A Lightweight OpenMP4 Run-time for Embedded Systems

Roberto E. Vargas†, Sara Royuela†, Maria A. Serrano†, Xavier Martorell†‡, Eduardo Quiñones†
†Barcelona Supercomputing Center (BSC), Barcelona, Spain
‡Universitat Politecnica de Catalunya (UPC), Barcelona, Spain
{roberto.vargas, sara.royuela, maria.serranogracia, xavier.martorell, eduardo.quinones}@bsc.es

Abstract— OpenMP is increasingly being adopted by current many-core embedded processors to exploit their parallel computation capabilities. Unfortunately, current run-time implementations of the latest specification (v4.0) are not suitable for processors relying on small and fast on-chip memories, due to its memory consumption. This paper proposes an OpenMP4 run-time that reduces the memory consumption while providing the same performance. Our run-time relies on a new compiler pass capable to generate the task dependency graph of OpenMP programs, which is then efficiently stored in memory.

I. Introduction

Modern many-core embedded platforms provide the level of performance required to face current and future challenges of embedded systems. These platforms incorporate suitable programming models to exploit their massive parallel computation capacities. This is the case of OpenMP, the de-facto standard for shared-memory architectures and widely adopted in the high-performance computing (HPC) domain. Recently, due to the introduction of new many-core embedded architectures (e.g. Kalray MPPA [3], STM P2012 [6], TI Keystone II [10]), OpenMP has also gained a lot of attention in the embedded domain [18][21][9]. Originally focused on massively data-parallel loop-intensive applications, the latest OpenMP specification [2] (OpenMP4) has evolved to consider dynamic, fine-grained and irregular parallelism. This includes a mature support for highly-unstructured task parallelism with features to express task dependencies. These features are very relevant for embedded systems, often running real-time applications modeled as *task dependency graphs* (TDG) [21].

Current OpenMP4 run-time implementations (e.g. libgomp [1], nanos++ [7]) require large data structures in memory to manage the tasking model. Modern many-core embedded designs, however, rely on computing fabrics with small on-chip memories that are accessible by a limited number of cores (usually organized in clusters), making these run-times unsuitable. This is the case of the MPPA processor, a many-core composed of 16 clusters of 16 cores each, coupled with 2 MB of private on-chip memory per cluster. As a result, MPPA (like other many-core embedded processors) only supports older OpenMP specifications (v3.1) with no task dependency features. There is therefore a need to implement memory efficient OpenMP4 run-times to fully exploit the performance opportunities of these platforms.

This paper proposes a new lightweight OpenMP4 run-time that reduces the memory consumed by the tasking data structures, while maintaining the same performance of current implementations. To do so, our run-time relies on a new compiler pass capable of generating the TDG of OpenMP programs including all task instances and dependencies that can exist at run-time. Then, our run-time efficiently stores and manages the statically generated TDG to determine the order in which tasks are executed. Our results show that this strategy reduces the run-time memory usage, enabling the execution of OpenMP4 programs in memory-constrained environments such as the MPPA.

II. OpenMP4 Tasking Model

OpenMP defines a *thread* as an execution entity associated with a stack and static memory, and a *task* as a specific instance of executable code generated when a thread encounters a `task` construct or a `parallel` construct. The region of code associated to a task construct is called *task region*.

An OpenMP program starts with a single thread of execution, called *initial*. This thread runs sequentially until it encounters a `parallel` construct, when a new *team* of threads is created. The number of threads in the team may be forced with the `num_threads` clause.

Tasks may be synchronized in two manners: (1) *task dependences*, by means of the clause `depend`, and (2) *synchronization constructs*, by means of the `taskwait` or the `barrier` directives. The `depend` clause 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. The synchronization directives instead, specify a waiting point on completion: the `barrier` construct implies all threads in the binding parallel region to wait for the others to complete; the `taskwait` construct implies the binding thread waits only for those tasks that are child of the binding task region. Note that all tasks bound to a given parallel region are guaranteed to have completed at the *implicit synchronization barrier* at the end of the parallel region.

