speedup. Since in the LINEAR microbenchmarks, there is no task nesting, there is no significant difference between *tied* (PSOC T) and *untied* (PSOC U) tasks. We thus explore a new configuration where tasks are recursively created to appreciate the difference.

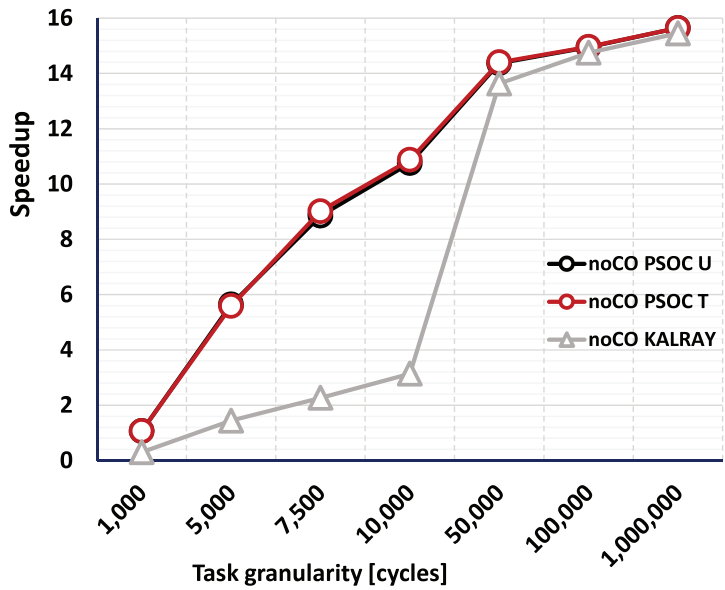


Figure 6.6 Speedup of the LINEAR benchmark (no cutoff).

6.4.2.2 Applications with a recursive generation pattern

Figure 6.7 shows the efficiency of our runtime for the recursive parallel pattern, considering *tied* and *untied* tasks. The RECURSIVE microbenchmark builds a binary tree of depth N = 9 (512 tasks) recursively. This is similar to a classical Fibonacci algorithm, where each of the two recursive calls is enclosed in a task directive. A taskwait directive is placed after the creation of the two tasks.

```
#pragma omp task  // The first task (root)
rec (0, 511);

int rec(int level, int maxlevel)
{
 if ( lev != maxlevel )
 {
    #pragma omp task          // Fist child task
    rec (level+1, maxlevel);
    #pragma omp task          // Second child task
    rec (level+1, maxlevel);
 }

 synth (W);       // W ALU instructions

 #pragma omp taskwait
}
```

The first result that we observe is that only *untied* tasks can achieve the maximum speedup. *Tied* tasks have a maximum speedup of 8. This effect is due to the behavior of taskwait in the presence of *tied* tasks. If a *tied* task is stuck on a taskwait and there are no children tasks in the WAITING state (e.g., few tasks generated at each recursion level, like in the binary tree), that task is bound to wait until the children have finished. Using a binary tree, this leads to exactly half of the threads getting stuck, which explains the maximum speedup observed in this configuration. This problem is circumvented by *untied* tasks, which can reschedule the threads hosting the stuck tasks to other ready tasks. Similar considerations to what we discussed in the previous section hold for the comparison between Kalray tasks and

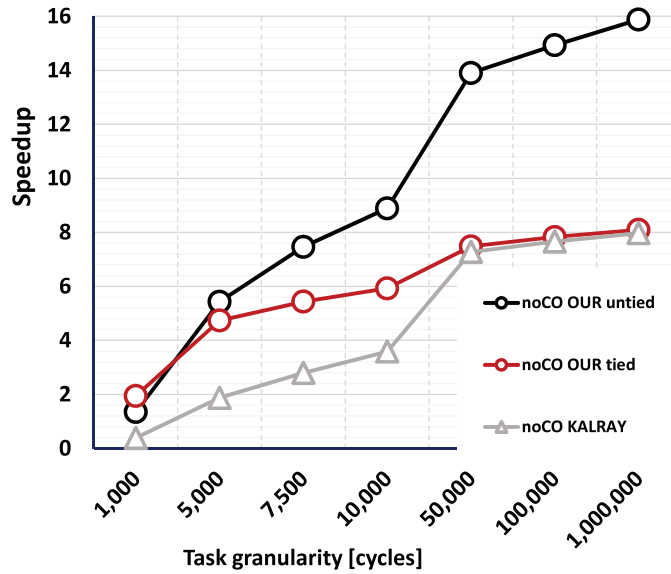


Figure 6.7 Speedup of the RECURSIVE benchmark (no cutoff).

our tied tasks (Kalray supports only *tied* tasks, so a comparison to *our untied* tasks is not directly feasible).

