Tareador: The Unbearable Lightness of Exploring Parallelism

Vladimir Subotic, Arturo Campos, Alejandro Velasco, Eduard Ayguade, Jesus Labarta, and Mateo Valero

Abstract The appearance of multi/many-core processors created a gap between the parallel hardware and sequential software. Furthermore, this gap keeps increasing, since the community cannot find an appealing solution for parallelizing applications. We propose Tareador as a mean for fighting this problem.

Tareador is a tool that helps a programmer explore various parallelization strategies and find the one that exposes the highest potential parallelism. Tareador dynamically instruments a sequential application, automatically detects data-dependencies between sections of execution, and evaluates the potential parallelism of different parallelization strategies. Furthermore, Tareador includes the automatic search mechanism that explores parallelization strategies and leads to the optimal one. Finally, we blueprint how Tareador could be used together with the parallel programming model and the parallelization workflow in order to facilitate parallelization of applications.

1 Introduction

Parallel programming became an urge, an urge that the programmers community fails to efficiently respond to. One of the biggest problems in the current computing industry is the steady-growing gap between the increasing parallelism offered by state-of-the-art architectures and the limited parallelism exposed in state-of-the-art applications. Consequently, software parallelism has become concern of every single programmer. However, parallelizing applications is far from trivial.

The community keeps inventing novel programming models as enablers for transition to parallel software. These novel programming models come in various flavours, offering different programming paradigms, levels of abstraction, etc. However, most of the novel programming models fail to get widely adopted. It takes a giant leap of faith for a programmer to take the already working application and to port it to a novel programming model. This is especially problematic because the

V. Subotic (✉) • A. Campos • A. Velasco • E. Ayguade • J. Labarta • M. Valero
Barcelona Supercomputing Center, Barcelona, Spain
e-mail: vladimir.subotic@bsc.es; arturo.sanemeterio@bsc.es; alejandro.velasco@bsc.es; eduard.ayguade@bsc.es; jesus.labarta@bsc.es; mateo@ac.upc.edu

© Springer International Publishing Switzerland 2015
C. Niethammer et al. (eds.), *Tools for High Performance Computing 2014*,
DOI 10.1007/978-3-319-16012-2_4

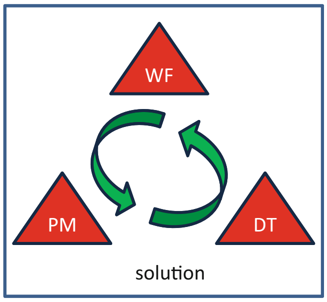


Fig. 1 There is a need to make the complete parallelization solution that will contain not only the development tool (*DT*), but also the proposal of the programming model (*PM*) and the description of workflow (*WF*) that guides the process of parallelizing applications

programmer cannot anticipate how would the application perform within the new programming model and thus whether the porting is worth the effort. Moreover, the programmer usually lacks development tools and a clear idea how the parallelization process should be conducted.

It is our belief that the programmers should be offered not only with the programming model, but with the whole parallelization solution – a solution that includes parallel programming model, parallelization development tool, and the parallelization workflow that describes how to use the development tool to port the sequential application to the selected parallel programming model. We believe that the three parts of the solution need to be tailored to work together in the bigger system (Fig. 1). The development tool should instrument the sequential application and provide to the user the information that is relevant in the context of the target parallel programming model. Finally, the parallelization workflow should glue the tool and the programming model and define the process of parallelizing applications.

This paper attempts to paint the big picture of parallelization solution, putting the special emphasize on the part of the parallelization development tool. More specifically, this paper contributes in the following two directions:

- We present Tareador – a tool for assisted parallelization of sequential applications. Tareador allows the programmer to understand the inner workings of the application, identify the dependencies between different parts of the execution and evaluate the parallelism inherent in the code. We describe Tareador at its current development phase, as a tool to explore potential parallelization strategies. Furthermore, we also discuss the planed future development of Tareador in an effort to make it a complete tool for assisted parallelization of applications.
- We describe the parallelization solution that includes Tareador. We propose a parallel programming model that best suits the Tareador obtained information, and describe the workflow that uses Tareador to parallelize application by porting it to the target programming model.

The rest of the paper is organized as follows. Section 2 illustrates the problem of finding the optimal parallelization strategy. Section 3 describes the implementation and usage of Tareador environment. Furthermore, in Sect. 4 we describe the automatic algorithm that automatically drives Tareador in exploration of good parallelization strategies. We describe the heuristics and metrics that guide the automatic search and show the results of automatic search in the case of couple of well-known applications. Furthermore, we include a broad discussion of how we see Tareador being a part of the whole environment for easy parallelization of applications (Sect. 5). We declare our selection of the parallel programming model and devise a custom parallelization workflow that would facilitate parallelization of applications. Finally, we conclude the paper with the related work on the topic (Sect. 6) and the conclusions of our study (Sect. 7).

2 Motivating Example

Parallelization of a sequential application consists of decomposing the code into tasks (e.g. units of parallelism) and implementing synchronization rules between the created tasks. However, even if the sequential application is simple, finding the optimal task decomposition can be a difficult job. The application may exhibit parallelism that is very distant and irregular, parallelism among sections of code that are mutually far from each other. This type of parallelism is very hard for the programmer to identify and expose without any development support. Thus, to find the optimal parallelization strategy, the programmer must know the source code in depth in order to identify all the data dependencies among tasks. Furthermore, the programmer must anticipate how will all the tasks execute in parallel, and what is the possible parallelism that these tasks can achieve.

Figure 2 shows a simple sequential application composed of four computational parts, the data dependencies among those parts, and some of the possible taskification strategies. Although the application is very simple, it allows various decompositions that expose different amount of parallelism. $T0$ puts all the code in one task and, in fact, presents a sequential code. $T1$ and $T2$ both break the application into two tasks but fail to expose any parallelism. On the other hand, $T3$ and $T4$ both break the application into 3 tasks, but while $T3$ achieves no parallelism, $T4$ exposes concurrency between C and D. Finally, $T5$ breaks the application into 4 tasks but achieves the same amount of parallelism as $T4$. Considering that increasing the number of tasks increases the runtime overhead, one can conclude

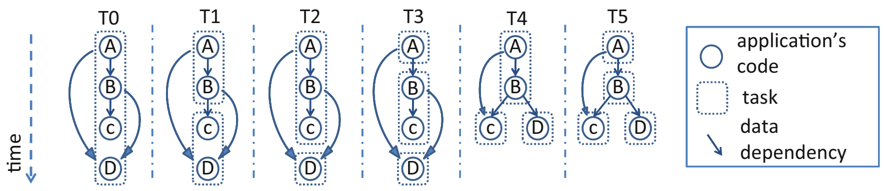


Fig. 2 Execution of different possible taskifications for a code composed of four parts

that the optimal taskification is $T4$, because it gives the highest speedup with the lowest cost of the increased number of tasks.

Nevertheless, compared to the presented trivial execution, a real-world application would be more complex in various aspects. A real application may have hundreds of thousands of task instances, causing complex and well populated dependency graphs. The large dependency graph would allow unpredictable scheduling decisions that would potentially exploit distant parallelism. Also, with the task instances of different duration, evaluating the potential parallelism would be even harder. Due to all this complexity, it is unfeasible for a programmer alone to do the described analysis and estimate the potential parallelism of a certain task decomposition. Therefore, we believe that it would be very useful to have an environment that quickly anticipates the potential parallelism of a particular taskification. We describe such a framework in the following section.