Listing 1 shows an OpenMP program that processes the elements of a blocked 2D matrix using a *wave-front* parallelization strategy [8]. The `parallel` construct (line 1) defines

Listing 1 Example of an OpenMP program.

```
1 #pragma omp parallel num_threads(8)
2 #pragma omp master              // Task region T0
3   for(int i=0; i<=2; i++) {
4     for(int j=0; j<=2; j++) {
5       if(i==0 && j==0) {       // Initial block
6         #pragma omp task depend(inout:m[i][j])
7         compute_block(i, j);   // Task region T1
8       } else if (i == 0) {     // Blocks in upper edge
9         #pragma omp task depend(in:m[i][j-1],
10                                inout:m[i][j])
11        compute_block(i, j);   // Task region T2
12      } else if (j == 0) {     // Blocks in left edge
13        #pragma omp task depend(in:m[i-1][j],
14                                inout:m[i][j])
15        compute_block(i, j);   // Task region T3
16      } else {                 // Internal blocks
17        #pragma omp task depend(in:m[i-1][j],in:m[i][j-1],
18                                in:m[i-1][j-1],inout:m[i][j])
19        compute_block(i, j); // Task region T4
20      }
21    }}
```

a team of 8 threads. The `master` construct (line 2) specifies that only the *master* thread (i.e. the thread creating the parallel region) 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. Thus, all tasks have an `inout` dependency on the computed block ($[i, j]$) (lines 6, 10, 14 and 18). T_2 and T_3 compute the upper and left edges, so they only consume the left ($[i, j-1]$) and the upper ($[i-1, j]$) blocks respectively (lines 9 and 13), whereas T_4 computes the internal blocks ($[i-1, j]$, $[i, j-1]$ and $[i-1, j-1]$) (lines 17-18). All tasks are guaranteed to have completed at the implicit barrier at the end of the parallel region (line 21).

III. The Static Construction of the TDG

A. Why Deriving the TDG Statically?

Current implementations of OpenMP4 tasking model (e.g. libgomp, nanos++) build the TDG at run-time for a twofold reason: (1) the TDG depends on the tasks that are instantiated (and so executed), which is determined in turn by the control flow graph (CFG); and (2) the addresses of the data elements upon which dependencies are build, are known at run-time.

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 run-time requires storing the hash tables in memory until a `taskwait` or a `barrier` directive is encountered. Since dependencies can only be defined 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 not a problem in HPC systems, in which large amounts of memory are available. However, this is not the case in the newest many-core embedded architectures. For example, the MPPA processor [3] integrates 16 clusters of 16-cores each, with a 2 MB on-chip private memory per cluster. Despite the overall size of the MPPA memory is 32 MB, clusters only have access to their private memory. The rest of memory is accessible through DMA operations (with a significant performance penalization), and so the complete program (including the OpenMP run-time library) must reside within the private memory. Therefore, it is of paramount importance that the memory consumed by the run-time is reduced to the bare minimum.

To minimize the memory used by the run-time, we develop a new compiler pass to statically build the complete TDG and keep it in memory. *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 run-time.

Statically deriving the TDG provides an extra benefit: it allows applying real-time DAG scheduling models[4], from which timing guarantees can be derived [21][15]. This is not addressed in this paper and remains as a future work.

Our compiler pass is composed of two stages: A *control/-data flow analysis stage* and a *task expansion stage*.

B. Control/Data Flow Analysis Stage

This stage identifies: (1) the control flow statements (selections and loops) that determine if a task is instantiated, and (2) the conditions to fulfill for two tasks to be dependent. This is done by the following three compiler analysis phases:

1. Parallel Control Flow Analysis. The *parallel control flow graph* (PCFG) [24] of an OpenMP program is generated, enriching the classic CFG representation with information about parallelism. The compiler may not be able to assert that two `depend` clauses designate the same memory location, e.g. arrays or pointers. Hence, PCFG synchronization edges are augmented with predicates defining the condition to be fulfilled for the edge to exist. In the example shown in Listing 1, the dependencies that matrix m originates among tasks depend on the values of i and j.

2. Induction Variables Analysis. On top of the PCFG, the compiler evaluates the iteration statements to discover the induction variables (IVs) and their evolution over the iterations, i.e. lower bound (lb), upper bound (ub) and stride (str).