In general, it is possible to see that RECURSIVE implies a much higher overhead than LINEAR. This is justified by a significantly increased contention for shared data structures (queues, trees, etc.), as in this pattern multiple threads are concurrently creating tasks. Even if we have struggled to make the lock-protected operations to operate on shared data structures as short as possible, their serialization over multiple requestors is evident. As a result, it takes an order of magnitude coarser tasks (around 100 K) than in the LINEAR case to achieve nearly ideal speedups. This is a typical situation where cutoff policies can help in significantly reducing the runtime overheads. We explore the adoption of cutoff policies in Section 6.4.2.4.

6.4.2.3 Applications with mixed patterns

The advantage of using *untied* tasks is particularly evident for applications presenting a mixed structure which includes both LINEAR and RECURSIVE task creation patterns. The MIXED microbenchmark depicted in Figure 6.8

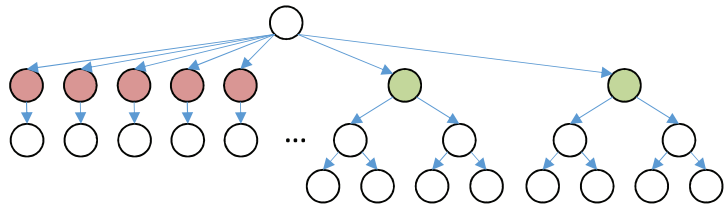


Figure 6.8 Structure of the MIXED microbenchmark.

is aimed at studying the behavior of such applications. A root task generates seven tasks in a LINEAR manner, each one spawning a single child with a long execution time and then performing a `taskwait`, plus another two tasks from within RECURSIVE binary trees of depth 5.

Figure 6.9 shows the results for this benchmark. Using *tied* tasks, 14 threads are allocated to execute the linear part of the application, seven of which are blocked by the `taskwait` directive. The ideal speedup of the application is 2, which our *tied* tasks reach for granularities of around 10 Kcycles.

Using *untied* tasks, only seven threads are allocated to the LINEAR part, which brings the ideal speedup to 9×. The maximum speedup achieved by our *untied* tasks is 8, due to a limitation of the tracing (performance

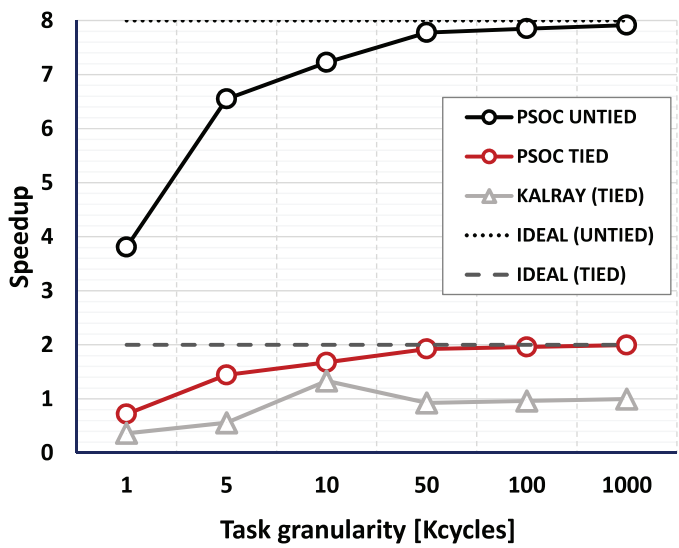


Figure 6.9 Speedup of the MIXED benchmark.

monitoring) of the Kalray platform. The root task of the hierarchy is the one performing time measurement and we were forced to declare this as a *tied* task to gather coherent clock values (allowing this task to migrate to other cores results in incoherent measurement). This limits the maximum achievable speedup to 8×, which our *untied* tasks achieve for granularities above 10 Kcycles.

Overall, *untied* tasks enable four times faster execution than *tied* tasks for application featuring mixed task creation patterns. Note that this result holds for any runtime implementation. Our solution makes this result visible for smaller tasks compared to other OpenMP tasking implementations. The Kalray implementation never enables any speedup in the considered range of task granularities (up to one million cycles) for this experiment.