3 Tareador Environment

Tareador allows the programmer to start from a sequential application, propose some decomposition of the sequential code into tasks and get fast estimation of the potential parallelism. The input to Tareador is a sequential code. Tareador compiler marks all logical sections of code as potential tasks. In addition, the user can manually annotate other potential task. The annotated code is executed sequentially – all annotated tasks are executed in the order of their instantiation. Tareador dynamically instruments the sequential execution and collects the log of memory usage of each potential task. Once the logs are generated, Tareador allows the programmer to select one task decomposition of the sequential code. For the selected decomposition, Tareador calculates inter-task dependencies and evaluates the potential parallelism of the decomposition providing to the user the results in the form of:

- Simulation of the potential parallel execution;
- Dependency graph of all task instances;
- Visualization of the memory usage of each task.

Tareador environment integrates various internally and externally developed tools. The framework (Fig. 3) takes the **input code** and compiles it with LLVM-based [1] **Tareador compiler**. The execution of the obtained binary generates Tareador **execution logs**. Further post-mortem processing of the execution logs is encapsulated into **Tareador GUI**. Tareador GUI allows the user to select one decomposition. Based on the selected task decomposition, **Tareador backend** consumes the execution logs to calculate final results. More specifically, Tareador backend generates execution trace that **Dimemas** [2] simulates to obtain **Paraver** [3] time-plots of the potential parallel execution. Also, Tareador backend produces the task dependency graph that can be visualized with **Graphviz** [4], as well as dataview information that can be visualized internally by Tareador GUI.

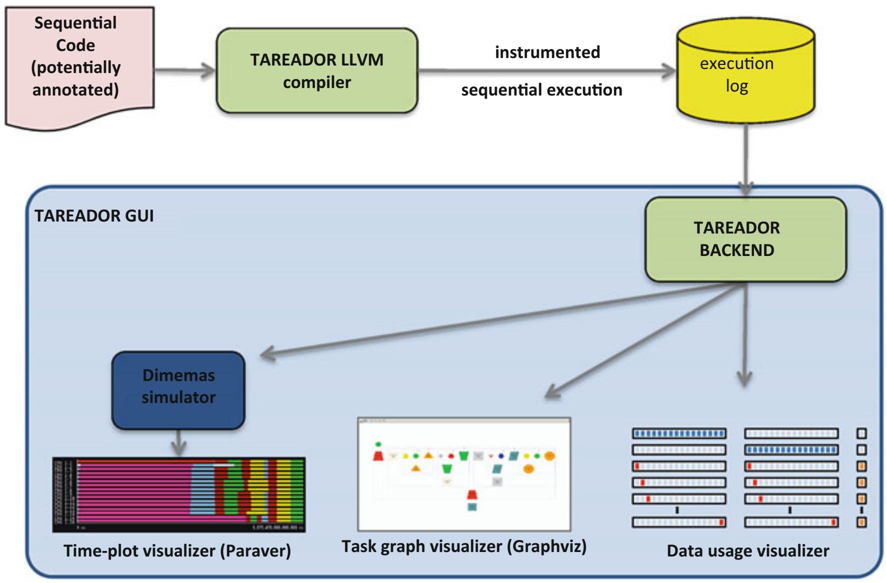


Fig. 3 Tareador framework

3.1 Implementation Details

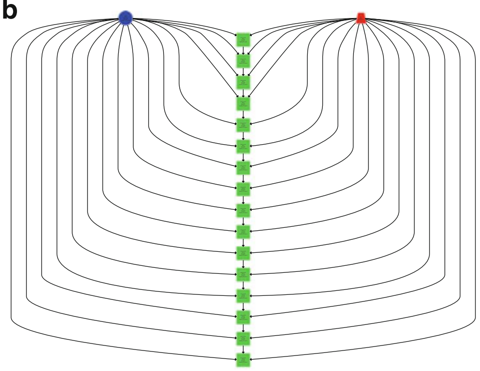
Tareador uses LLVM framework to dynamically instrument the sequential application and collect the log of all potential tasks and their memory usage. Tareador compiler injects to the original sequential execution instrumentation callbacks that collect the data needed for Tareador analysis. First, Tareador compiler must mark all the potential tasks in the execution. The compiler marks as a potential task every logical code section that can take a significant amount of time – each function, loop or loop iteration. Also, the compiler allows the user to manually annotate any potential task by wrapping an arbitrary code sections using Tareador API (example in Sect. 3.2). Furthermore, Tareador intercepts and processes each memory access of the sequential execution. Finally, based on the dynamically collected information, Tareador flushes the execution log that contains all intercepted potential tasks and their memory usage. The resulting log is indexed to allow fast post-mortem browsing.

Tareador GUI allows the user to easily browse different task decompositions of the instrumented execution. Given the configuration of one task decomposition, Tareador backend consumes the execution logs to evaluate the parallelism of the decomposition. The backend finds all the specified tasks in the execution log, calculates data-dependencies between them and prepares outputs for different visualization tools. Finally, GUI allows the user to see all the obtained results and select how to refine the decomposition to achieve higher parallelism.

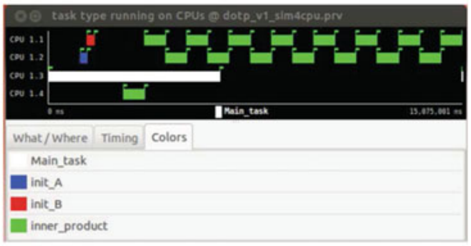
a

```
01    int void dot_product (long N, double A[N],
02                                 double B[N], double *acc) {
03      double prod;
04      *acc=0.0;
05      for (int i=0; i<N; i++) {
06        tareador_start_task("inner_product");
07        prod = A[i]*B[i];
08        //tareador_disable_object(acc);
09        *acc+= prod;
10        //tareador_enable_object(acc);
11        tareador_end_task("inner_product");
12      }
13    }
14
15    int main() {
16      tareador_ON ();
17      tareador_start_task("init_A");
18      for (int i=0; i< N; i++) A[i]=i;
19      tareador_end_task("init_A");
20
21      tareador_start_task("init_B");
22      for (int i=0; i< N; i++) B[i]=2*i;
23      tareador_end_task("init_B");
24
25      dot_product (N, A, B, &result);
26      tareador_OFF ();
27    }
```

b



c



task type running on CPUs @ dotp_v1_sim4cpu.prv

CPU 1.1
CPU 1.2
CPU 1.3
CPU 1.4

0 ns   Main_task   15,075,001 ns

What / Where | Timing | Colors

- Main_task
- 🟦 init_A
- 🟥 init_B
- 🟩 inner_product

d

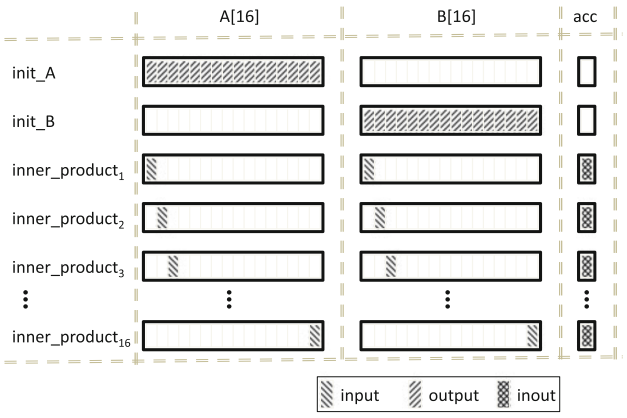


Fig. 4 Applying Tareador on dot product kernel. (**a**) Source code. (**b**) Task dependency graph. (**c**) Potential parallel execution (4 cores). (**d**) Visualization of memory accesses