3. Range Analysis [19]. Finally, this phase (augmented with support for OpenMP), computes the values of the variables at any point of the program in four steps: (1) generate a set $\mathcal{C}$ of equations that constrains the values of each variable

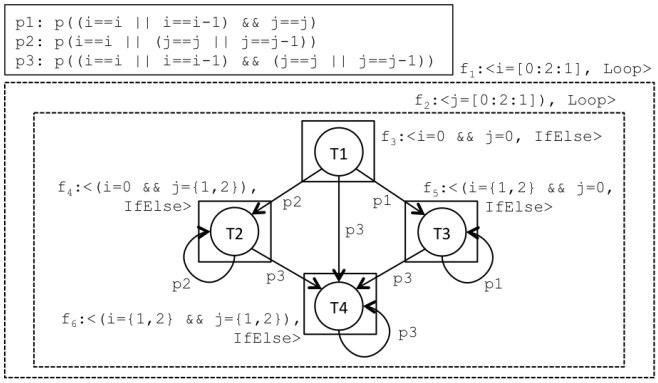


Fig. 1. asTDG of the OpenMP program in Listing 1.

(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* (SCC) to avoid cycles; (4) propagate the ranges over the SCCs in topological order.

These three phases provide the information needed to generate an *augmented static TDG* (asTDG) with data and control flow knowledge. The asTDG is defined by the tuple $\langle N, E, C \rangle$, where: $N = \{V \times T_N\}$ is the set of nodes with its 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; and $C = N \times \{F\}$ 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.

Fig. 1 shows the asTDG of the OpenMP program in Listing 1. It includes the set of task constructs $N = T_1, T_2, T_3, T_4$ from lines 6, 9, 13 and 17, all with type $T_N = Task$. $f_i \in F$ are the control flow statements `for` and `if` at lines 3, 4, 5, 8, 12 and 16, attached to the corresponding tasks in N, and include information about: (1) the IVs of each loop i, j, both with $lb = 0$, $ub = 2$ and $str = 1$ (dashed-line boxes); (2) the conditions of the selection statements enclosing each task (solid-line boxes); and (3) the ranges of the variables in those conditions, e.g. 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 $p1((i == i \,||\, i == i-1)\ \&\&\ j == j)$, evaluates to *true*, meaning that the edge exists when $i = 0, j = 0$ for T_1 and $i = 1, j = 0$ for T_3.

For simplicity, fig. 1 only includes the dependencies that are actually expanded in the next stage (Section C). The actual asTDG has edges between any possible pair of tasks because all they have `inout` dependencies on the element $m[i][j]$.

C. Task Expansion Stage

Based on the asTDG, this stage generates an *expanded static TDG* (esTDG) representing the complete execution of the program in two phases: (1) expand control flow structures (i.e. decide which branches are taken and how many iterations are executed) to determine which tasks are actually instantiated; and (2) resolve the synchronization predicates to conclude which tasks have actual dependencies.

1. Control flow expansion. Control flow structures are expanded from outer to inner levels. In the asTDG in fig. 1, the outer Loop $f1$ is expanded first, then the inner Loop $f2$, and finally the if-else structures $f3, f4, f5$ and $f6$. Each expansion requires the evaluation of the associated expressions to determine the values of each variable. For example, when the outer loop $f1$ is expanded, each iteration is associated with the corresponding values of i.

The expansion process creates two identifiers: (1) a unique *static task construct identifier* (sid_t), labeling each `task` construct, and (2) a identifier of the loops involved in the creation of a task (l_i), labeling each loop expansion step. The latter is crucial for dependency resolution, in order to avoid dependencies from later to previously instantiated tasks (backward dependencies), and both are necessary for generating a task id, t_{id}, used by our lightweight run-time to identify each task instance (see Section IV, Equation 1 for further details).

The control flow expansion results in a temporary TDG in which all tasks instantiated at run-time are defined but synchronization predicates are not solved.