6.4.2.4 Impact of cutoff on LINEAR and RECURSIVE applications

We repeated the experiments with LINEAR and RECURSIVE microbenchmarks considering a higher number of tasks (2,048). This configuration saturates the runtime data structures and activates *cutoff* mode. Figures 6.10 and 6.11 show the results for this experiment.

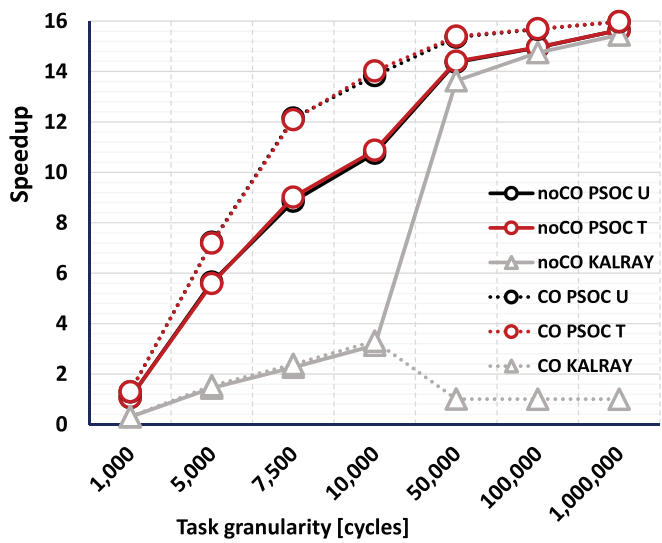


Figure 6.10 Speedup of the LINEAR benchmark (with cutoff).

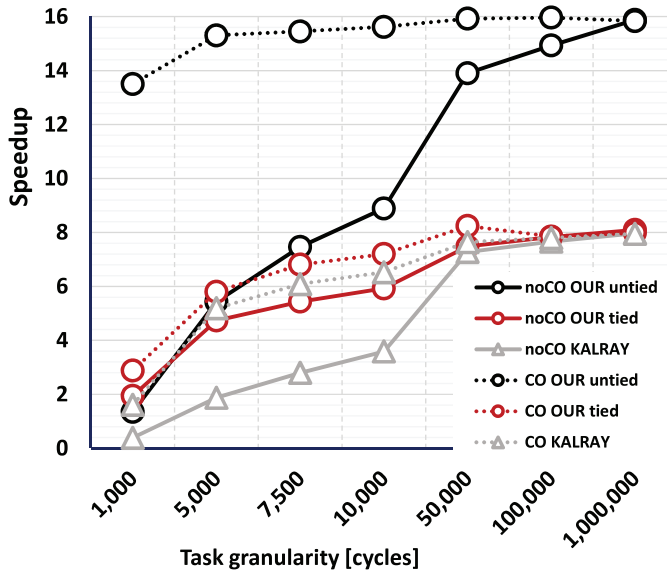


Figure 6.11 Speedup of the RECURSIVE benchmark (with cutoff).

Focusing on the LINEAR pattern, the adoption of cutoff greatly mitigates overhead effects, and we can achieve nearly ideal speedups for an order of magnitude smaller tasks compared to Kalray tasks. It also has to be noted that cutoff mode is not properly supported for LINEAR patterns in the original Kalray runtime. Enabling cutoff mode in this configuration simply seems to disable parallelism completely. Focusing on the RECURSIVE pattern, the use of cutoff policies proves extremely beneficial, with nearly ideal speedups for very fine-grained tasks (in the order of thousand cycles).

6.4.2.5 Real applications

To assess the performance of our tasking runtime on real applications, we execute the benchmarks from the Barcelona OpenMP Task Suite (BOTS) [12], which includes a wide set of real-life applications parallelized with OpenMP tasks.

Figure 6.12 shows the speedup of applications for different configurations, comparing the Kalray SDK (KALRAY) with different configurations of our runtime, using *tied* tasks (PSOC tied), *untied* tasks (PSOC untied), and *untied* tasks with cutoff (PSOC untied CO2).