3.2 Illustration of Tareador Usage

This section illustrates the usage of Tareador by applying it on a simple code of dot product computation (Fig. 4). Figure 4a shows the original sequential code (code in black). The code initializes two buffers operands in two loops, and then computes the result in function *dot_product*. In order to prepare the execution for Tareador instrumentation, the user adds the gray code lines. To mark which code section will be instrumented by Tareador, the user inserts functions *tareador_ON* and *tareador_OFF* (code lines 16 and 26). Furthermore, to propose one task decomposition, the user inserts calls *tareador_start_task* and *tareador_end_task*. Additional strings passed to these functions mark the name of the task that is encapsulated by the matching calls. In the presented examples, the selected task decomposition splits the sequential execution into 2 initializing tasks (*init_A* and *init_B*) and 16 computational tasks (*inner_product*), one for each iteration of the loop in *dot_product* function. It is important to note that Tareador can work without these user annotations that mark the decomposition. Tareador compiler can automatically mark all the potential tasks and then the different task-decomposition could be browsed through Tareador GUI, without the need to modify the target sequential code.

For the selected task decomposition, Tareador automatically evaluates the potential parallelism. First output that evaluates parallelism is a tasks dependency graph (Fig. 4b). The tasks dependency graph is a directed cyclic graph where each node represents a task instance, while each edge represents a data-dependency between two task instances. In the presented example, the blue and red nodes represent tasks *init_A* and *init_B*, while the green nodes represent instances of *inner_product*. The graph shows that each instance of *inner_product* depends on *init_A*, *init_B* and the previous instance of *inner_product* (if any). The second Tareador output is the time-plot of the potential parallel execution of the selected decomposition (Fig. 4c). The figure shows, for each of the 4 cores in the parallel machine (y-axis), which task executes in any moment of time (x-axis). The colors representing task types match the colors from the dependency graph. The presented plot confirms that green task instances (*inner_product*) are serialized.

Tareador's dataview visualization can further pinpoint the memory objects that impede parallelism. Figure 4d shows for all task instances the memory access patterns within the objects of interest. As expected, the initialization tasks (*init_A* and *init_B*) access only their target arrays. On the other hand, each instance of *inner_product* reads one element from both arrays A and B and increments *acc* (inout access stands for both input and output). Therefore, dependencies between instances of *inner_product* are caused by the memory object *acc*.

Going back to the source code for from Fig. 4a, we can recognize the dependency on the object *acc* as an apparent case that can be avoided using reduction. Thus, Tareador allows the user to evaluate the potential parallelism if the dependency on *acc* is to be avoided using reduction. The user can uncomment the code lines 8 and 10 and declare that, within the encapsulated code snippet, the memory accesses to *acc* should be omitted. In other words, the user instructs Tareador to ignore the dependency on the object *acc*. Consequently, the resulting decomposition allows concurrency between instances of *inner_product*, as shown in Fig. 5.

For the illustration purpose, in this section we decided to describe the usage of the **lite** mode of Tareador. The lite mode requires the user to manually mark the task decomposition of the code. Every time the user specifies the decomposition,

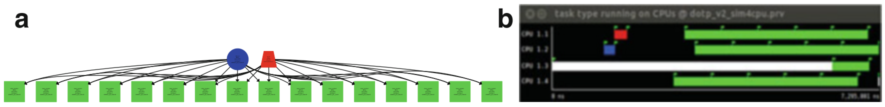


Fig. 5 Applying Tareador on dot product kernel (disabled accesses to *acc*). (**a**) Task dependency graph. (**b**) Potential parallel execution (4 cores)

Tareador outputs the described results. Conversely, the **original** Tareador mode requires no annotations of the target sequential code. In the original mode, Tareador compiler automatically marks all the potential tasks in the sequential code, and lets the user browse all the potential decompositions through Tareador GUI. For the purpose of parallelizing applications, original mode is much more efficient than then lite mode. However, for demonstrative/teaching purposes, the lite mode is preferred. This is because the lite mode makes the students be more involved with the actual target code. Every time the student wants to change the decomposition, she is forced to interact with the target code and understand better the sources of parallelism. The lite mode of Tareador has been successfully introduced into the teaching curriculum of parallel programming courses at the Technical University of Catalonia.

4 Automatic Exploration of Parallelism with Tareador

In our prior work [5] we demonstrated how a programmer can use Tareador to iteratively explore the task decomposition space and find the decomposition that exposes sufficient parallelism to efficiently deploy multi-core processors. However, the presented process relied strongly on programmer's experience to guide the search. To further facilitate the process of finding optimal parallelization strategy, our next goal is to formalize the programmers' experience into an autonomous algorithm for automatic search of potential parallelization strategies. The rest of this section describes the autonomous algorithm and metrics and heuristics that define it.

The automatic exploration of parallelization strategies is based on: evaluating parallelism of various decompositions; collecting key parameters that identify the parallelization bottlenecks; and refining decompositions in order to increase parallelism. The search algorithm is illustrated in Fig. 6. The inputs of this algorithm are the original unmodified sequential code and the number of cores in the target platform. The search algorithm passes through the following steps:

1. Start from the most coarse-grain task decomposition, i.e. the one that considers the whole main function as a single task.
2. Perform an estimation of the potential parallelism of the current task decomposition (the speedup with respect to the sequential execution).

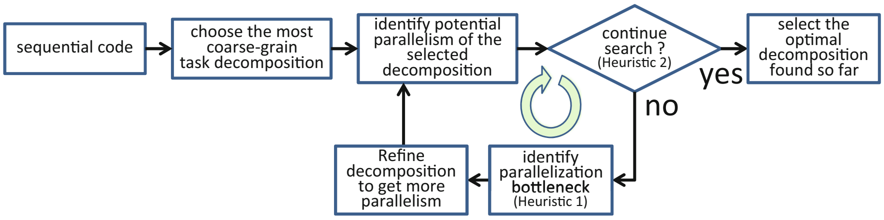

sequential code
choose the most coarse-grain task decomposition
identify potential parallelism of the selected decomposition
continue search ? (Heuristic 2)
yes
no
select the optimal decomposition found so far
identify parallelization bottleneck (Heuristic 1)
Refine decomposition to get more parallelism

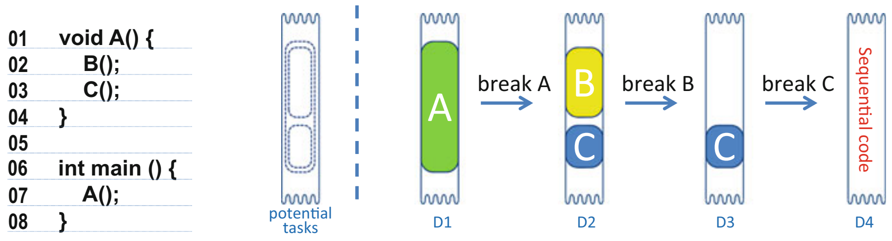
Fig. 6 Algorithm for exploring parallelization strategies

3. If the exit condition is met (*Heuristic 2*), finish the search.
4. Else, identify the parallelization bottleneck (*Heuristic 1*), i.e. the task that should be decomposed into finer-grain tasks.
5. Refine the current task decomposition in order to avoid the identified bottleneck. Go to step 2.