2. Synchronization Predicates Resolution. The value of the variables propagated in the previous stage is used to evaluate predicates and decide which edges actually exist. Likewise, previously generated identifiers l_i are used to eliminate backwards dependencies, e.g. the T_1 instance cannot dependent on any T_2 instance as T_1 comes first according to the control flow.

Fig. 2(a) shows the esTDG of the program in Listing 1. It contains all task instances and all dependencies that could exist at run-time. Each instance contains the task id t_{id} (computed with Equation 1) that allows the run-time to identify the task instances in the esTDG, and the corresponding task construct from which it derives. Transitive dependencies (dashed arrows) are included as well, although they can be removed because they are redundant.

D. Compiler Complexity

In order to define the complexity of the compiler, we analyse the complexity of the two stages.

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 Cyclomatic Complexity [16] metric is usually used; the latter has been proved to have an asymptotic linear complexity [19].

The complexity of the task expansion stage is dominated by the computation of the dependencies among tasks, which is performed using a Cartesian product: The input dependency

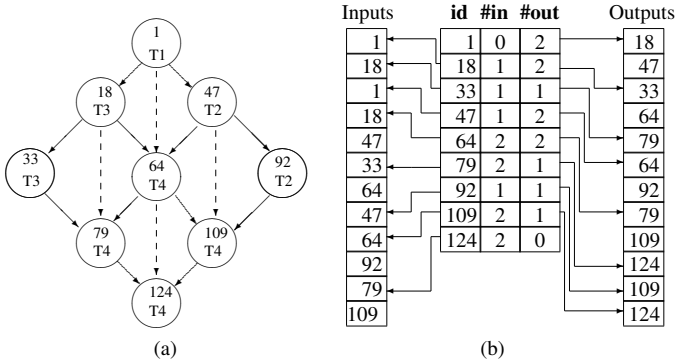


Fig. 2. (a) esTDG of the OpenMP program in Listing 1, (b) Sparse matrix data structure implementing esTDG shown in figure (a).

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.

E. Missing Information When Deriving the esTDG

In case our framework cannot derive some information (mostly when control-flow statements and dependencies contain pointers that may alias or arrays with unresolved subscripts), it still generates an esTDG that correctly represents the execution of the program.

When an if-else statement cannot be evaluated, all its related tasks in C are considered for instantiation. Two situations are equivalent at run-time: (1) a predecessor task never existed because the associated condition evaluates to false, and (2) a predecessor task has already been executed. As a result, it is not wrong to define a dependency between two tasks if one of them eventually does not exist. If a loop cannot be expanded because its boundaries are unknown, we disable parallelism across iterations by inserting a `barrier` at the end of the loop. Lastly, dependencies whose predicate cannot be evaluated are always kept, forcing the involved tasks to be serialized.

The situations described above will result in a bigger esTDG (when if-else conditions cannot be evaluated) or in a performance loss (when loop bounds or synchronization predicates cannot be determined), although guarantee a correct esTDG. In the worst-case scenario, where no information can be derived at compile-time, the resultant esTDG corresponds to the sequential execution of the program, i.e. all tasks are assumed to be instantiated, and all loop iterations and tasks execution are serialized. It is important to remark can embedded applications can often derive all the required information to complete the TDG expansion, as it is required for timing analysis [22].

IV. A Lightweight OpenMP4 Run-Time

Our run-time uses the esTDG to schedule tasks while honoring their dependencies. We have devised an optimal and minimal data structure to store the esTDG: a *sparse matrix*, which reduces considerably the memory consumption of the run-time. Fig. 2(b) shows the sparse matrix implementation of the esTDG presented in fig. 2(a). There, each entry contains an unique task identifier t_{id}, and stores in separate arrays the tasks it depends on (input dependencies), and the tasks depending on it (output dependencies). Moreover, the sparse matrix is sorted using the t_{id}, so a dichotomic search can be applied.