On average, programs executing on top of our runtime show a speedup of 12×, compared to only 8× for the original Kalray SDK. The benefits of cutoff

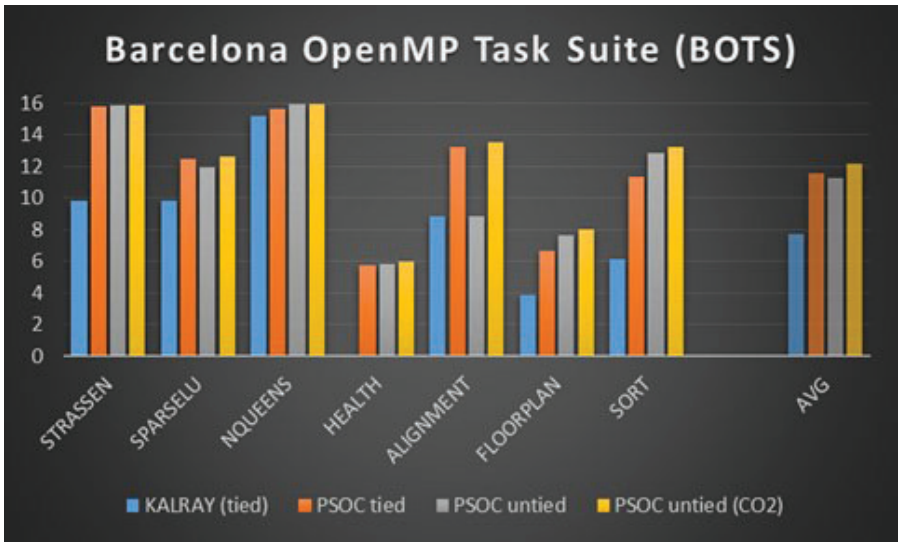


Figure 6.12 Speedups for the BOTS benchmarks.

here are minimal, since the bottleneck is limited parallelism in the application rather than runtime overhead. The marginal improvements enabled by cutoff, where present, are usually due to better memory usage (tasks in cutoff use less memory for the runtime, which is used for application data instead).

6.4.3 Evaluation of the Task Dependency Mechanism

This section evaluates the use of a sparse matrix to implement the TDG upon which the task dependency mechanism is built as presented in Section 6.3.1.

Concretely, we implement our task dependency mechanism on top of the GNU libgomp library included in GCC version 4.7.2, which supports tasks but not dependencies, and compare it with the libgomp library included in GCC 4.9.2, which implements a dependency checker based on a hash table structure.

The reason to implement our mechanism on a library not supporting dependencies is that both implementations differ only in the dependency checker, and so being easier to incorporate a new one, rather than replacing it. Moreover, to ensure that results are not affected by the version of the library, we executed the applications considered in this section without dependence clauses. Despite the incorrect result, the numbers revealed that both libraries

have the exact same memory usage and performance, demonstrating that the memory increment is exclusively caused by using different dependency checkers.

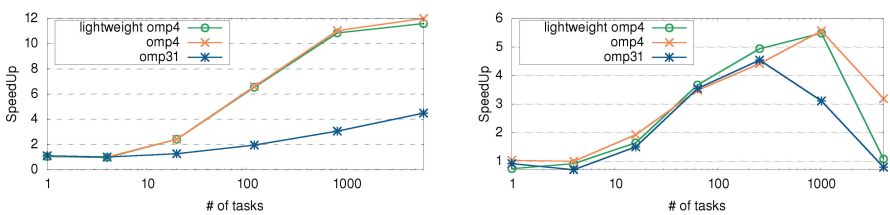
Moreover, we consider two applications, one from the HPC domain, i.e., a *cholesky factorization* [13] used for efficient linear equation solvers and Monte Carlo simulations, and one from the embedded domain, i.e., an application resembling the *3D path planning* [14] (r3DPP) used for airborne collision avoidance.

For comparison purposes, the applications have been parallelized with task dependencies, i.e., using the `depend` clause, and without dependencies, i.e., using only `task` and `taskwait` directives.

6.4.3.1 Performance speedup and memory usage

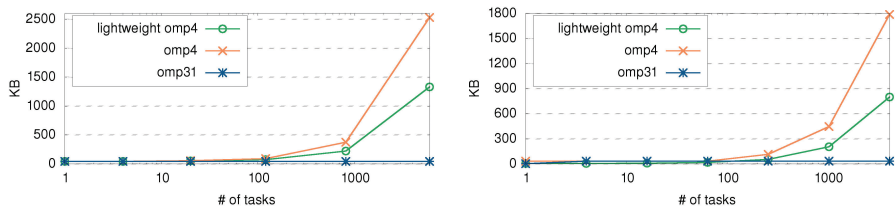
Figures 6.13 and 6.14 show the performance speedup and the runtime memory usage (in KB) of the Cholesky and r3DPP, when varying the number of instantiated tasks, ranging from 1 to 5984 and 4096, respectively, and considering the three libgomp runtimes implementing a dependency checker based on a hash table, on a sparse matrix, and one with not dependency checker (labeled *omp4*, *omp 3.1*, and *lightweight omp4*, respectively).