4.1 Algorithm Heuristics

In the following sections, we further describe the design choices made in designing the mentioned heuristics. Nevertheless, first we must define more precise terminology. Primarily, we must make a clear distinction between a **task type** (function that is encapsulated into task) and a **task instance** (dynamic instance of that function). For instance, if function *compute* is encapsulated into a task, we will say that *compute* is a **task type**, or just a **task**. Conversely, each instantiation of *compute* we will call a **task instance**, or just an **instance**. A task instance is atomic and sequential, but various instances (of same or different task type) can execute concurrently among themselves.

Also, we will often use a term **breaking a task** to refer to the process of transforming one task into more fine-grain tasks. For example, Fig. 7 illustrates decomposition refining in a case of a simple code. The process starts with the most coarse-grain decomposition ($D1$) in which function A is the only task. By breaking task A, we obtain decomposition $D2$ in which A is not a task and instead its direct children (B and C) become tasks. If in the next step we break task B, assuming that B contains no children tasks, B will be serialized (i.e. B is not a task anymore and its computation becomes a part of the sequential execution). Similarly, the next refinement serializes task C and leads to the starting sequential code. At this point, no further refinement is possible, so the iterative process naturally stops.



```
void A() {
    B();
    C();
}

int main () {
    A();
}
```

Fig. 7 Iterative refinement of decompositions

4.1.1 Heuristic 1: Which Task to Break

In the manual search for an efficient decomposition, the programmer decides which task is the parallelization bottleneck. The practice shows that the bottleneck task is often one of the following:

1. **The task whose instances have long duration**, because a long instance may cause significant load imbalance.
2. **The task whose instances have many dependencies**, because an instance with many dependencies may be a strong synchronization point.
3. **The task whose instances have low concurrency**, because an instance with low concurrency may prevent other instances to execute in parallel.

Our goal is to formalize this programmer experience into a simple set of metrics that can lead an autonomous algorithm for exploring potential task decompositions. The goal is to define a **cost function** for task type i as:

$$\overline{t_i} = \overline{l_i(p_l)} + \overline{d_i(p_d)} + \overline{c_i(p_c)} \tag{1}$$

where l_i, d_i and c_i are functions that calculate the partial costs related to tasks' length, dependencies count and concurrency level. On the other hand, parameters p_l, p_d and p_c are empirically identified parameters that tune the weight of each partial cost within the overall cost. The following paragraphs further describe the operands from Eq. 1.

Metric 1: Task Length Cost

A task type that has long instances is a potential parallelization bottleneck. Thus, based on the length of instances, we define a metric called length cost of a task type. Length cost of some task type is proportional to the length of the longest instance of that task. Therefore, if task i has instances whose lengths are in the array T_i, the length cost of task i is:

$$l_i = \max(t), t \in T_i \tag{2}$$

Furthermore, we define a normalized length cost of task i as:

$$\overline{l_i(p)} = \frac{(l_i)^p}{\sum_{j=1}^{N}(l_j)^p}, \quad 0 \le p < \infty \tag{3}$$

where the control parameter p is used to tune the distribution of normalized costs (explained later in this section).

Metric 2: Task Dependency Cost

A task type that causes many dependencies is another potential parallelization bottleneck. Thus, based on the number of dependencies (sum of incoming and outgoing dependencies), we define a metric called dependency cost of a task type. Dependency cost of some task is proportional to the maximal number of dependencies caused by some instance of that task. Therefore, if task i has instances whose numbers of dependencies are in the array D_i, the dependencies cost of task i is:

$$d_i = \max(z), z \in D_i \tag{4}$$

Furthermore, using a control parameter p, we define the normalized dependency cost of task i as:

$$\overline{d_i(p)} = \frac{(d_i)^p}{\sum_{j=1}^{N}(d_j)^p}, \quad 0 \le p \le \infty \tag{5}$$

Metric 3: Task Concurrency Cost

A task type that has low concurrency is another potential parallelization bottleneck. Concurrency of some instance is determined by the overall utilization of the machine during the execution of that instance. Thus, we define concurrency cost of some task to be inversely proportional to the average number of cores that are efficiently utilized during the execution of that task. Therefore, if task i has task instances which run for time $T_{i,j}$ while there are j cores efficiently utilized, the concurrency cost of task i is:

$$c_i = \frac{\sum_{j=1}^{cores}\frac{T_{i,j}}{j}}{\sum_{j=1}^{cores}T_{i,j}} \tag{6}$$

Again, using a control parameter p, we define the normalized concurrency cost of task i as:

$$\overline{c_i(p)} = \frac{(c_i)^p}{\sum_{j=1}^{N}(c_j)^p}, \quad 0 \le p \le \infty \tag{7}$$

Control Parameter p

Introduction of the parameter p provides the mechanism for controlling the mutual distance of the normalized costs for different tasks. For instance, let us assume that the application consists of two task instances, A and B, where A is two times longer than B. If the control parameter p_l is equal to 1, the normalized length costs for tasks A and B are 0.67 and 0.33, respectively. However, if the control parameter p_l is equal to 2, the costs for tasks A and B become 0.8 and 0.2, respectively.

Therefore, by changing parameter p of some metric, we can control the impact of that metric on the overall cost. For example, if the control parameter for length cost is 0, all task types will have the same normalized length cost, independent of the length of task instances. Thus, the length of tasks would have no impact on the overall cost. On the other hand, if the control parameter for length cost is infinite, the task type with the longest instance will have the normalized length cost of 1, while all other task types will have the normalized length cost of 0. This way, the impact of the task length on the overall cost would be maximized.

4.1.2 Heuristic 2: When to Stop Refining the Decomposition

The algorithm also needs a condition to stop the iterative search. Iterative search leads to fine grain decompositions that instantiate a very high number of tasks. An excessive number of tasks causes a very complex and computation intensive evaluation of the potential parallelism. Thus, to make the complete automatic search viable, we must adopt the exit condition that will prevent processing unnecessary decompositions.

To construct the Heuristic 2, we must create a system for rating the quality of a decomposition. Our basic rating system consists of two rules. First, out of all tested decompositions, the optimal decomposition is the one that achieves the highest parallelism. Second, if the optimal decomposition achieves the parallelism of s_{opt} and instantiates t_{opt} tasks, and some other decomposition i achieves the parallelism of s_i and instantiates t_i tasks, the relative quality of decomposition i compared to the optimal decomposition is:

$$Quality_i = \left(\frac{s_i}{s_{opt}}\right) \cdot \left(\frac{t_{opt}}{t_i}\right)^{exp\_tasks}, \quad 0 \le exp\_tasks \le 1 \tag{8}$$

Thus, the relative quality of some decomposition drops as the achieved parallelism drops and as the number of instantiated tasks increases. Furthermore, the parameter *exp_tasks* serves to tune the impact of the number of instantiated tasks.