The mechanism used to match the tasks instantiated at run-time with the tasks in the esTDG is the t_{id}. This identifier is computed at both compile-time and run-time, and all the information is ensured to be available in both places. It is based on the sid_t and l_i identifiers (see Section III.C), both introduced by the compiler. In order to obtain the same l_i at compile-time and at run-time, the compiler introduces a *loop stack*, and *push* and *pop* operations are inserted 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 *loop 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. t_{id} is then computed by the compiler and the run-time as follows:

$$t_{id} = sid_t + T \times \sum_{i=1}^{L_t} l_i \cdot M^i \quad (1)$$

Where T is equal to the number of task constructs in the source code plus one, 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 value of the loop unique identifier at nesting level i (computed during expansion at compile-time or with the loop stack at run-time), and M the maximum number of iterations of any considered loop.

It is important to note that a task construct can generate multiple tasks instances, and so the use of loop properties in Equation 1 (L_t, l_i, i and M), guarantees that a unique task identifier for each task instance. Hence, task instances from different loop iterations will result in different t_{id} because every nesting level l_i is multiplied by the maximum number of iterations M.

Consider task T4, with identifier 79, from fig. 2(a). This task instance corresponds to the computation of the matrix block $m[2,1]$. We calculate its identifier as follows: (1) $sid_{T4} = 4$, because T4 is the fourth task in sequential order found while traversing the source code; (2) $T = 5$ because there are 4 task constructs in the source code; (3) $L_{T4} = 2$, the two nested loops enclosing T4; (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: $T4_{id} = 4 + 5(2 * 3^1 + 1 * 3^2) = 79$.

Finally, each task instance 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 or has finished; it is 0 if the task is ready to run; and it is > 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 run-time task scheduler works as follows: when a new task is created, the run-time 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 esTDG contains tasks whose related if-else statement condition has not been determined at compile-time (see Section E) and it evaluates to $false$ at run-time, the value of the counter is the same as the tasks would have already finished, i.e. -1.

TABLE I
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

V. EVALUATION

A. Experimental Setup

OpenMP Framework. Our compiler pass has been developed in Mercurium [7], a source-to-source compiler compatible with OpenMP4. Our lightweight run-time has been developed on top of the GNU libgomp library implementing OpenMP3.1 (included in GCC version 4.7.2), which supports tasks but not dependencies. There is a twofold reason to use a library implementing OpenMP3.1 rather than OpenMP4 [1]: (1) both implementations only differ in the dependency checker (being easier to incorporate a new one, rather than replacing it); (2) the MPPA processor only supports OpenMP3.1. We also consider the libgomp library included in GCC 4.9.2, which implements OpenMP4, for comparison purposes.

Applications. From the HPC domain, we consider a *cholesky factorization* [5], useful for efficient linear equation solvers and Monte Carlo simulations. Cholesky can also be used to accelerate *Kalman filter*, implemented in autonomous vehicle navigation systems to detect pedestrians and bicyclists positions [14]. From the embedded domain, we consider an application resembling the *3D path planning* [8] (r3DPP), used for airborne collision avoidance. For comparison purposes, the applications have been parallelized with OpenMP3.1 too, using `task` and `taskwait` directives.

Platform Setups. Our experiments consider two different processor setups: (1) two Intel Xeon CPU E5-2670 processors, featuring 8 cores each, with 20 MB L3; and (2) the MPPA processor [3] featuring 256 cores organized in 16 clusters of 16 cores each, and 2 MB of private on-chip memory per cluster. The former executes a complete Linux system, in which OpenMP3.1 and OpenMP4.0 are supported; the latter, only supports OpenMP3.1. The Intel Xeon processor is used only for comparision purposes, as there is no OpenMP4 runtime implemented for the MPPA.

B. Performance Speed-up and Memory Usage

Fig. 3(a) and fig. 4(a) show the performance speed-up achieved by Cholesky and r3DPP respectively in the Intel Xeon processor, when varying the number of instantiated tasks, ranging from 1 to 5984 and 4096 respectively, and considering the three libgomp run-times: OpenMP4, OpenMP3.1 and OpenMP3.1 augmented with our dependency checker (labeled as *omp4*, *omp 3.1* and *lightweight omp4* respectively). The performance has been computed with the average of 100 executions. Similarly, fig. 3(b) and fig. 4(b) show the heap memory usage (in KB) of the three OpenMP run-times when executing Cholesky and r3DPP respectively in the Intel Xeon processor and varying the number of instantiated tasks as well. The memory usage has been extracted using *Valgrind Massif* [17] tool, which allows profiling the heap memory consumed by the run-time in which the TDG structure is maintained.