The performance has been computed with the average of 100 executions. Similarly, Figures 6.14a,b show the heap memory usage (in KB) of the three OpenMP runtimes when executing Cholesky and r3DPP respectively and varying the number of instantiated tasks as well. The memory usage has been extracted using *Valgrind Massif* [15] tool, which allows profiling the heap memory consumed by the runtime in which the TDG structure is maintained.



(a) Cholesky (b) r3DPP

Figure 6.13 Performance speedup of the Cholesky (a) and r3DPP (b) running with *lightweight omp4, omp4*, and *omp 3.1*, and varying the number of tasks.



(a) Cholesky (b) r3DPP

Figure 6.14 Memory usage (in KB) of the Cholesky (a) r3DPP (b) running with *lightweight omp4, omp4*, and *omp 3.1*, and varying the number of tasks.

For these experiments, we consider an Intel Xeon CPU E5-2670 processors, featuring eight cores each, with 20 MB L3. The reason is that it incorporates the libgomp library included in GCC 4.9.2 supporting dependency checker based on a hash table.

We observe that both performance and memory usage depend on the number of instantiated tasks: the higher the number of instances, the better the performance, as the chances of parallelism increase. When the number of tasks is too high, however, the overhead introduced by the runtime and the small workload of each task slows down the performance.

As shown in Figure 6.13, our *lightweight omp4* obtains the same performance speedups as the *omp4* implementation for the two applications, and outperforms *omp 3.1*. However, when observing the memory usage in Figure 6.14, it rapidly increases for *omp4*, requiring much more memory than the runtime based on the sparse matrix, i.e., the *lightweight omp4*.

It is also interesting to observe the parallelization opportunities brought by the depend clause, which makes the performance of Cholesky (Figure 6.13a) to increase significantly compared to not using them, with a speedup increment from 4x to 12x when instantiating 5,984 tasks. At this point, *omp4* consumes 2.5 MB while our *lightweight omp4* requires less than 1.3 MB. The memory consumed by *omp3.1* is less than 100 KB (Figure 6.14a). In fact, the *omp3.1* memory consumption is similar for all the applications because no structure for dependencies management is needed.

For the r3DPP, the depend clause achieves a performance speedup of 5.2x and 5.8x with *omp4* and *lightweight omp4*, respectively, when instantiating 1,024 tasks (Figure 6.13b). At this point, *omp4* consumes 400 KB in front of the 200 KB consumed by *lightweight omp4* (Figure 6.14b). Not considering dependencies, i.e., *omp31*, achieves a maximum performance of 4.5x when 256 tasks are instantiated (Figure 6.13b). When the number of task instances

Table 6.1 Memory usage of the sparse matrix (in KB), varying the number of tasks instantiated

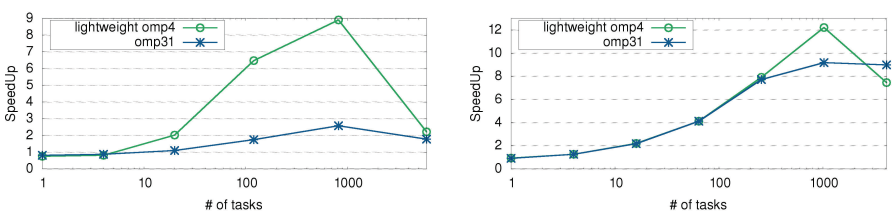
Cholesky	**Tasks**	4	20	120	816	5984
	KB	0.11	0.59	3.80	27.09	204.19
r3DPP	**Tasks**	16	64	256	1024	4096
	KB	00.47	1.94	7.88	31.75	127.5

increases to 4096, all runtimes suffer a significant performance degradation because the number of instantiated tasks is too high compared to the workload computed by each task.

Table 6.1 shows the size of the sparse matrix data structure implementing the esTDG of each application when varying the number of instantiated tasks (the memory consumption reported in Figures 6.14a,b already includes it).

6.4.3.2 The task dependency mechanism on the MPPA

To evaluate the benefit of the task dependency mechanism on a memory constrained manycore architecture, we evaluated it on the MPPA processor. Figure 6.15 shows the performance speedup of Cholesky (a) and r3DPP (b) executed in one MPPA cluster, considering the *lightweight omp4* and *omp31* runtimes and varying the number of tasks. Note that *omp4* runtime experiments are not provided because MPPA does not support it. Memory consumption is the same as the one shown in Figure. 6.14 r3DPP increases the performance speedup from 9x to 12x when using our *lightweight omp4* rather than *omp3.1* and only consuming 200 KB. Cholesky presents a significant speedup increment when instantiating 816 tasks, i.e., from 2.5x to 9x, consuming only 220 KB.