Finally, Heuristic 2 mandates that the iterative search stops if the current decomposition has relative quality lower than some threshold value:

$$Quality_i < (Q_{threshold})^{\frac{cores}{s_{opt}}}, \quad 0 \leq Q_{threshold} \leq 1 \tag{9}$$

The right side of this expression increases with the increase of the parallelism of the optimal task decomposition. Thus, if the optimal found parallelism is close to the theoretical maximum (number of cores in the target machine), finding a better decomposition is unlikely, so the algorithm should tolerate only low quality degradations. On the other hand, if the optimal found parallelism is far from the theoretical maximum, the algorithm should be more aggressive in finding a better decomposition, and therefore allow high degradations of quality.

4.2 Tareador Environment for Automatic Exploration of Parallelism

In order to adapt Tareador for automatic exploration of parallelism, into the original environment we additionally introduced **Paramedir** and search **Driver**. Paramedir [6] (the non-graphical user interface to the Paraver) extracts parallelization metrics described in Sect. 4.1. On the other hand, the Driver iteratively explores parallelism by specifying in each iteration a different *list of tasks* that compose the current task decomposition (Fig. 8). More specifically, the Driver guides the environment through the following steps:

1. **Generate execution logs**: dynamically instrument the sequential application and derive execution logs.
2. **Select the starting decomposition**: put the whole main into one task.
3. **Estimate the parallelism of the current decomposition**: generate traces that estimate the parallelism of the current decomposition.
4. **If the exit condition is fulfilled, finish**: if the *Quality* of the current decomposition is unsatisfactory (Heuristic 2), end the search.
5. **Else, identify the parallelization bottleneck**: process the traces with Paramedir to derive metrics that identify the bottleneck task (Heuristic 1).
6. **Refine the current decomposition to increase parallelism**: break the bottleneck task into its children tasks, if any. Update the *list of tasks* that should be included in the next decomposition.
7. **Proceed to the next iteration**: go to step 3.

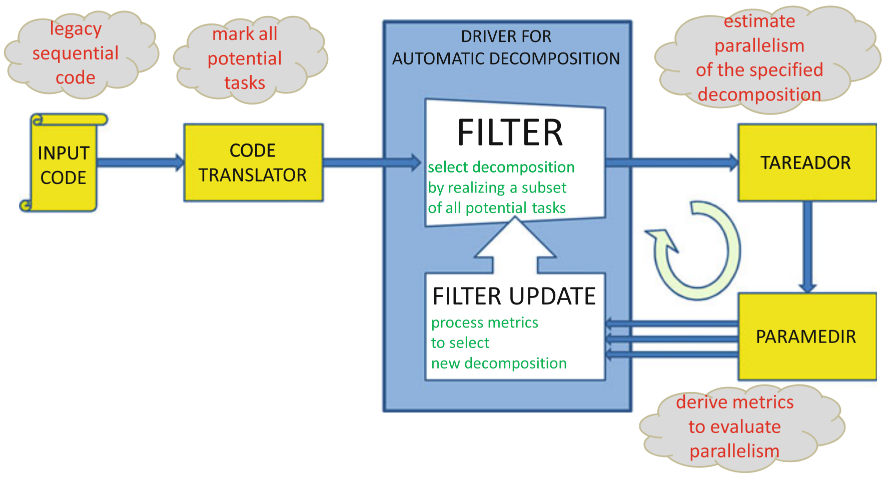

legacy sequential code
mark all potential tasks
DRIVER FOR AUTOMATIC DECOMPOSITION
estimate parallelism of the specified decomposition
INPUT CODE
CODE TRANSLATOR
FILTER
select decomposition by realizing a subset of all potential tasks
TAREADOR
FILTER UPDATE
process metrics to select new decomposition
PARAMEDIR
derive metrics to evaluate parallelism

Fig. 8 Environment to automatically explore possible task decompositions

Table 1 Empirically identified parameters of the automatic search

p_l	p_d	p_c	exp_{tasks}	$Q_{threshold}$
1	1	3	$log_{10}1.5$	0.75

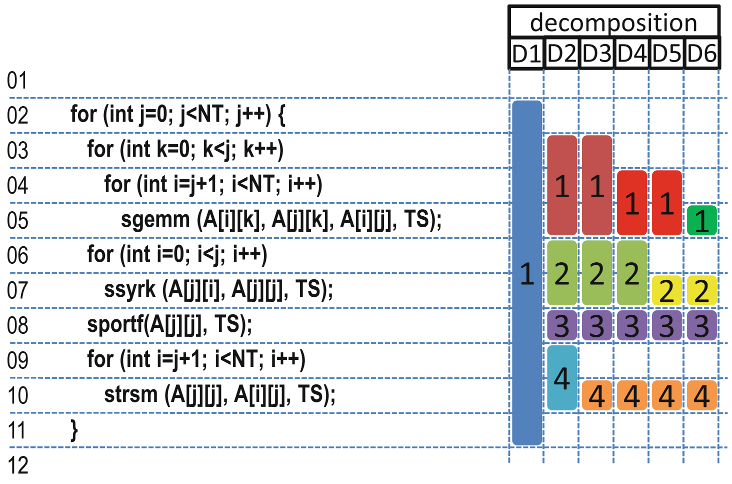
4.3 *Experiments*

Our experiments explore possible parallelization strategies for two well-known applications (Cholesky and LU factorization). We select a homogeneous multi-core processor as the simulated target platform. The goal of our experiments is to show that the proposed search algorithm, metrics and heuristics can find decompositions that provide significant parallelism.

Table 1 lists the empirically identified values for the parameters defined in Sect. 4.1. As already mentioned, the total cost function is a sum of length, dependency and concurrency cost (Eq. 1). Moreover, since our initial experiments showed that concurrency criterion prevails very rarely, we decided to increase the weight of the concurrency cost. Furthermore, in Eq. 8, we set the parameter exp_{tasks} so that the increase of task instances by a factor of 10 is equivalent to the decrease of parallelism by a factor of 1.5. Finally, in Eq. 9, parameter $Q_{threshold}$ was set empirically to allow sufficient quality degradation for a flexible search.

4.3.1 Illustration of the Iterative Search

To illustrate the algorithm we will use the example of parallelizing Cholesky sequential code on a simulated machine with 4 cores. Figure 9 presents (on the left) the code of Cholesky and illustrates (one the right) how the code can be encapsulated into tasks for various decompositions ($D1$–$D6$). Note that marked task types (boxes with numbers) may generate multiple task instances, and that



```
01
02    for (int j=0; j<NT; j++) {
03      for (int k=0; k<j; k++)
04        for (int i=j+1; i<NT; i++)
05          sgemm (A[i][k], A[j][k], A[i][j], TS);
06      for (int i=0; i<j; i++)
07        ssyrk (A[j][i], A[j][j], TS);
08      sportf(A[j][j], TS);
09      for (int i=j+1; i<NT; i++)
10        strsm (A[j][j], A[i][j], TS);
11    }
12
```

Fig. 9 Cholesky: decomposition of the code into tasks

Table 2 Cholesky: task costs (Heuristic 1)