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 run-time and the small workload of each task, slows-down the performance.

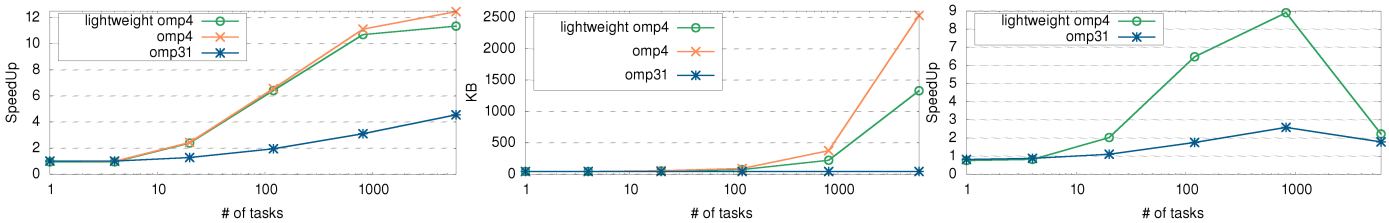
Our *lightweight omp4* obtains the same performance speed-ups as the *omp4* implementation, and outperforms *omp 3.1*. However, in case of *omp4*, the memory usage rapidly increases, requiring much more memory than our run-time.

The parallelization opportunities brought by the `depend` clause make the performance of Cholesky (fig. 3(a)) to increase significantly compared to the OpenMP 3.1 model, with a speed-up increment from 4x to 12x when instantiating 5984 tasks. At this point, *omp4* consumes 2.5MB while our *lightweight omp4* requires less than 1.3MB. The memory consumed by *omp3.1* is less than 100KB (fig. 3(b)). In fact, the *omp3.1* memory consumption is similar for all the applications because no structure for dependencies management is needed.

For the r3DPP (fig. 4(a)), the tasking model achieves a performance speed-up of 5.2x and 5.8x with *omp4* and *lightweight omp4* respectively, when instantiating 1024 tasks. At this point, *omp4* consumes 400 KB in front of the 200 KB consumed by *lightweight omp4* (fig. 4(b)). *omp31* achieves a maximum performance of 4.5x when 256 tasks are instantiated. When the number of task instances increases to 4096, all run-times suffer a significant performance degradation because the number of instantiated tasks is too high compared to the workload computed by each task. The *lightweight omp4* suffers a higher performance penalization due to the dichotomic search.

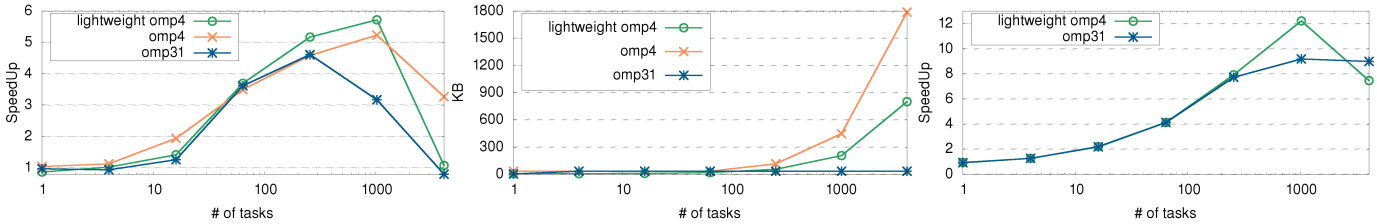
Table I shows the size of the sparse matrix data structure implementing the esTDG of each application when varying the number of instantiated tasks (the memory consumption reported in figs. 3(b) and 4(b) already includes it).

[1]To ensure that results are not affected by the version of the library, we executed the applications considered in this paper 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.



(a) Intel Xeon speed-up (b) Memory usage (c) MPPA speed-up

Fig. 3. Performance speed-up and memory usage (in KB) of the Cholesky running with *lightweight omp4, omp4* and *omp 3.1*, and varying the number of tasks.