(a) Cholesky (b) r3DPP

Figure 6.15 Performance speedup of the Cholesky (a) and r3DPP (b) running on the MPPA with *lightweight omp4, omp4*, and *omp 3.1*, and varying the number of tasks.

6.5 Summary

This chapter has illustrated the design of the OpenMP runtime for a heterogeneous platform including a *host* processor and an embedded manycore accelerator. The complete software stack is composed of an offloading library and a tasking runtime library, which have been described in detail. The OpenMP runtime provides specific features to support the scheduling policies that have been defined in Chapter 4, and it also implements the TDG required to support the task dependency mechanism as presented in Section 6.3.1. The chapter has discussed how to enable maximum exploitation of the available hardware parallelism via the *untied* task model, highlighting the key design choices to achieve low overhead. Experimental results show that this enables up to four times faster execution than *tied* tasks, which improves on average by 60% over the native Kalray SDK.

References

[1] Marongiu, A., Capotondi, A., Tagliavini, G., and Benini, L., "Simplifying Many-Core-Based Heterogeneous SoC Programming With Offload Directives." In *IEEE Transactions on Industrial Informatics*, vol. 11, pp. 957–967, 2015.

[2] Mitra, G., Stotzer, E., Jayaraj, A., and Rendell, A. P., "Implementation and optimization of the OpenMP accelerator model for the TI Keystone II architecture." In *International Workshop on OpenMP*, Springer, pp. 202–214, 2014.

[3] Rosenstiel, W., and Thiele, L. (editors), *Design, Automation and Test in Europe Conference and Exhibition, DATE 2012, Dresden, Germany.* IEEE, 2012.

[4] Podobas, A., Brorsson, M., and Faxén, K.-F., A comparative performance study of common and popular task-centric programming frameworks. *Concurr. Comput. Pract. Exp.* 27, 1–28, 2015.

[5] Duran, A., Teruel, X., Ferrer, R., Martorell, X., and Ayguade, E., "Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP." In *2009 International Conference on Parallel Processing*, pp. 124–131. IEEE, 2009.

[6] Burgio, P., Tagliavini, G., Marongiu, A., and Benini, L., "Enabling fine-grained OpenMP tasking on tightly-coupled shared memory clusters." In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, pp. 1504–1509. EDA Consortium, 2013.

[7] Rochange, C., Bonenfant, A., Sainrat, P., Gerdes, M., Lobo, J., et al., "WCET analysis of a parallel 3D multigrid solver executed on the MERASA multi-core." In *WCET*, 2010.

[8] Kumar, S., Hughes, C. J., and Nguyen, A, "Carbon: Architectural Support for Fine-grained Parallelism on Chip Multiprocessors." In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pp. 162–173. ACM, 2007.

[9] Serrano, M. A., Melani, A., Vargas, R., Marongiu, A., Bertogna, M., and Quiñones, E., "Timing Characterization of OpenMP4 Tasking Model." In *Proceedings of the 2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '15, pp. 157–166. IEEE Press, 2015.

[10] Marlin, C. D., *Coroutines: a programming methodology, a language design and an implementation*. Number 95 in Lecture Notes in Computer Science. Springer Science and Business Media, 1980.

[11] Duran, A., Corbalán, J., and Ayguadé, E., "Evaluation of OpenMP task scheduling strategies." In *International Workshop on OpenMP*, pp. 100–110. Springer, 2008.

[12] Duran, A., Corbalan, J., and Ayguade, E., "An adaptive cut-off for task parallelism." In *2008 SC – International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–11. IEEE, 2008.

[13] Bascelija, N., Sequential and Parallel Algorithms for Cholesky Factorization of Sparse Matrices. *WSEAS: Mathematic. Appl. Sci. Mech.* 2013.

[14] Cesarini, D., Marongiu, A., and Benini, L., "An optimized task-based runtime system for resource-constrained parallel accelerators." In *2016 Design, Automation and Test in Europe Conference and Exhibition, DATE 2016*, Dresden, Germany, pp. 1261–1266, 2016.

[15] Nethercote, N., et. al., "Building Workload Characterization Tools with Valgrind." In *IISWC*, 2006.