Decomposition	Speedup	Task #1				Task #2				Task #3				Task #4			
		$\bar{l}_i(1)$	$\bar{d}_i(1)$	$\bar{c}_i(3)$	$\bar{t}_i$	$\bar{l}_i(1)$	$\bar{d}_i(1)$	$\bar{c}_i(3)$	$\bar{t}_i$	$\bar{l}_i(1)$	$\bar{d}_i(1)$	$\bar{c}_i(3)$	$\bar{t}_i$	$\bar{l}_i(1)$	$\bar{d}_i(1)$	$\bar{c}_i(3)$	$\bar{t}_i$
D1	1.00	1.00	1.00	1.00	3.00												
D2	1.30	0.51	0.21	0.24	0.96	0.29	0.25	0.12	0.67	0.03	0.13	0.15	0.31	0.17	0.41	0.49	1.06
D3	1.49	0.59	0.44	0.48	1.51	0.34	0.18	0.22	0.74	0.04	0.18	0.27	0.48	0.03	0.21	0.03	0.27
D4	2.30	0.42	0.25	0.04	0.71	0.49	0.24	0.50	1.22	0.05	0.24	0.42	0.70	0.04	0.28	0.04	0.36
D5	3.41	0.72	0.27	0.11	1.10	0.12	0.17	0.13	0.43	0.09	0.25	0.61	0.95	0.07	0.30	0.15	0.52
D6	3.64	0.31	0.21	0.13	0.65	0.30	0.17	0.22	0.70	0.22	0.25	0.50	0.97	0.17	0.36	0.14	0.68

the code outside of marked tasks belongs to the *master task* (sequential part of execution that spawns worker tasks). Table 2 shows the speedup achieved in each decomposition and the costs that guide the iterative search. The algorithm starts from the most coarse-grain decomposition $D1$ that puts the whole execution into one task. There is only one task (#1, lines 2–11), which is automatically the critical task that needs to be broken. Refining $D1$ generates decomposition $D2$ that achieves the speedup of 1.30 (Table 2) and consists of 4 different task types (Fig. 9): #1 that covers the first loop (lines 3–5); #2 that covers the second loop (lines 6–7); #3 that covers function *spotrf_tile* (line 8); and #4 that covers the third loop (lines 9–10). Heuristic 1 identifies task #4 as the most critical, mostly due to its high concurrency cost. Thus, the following decomposition ($D3$) breaks the task #4 and obtains the parallelism of 1.49. In $D3$, the algorithm identifies task #1 as the bottleneck (due to its high length). Further iterations of the algorithm pass through decompositions $D4$, $D5$ and $D6$ that provide speedups of 2.30, 3.41 and 3.64, respectively.

4.3.2 Results

This subsection presents the results obtained by applying our algorithm on a set of applications. For each application, we present four plots that illustrate the process

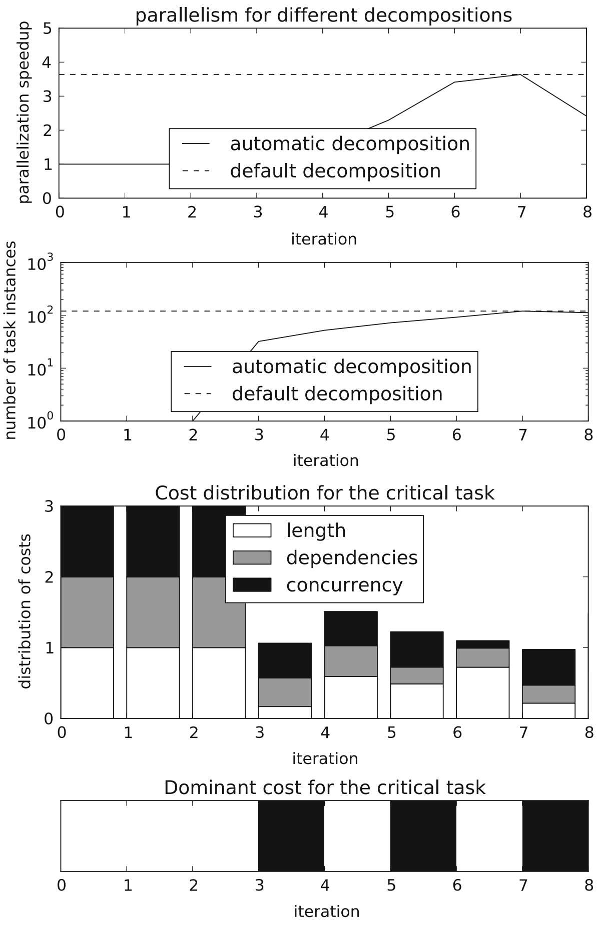


Fig. 10 Cholesky on 4 cores

of automatic task decomposition. The first plot presents the parallelism of all tested decompositions – the speedup over the sequential execution of the application. The second plot shows the number of task instances generated by each decomposition. Also, the first two plots show the parallelism and the number of instances in the *reference task decomposition* (the decomposition selected and implemented by an expert programmer). The third plot presents the cost distribution for the bottleneck task of each iteration. Finally, the fourth plot shows the most dominant cost for the bottleneck task.

The proposed search algorithm finds decompositions with very high parallelism, sometimes finding the decomposition manually selected by an expert programmer. The algorithm finds the reference decomposition for Cholesky in iteration 7 (Fig. 10). In order to get to this decomposition, the algorithm refines decompositions based on the concurrency criterion in iterations 3 and 5. Soon after finding the

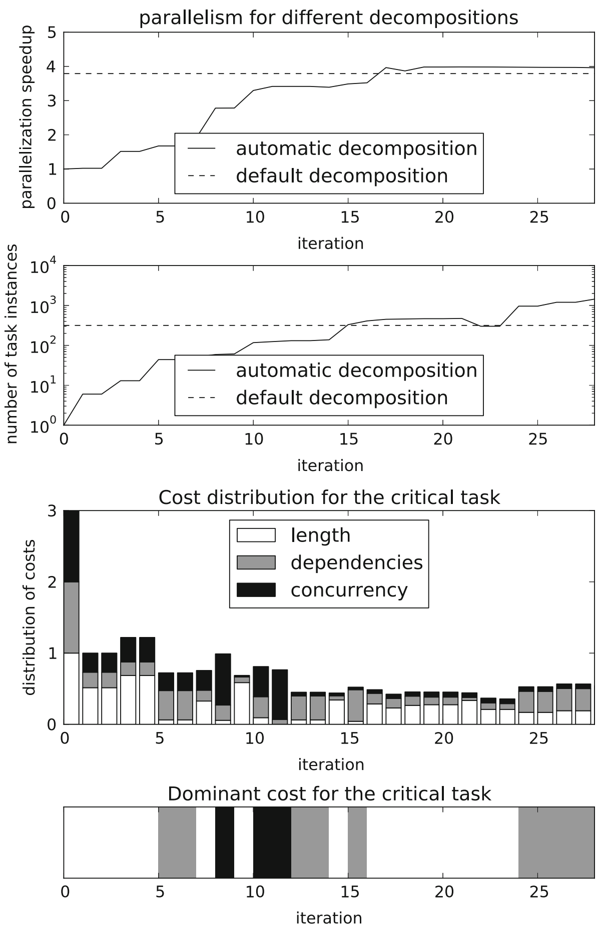


Fig. 11 Sparse LU on 4 cores

reference decomposition, the algorithm passes through the decomposition that activates the mechanism for stopping the search (Heuristic 2).