(a) Intel Xeon speedup (b) Memory usage (c) MPPA speedup

Fig. 4. Performance speed-up and memory usage (in KB) of the r3DPP running with *lightweight omp4, omp4* and *omp 3.1*, and varying the number of tasks.

To evaluate the benefit of OpenMP4 on a memory constrained many-core architecture, we evaluated our lightweight run-time on the MPPA processor. Figs. 3(c) and 4(c) show the performance speed-up of Cholesky and r3DPP executed in one MPPA cluster, considering the *lightweight omp4* and *omp31* run-times and varying the number of tasks. Note that *omp4* run-time experiments are not provided because MPPA does not support it. Memory consumption is the same as the one shown in fig. 3(b) and 4(b). r3DPP increases the performance speed-up from 9x to 12x when using our *lightweight omp4* rather than *omp3.1* and only consuming 200 KB. Cholesky presents a significant speed-up increment when instantiating 816 tasks, i.e. from 2.5x to 9x, consuming only 220 KB.

C. Impact of Missing Information When Deriving the esTDG

The impact of missing information when deriving the esTDG may vary depending on the amount of unknown data. For example, in Listing 1, not determining the condition in line 5 (f_3), results in a bigger esTDG than not determining the condition in line 16 (f_6). The reason is because the task protected with f_3 (T_1) is instantiated only once, while the task protected with f_6 (T_4) is instantiated four times (see fig. 2(a)), and so considering that T_1 is always instantiated has a higher impact than considering T_4.

Despite our compiler-pass has been able to derive the complete esTDG of the two applications, in order to properly measure the impact of missing information in terms of esTDG size we evaluate the case in which the compiler cannot obtain all information from r3DPP. First, we identify those if-else statements with the highest and lowest impact (in each scenario the compiler is unable to determine 25% of the conditions). Assuming that 1024 tasks are instantiated (peak performance), the scenario with the highest impact increases the esTDG by 126.67% (71.97 KBs); the scenario with the lowest impact increases the esTDG by only 7.77% (34.22 KBs). Second, we assume the loop bounds cannot be determined. In this case, the execution becomes sequential, as most of the parallelism of the application comes from executing multiple iterations in parallel (similar with Cholesky). It is important to remark that, in real-time it is mandatory to determine the loop bounds in order to derive timing guarantees [22].

VI. Related work

Sarkar et al.[26] presented a framework for partitioning and scheduling tasks at compile-time balancing tasks granularity, and thus overhead and parallelism. Vijaykumar et al.[25] proposed a set of heuristics to generate a TDG for massively data-parallel applications based on the CFG and use-definition analysis, aiming at reducing communication and synchronization overheads. Yet none of these methodologies are able to create at compile-time a TDG for complex and irregular algorithms.

Pugh et al.[11] proposed a new technique to detect dependencies at compile-time and map HPC kernels into cluster nodes. The disadvantage is that it is expensive and introduces overhead to the compiler while, in our proposal, much simpler dependency analysis are enough to check tasks parallelism.

Tzenakis [12] et al. implemented task instantiation, dependence analysis and scheduling techniques and proved their efficiency over other run-times such as SMPSs [20]. However, this method has run-time overhead and requires heavy data-structures for dynamic dependency checking. Finally, some hybrid approaches [23] have been presented to try to get the best of both static and dynamic methods, but it still introduces

too much overhead, as they recognize.

OpenMP run-times for multi-core platforms with limited memory resources have been proposed [13] for OpenMP2.5 in which the tasking model is not supported.

VII. Conclusions

This paper presents a novel framework to allow executing OpenMP4 programs in many-core embedded processors relying on small on-chip memories. Our framework is composed of: (1) a new compiler pass capable of generating an *expanded static TDG* (esTDG) of OpenMP programs, containing all tasks and all dependencies that can possibly exist, and (2) a lightweight memory efficient run-time library based on the esTDG. Our tests reveal that, compared to current OpenMP4 implementations, our run-time significantly reduces the memory consumption, providing a similar speed-up. Moreover, we empower the MPPA processor to execute OpenMP4 programs.

Acknowledgments

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

References

[1] GCC-GNU libgomp. `http://goo.gl/46514p`.