Sparse LU (Fig. 11), as a more complex application, demonstrates the power of our search. Compared to Cholesky, Sparse LU forces the algorithm to use various bottleneck criteria through the exploration of decompositions. It is interesting to note that the search finds a wide range of decompositions (iterations 17–28) that provide higher parallelism than the reference decomposition. In this case, it is unclear which of these decompositions is the optimal one. Quantitative reasoning suggests that the optimal task decomposition is the one that provides highest parallelism with the lowest number of created task instances. Following this reasoning, the optimal decomposition (iteration 22) achieves the speedup of 3.98 with the cost of 301 instantiated tasks (note the sudden drop in the number of task instances). On the

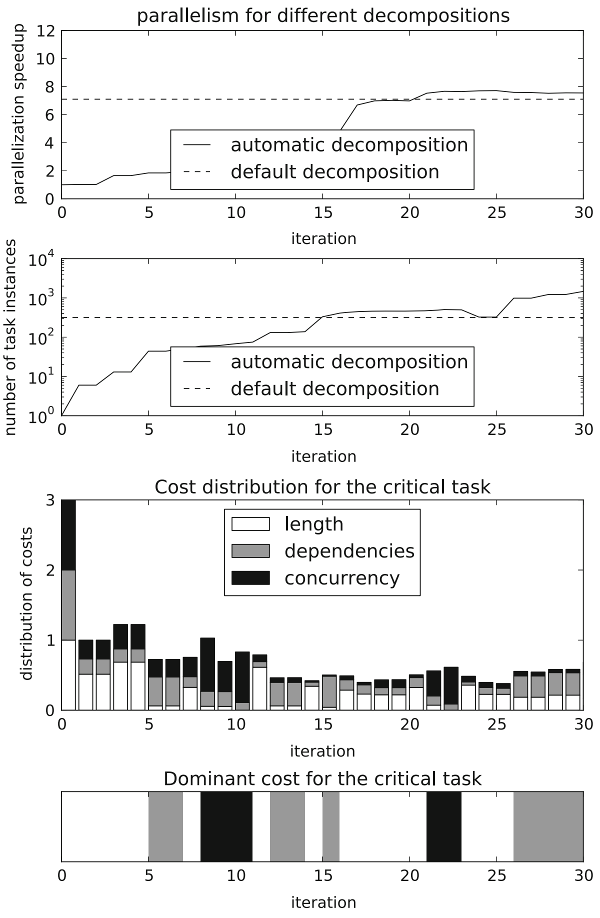


Fig. 12 Sparse LU on 8 cores

other hand, qualitative reasoning suggests that, within a set of decompositions that provide similar parallelism generating a similar number of instances, the optimal decomposition is the one that is the easiest to express using semantics offered by the target parallel programming model. For example, our algorithm may find a decomposition that extracts very irregular parallelism that cannot be expressed using a fork-join programming model. In that case, it is programmer's responsibility to, out of few offered efficient task decompositions, identify the one that can be straightforwardly implemented using a specific programming model.

It is also interesting to study how the algorithm adapts to the target parallel machine. Changing the parallelism of the target machine changes the simulation of the parallel execution of the tested decomposition. Thus, changes the normalized concurrency cost, while dependency and length cost remain the same. Figures 12 and 13 illustrate potential decompositions for Sparse LU for executing on machines

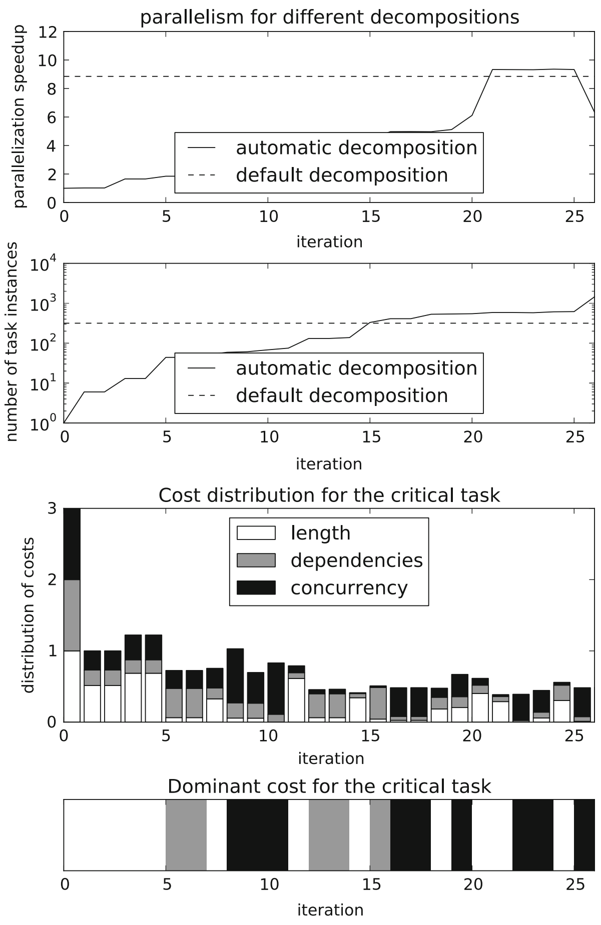


Fig. 13 Sparse LU on 16 cores

with 8 and 16 cores. In the experiments with 8-core target machine (Fig. 12), the reference decomposition achieves the speedup of 7.1 at the cost of generating 316 task instances. The automatic search finds a wide range of decompositions (iterations 21–30) that provide slightly higher parallelism than the reference decomposition. On the other hand, in the experiments with 16-core target machine (Fig. 13), the reference decomposition achieves the speedup of 8.85 (316 instances). The algorithm finds only five decompositions (iterations 21–25) that provide higher parallelism than the default decomposition. It is also interesting to note that in the experiment with 16-core target machine, the algorithm more often refines the decomposition using the concurrency criterion. This happens because, despite the fine granularity of decompositions, the algorithm cannot find decomposition with parallelism close to the theoretical maximum of 16 (number of cores in the target machine).

5 Discussion: Tareador in the Big Scheme of Things

This section describes our idea of Tareador's role in the complete solution for parallelizing applications. As already mentioned, the solution for parallelization of applications must include not only the development tool but also the appropriate parallel programming model and the parallelization workflow. Parallel programming model should be chosen so the results obtained by the development tool can be useful in expressing parallelism. On the other hand, the parallelization workflow should describe how the development tool is used in the process of parallelizing applications for the target parallel programming model.

5.1 OmpSs (OpenMP 4.0)

In order to design a good parallelization solution, the target parallel programming model should follow the philosophy of the development tool. Different parallel programming models mainly differ in the nature of the parallelism that can be exploited, and the way in which the parallelism is expressed. Most of the mainstream parallel programming models rely on the fork-join parallelism [7, 8] Fork-join parallelism in the original context follows all-or-nothing philosophy – it distinguishes between the sequential sections that have no parallelism and parallel sections where everything is parallel. Thus, these programming model allow expressing which code sections are sequential and which parallel. Other programming models rely on the dataflow paradigm [9, 10]. Dataflow parallelism tries to define for each pair of tasks whether they are dependent or not. In most of the dataflow programming models, the programmer expresses memory usage of each task type and then the runtime system dynamically calculates intertask dependencies.