[2] OpenMP API, Version 4.0. Technical report, OpenMP ARB, July 2013.

[3] B. D. de Dinechin, D. van Amstel, M. Poulhies, G. Lager. Time-critical computing on a single-chip massively parallel processor. In *DATE*, 2014.

[4] S. Baruah. Improved multiprocessor global schedulability analysis of sporadic DAG task systems. In *ECRTS*, 2014.

[5] N. Bascelija. Sequential and parallel algorithms for cholesky factorization of sparse matrices. *WSEAS: Mathematical Applications in Science and Mechanics*, 2013.

[6] L. Benini, E. Flamand, D. Fuin, and D. Melpignano. P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator. In *DATE*, 2012.

[7] BSC. OmpSs (Mercurium compiler and Nanos RTL). `pm.bsc.es`.

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

[9] C. Wang, S. Chandrasekaran, B. Chapman, J. Holt. libEOMP: a portable OpenMP runtime library based on MCA APIs for embedded systems. In *PMAM*, 2013.

[10] E. Stotzer, A. Jayraj, M. Ali, A. Friedmann, G. Mitra, et al. OpenMP on the Low-Power TI Keystone II ARM/DSP SoC. In *IWOMP*. 2013.

[11] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, J. Dongarra. Dague: A generic distributed dag engine for high performance computing. *Parallel Computing*, 38(1-2):37–51, 2012.

[12] G. Tzenakis, A. Papatriantafyllou, H. Vandierendonck, P. Pratikakis,et al. Bddt: block-level dynamic dependence analysis for deterministic task-based parallelism. *SIGPLAN Not.*, 47(8), 2012.

[13] L. Tao, Z. Ji, Q. Wang. Research on OpenMP algorithms on memory limited embedded multicore platform. In *Journal of Computational Information Systems*, 2010.

[14] J. Levinson, J. Askeland, J. Becker, and J. Dolson. Towards fully autonomous driving: Systems and algorithms. *Intelligent Vehicles Symposium (IV), 2011 IEEE*, 2011.

[15] Maria A. Serrano Alessandra Melani, Roberto Vargas, Andrea Marongiu, Marko Bertogna, Eduardo Quinones. Timing characterization of openmp4 tasking model. In *International Conference on Compilers, Architectures and Synthesis of Embedded Systems (CASES)*, 2015.

[16] T. McCabe. A complexity measure. *IEEE Transactions of Software Engineering*, 1976.

[17] N. Nethercote, R. Walsh, J. Fitzhardinge. Building workload characterization tools with valgrind. In *IISWC*, 2006.

[18] P. Burgio, G. Tagliavini, A. Marongiu, L. Benini. Enabling fine-grained OpenMP tasking on tightly-coupled shared memory clusters. In *DATE*, 2013.

[19] R. E. Rodrigues, V. H. Sperle Campos, F. M. Quinto Pereira. A fast and low-overhead technique to secure programs against integer overflows. In *CGO*, 2013.

[20] R. M. Badia, J. R. Herrero, J. Labarta, J. M. Perez, E. S. Quintana-Orti, et al. Parallelizing dense and banded linear algebra libraries using SMPSs. *Concurrency and Computation: Practice and Experience*, 21(18), 2009.

[21] R. Vargas, E. Quinones, A. Marongiu. OpenMP and Timing Predictability: A Possible Union? In *DATE*, 2015.

[22] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, et al. The worst-case execution-time problem–overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 2008.

[23] S. Arandi, G. Michael, P. Evripidou, C. Kyriacou. Combining compile and run-time dependency resolution in data-driven multithreading. In *DFM Workshop*, 2011.

[24] S. Royuela, R. Ferrer, D. Caballero, X. Martorell. Compiler analysis for OpenMP tasks correctness. In *CF*, 2015.

[25] T. N. Vijaykumar, G.S. Sohi. Task selection for a multiscalar processor. In *ACM/IEEE International Symposium on Microarchitecture*, 1998.

[26] V. Sarkar, J. Hennessy. Compile-time partitioning and scheduling of parallel programs. In *Symposium on Compiler Construction*, 1986.