In a parallelization solution that includes Tareador, we propose using OmpSs as the target parallel programming model. OmpSs [9] is a programming model developed in Barcelona Supercomputing Center as a forefront for OpenMP [7]. In fact, most of the main OmpSs ideas are already introduced in OpenMP 4.0 standard [11]. OmpSs is a directive-based dataflow parallel programming model. Figure 14 illustrates how OmpSs extracts parallelism in Cholesky kernel. Figure 14a shows the sequential Cholesky code (code in black-bold) and the OmpSs pragma directives needed to expose parallelism (code in gray). Thus, if a programmer wants some function to execute as a task, she must annotate the function declaration with the pragma directive that specifies the directionality of each function argument (input, output, inout). Based on these directives, the runtime system dynamically generates the dependency graph of all task instances (Fig. 14b). Once the dependency graph is generated, the runtime can execute task instances out of order, as long as dependencies are satisfied. This type of dataflow execution allows extracting very irregular parallelism that cannot be expressed with fork-join parallelism.

a

```
#pragma omp task inout ([TS][TS]A)
void sportf (float *A, int TS);
#pragma omp task input ([TS][TS]T) inout ([TS][TS]B)
void strsm (float *T, float *B, int TS);
#pragma omp task input ([TS][TS]A, [TS][TS]B) inout ([TS][TS]C)
void sgemm (float *A, float *B, float *C, int TS);
#pragma omp task input ([TS][TS]A) inout ([TS][TS]C)
void ssyrk (float *A, float *C, int TS);

void cholesky (int NT, float *A[NT][NT]) {
  for (int k=0; k<NT; k++) {
  ●sportf(A[k][k], TS);
    for (int i=k+1; i<NT; i++)
  ●   strsm (A[k][k], A[k][i], TS);
    for (int i=k+1; i<NT; i++) {
      for (int j=k+1; j<i; j++)
  ●       sgemm (A[k][i], A[k][j], A[j][i], TS);
  ●   ssyrk (A[k][i], A[i][i], TS);
    }
  }
}
```

b

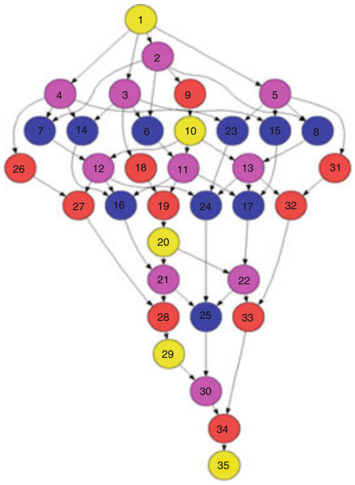


Fig. 14 Cholesky parallelized with OmpSs. (**a**) Source code. (**b**) Task dependency graph (**Note**: The code listing marks for each function the color of the node that represents it in the dependency graph)

Furthermore, numerous research papers prove that OmpSs outperforms OpenMP in many well-known scientific applications [12, 13].

We consider OmpSs programming model as a specially good fit for Tareador development tool. For a specified task decomposition, Tareador dynamically calculates the memory usage of each task and then calculates potential inter-task dependencies. Therefore, the information obtained by Tareador naturally maps to the information needed to express OmpSs parallelism (directionality of function arguments inside pragmas). Furthermore, OmpSs facilitates automatic generation of parallel code. Being a directive-based parallel programming model, OmpSs adds pragma annotations, but maintains the original code structure of the sequential application. Therefore, it is easy to suggest how to change the sequential code in order to parallelize it, much easier than in the case when the code structure needs to be modified (for instance, in the case of pthreads [14]).

5.2 Parallelization Workflow

Figure 15 illustrates the parallelization workflow for using Tareador. At the current stage of development, Tareador serves as a tool to explore potential parallelism in sequential applications. Thus, starting from the sequential code (#0), Tareador LLVM module (#1) generates execution logs (#2). Processing the logs, Tareador GUI explores various task decompositions (#3) and evaluates their potential

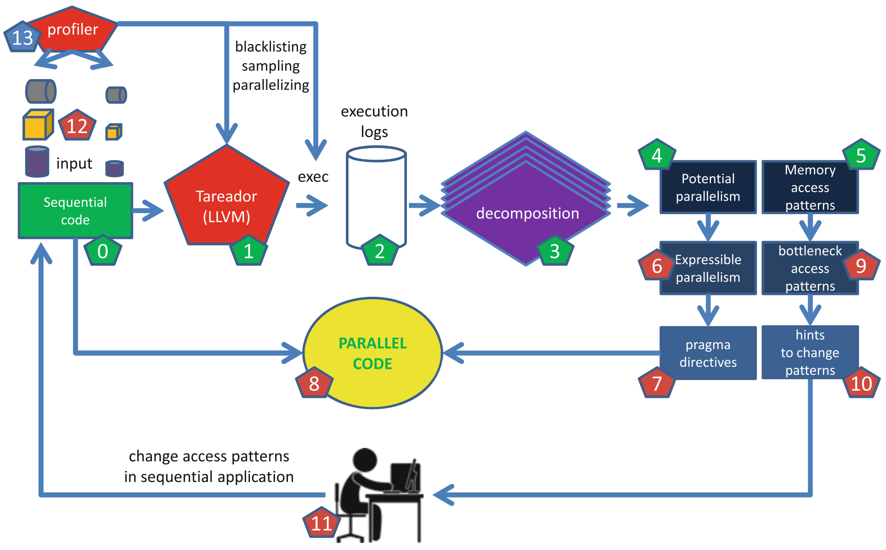


Fig. 15 Parallelization workflow using Tareador

parallelism (#4). Furthermore, Tareador collects and visualizes memory access patterns of all all tasks (#5). This part of development is already finished.

The rest of the Figure represents planed future development of Tareador environment. Currently, Tareador evaluates only unbounded parallelism – parallelism not limited by constraints of the target parallel programming model. In other words, Tareador identifies two tasks as dependent iff the first task writes at least on byte that the second task reads. However, Tareador omits to consider whether the tested decomposition can be implemented – whether the targeted parallel programming model offers parallelization primitives that can express the identified parallelism. Thus, the next step for Tareador is to evaluate the portion of the potential parallelism that can be expressed using the target parallel programming model (#6). Once this step is finished, Tareador should output the content of pragma primitives that are required to expose parallelism (#7). Thus, the correct parallel code could be obtained (#8). However, in most applications, the parallel efficiency of the code obtained this way will be unsatisfactory.

The potential of automatic parallelization is limited by unfavorable access patterns to some memory objects. If simple decomposition of the sequential code into tasks cannot provide sufficient expressible parallelism, Tareador should pinpoint the memory objects with unfavorable access patterns (#9). Also, Tareador should suggest to the programmer how to change these access patterns (#10) in order for the automatic parallelization to be more efficient. Then, the programmer should manually change the culprit access patterns (#11) and pass the new application for the next attempt of automatic parallelization. It is important to note that in the proposed workflow, the programmer modifies only the sequential application and adapts it so the automatic parallelization could be more efficient. Finally, Tareador should process the application for different inputs (#12) that exercise potentially different parts of the source code. The parallelization strategy that suits all tested inputs should be accepted as the final one.

Nevertheless, Tareador development will continuously dedicate to optimization of the instrumentation (#13). Currently we are developing the profiler tools whose output should facilitate optimization of the original Tareador instrumentation. Firstly, the profile information should allow better blacklisting of code sections that are promoted into potential tasks. Furthermore, profiler should allow sampling by pinpointing smaller parts of execution that can be representatives of the whole run. Also, profile information should suggest how to parallelize the instrumentation by separately instrumenting different parts of execution and then merging the obtained logs. Finally, the profiler should monitor the process of reducing the problem size while preserving the characteristic behavior of the application.

6 Related Work

Numerous tools to assist parallelization have been proposed in the past years both from the academia and the industry. Regarding tools proposed by the academia, the ones closest to the environment that has been proposed in this paper are Embla, Kremlin, and Alchemist. In particular, Embla [15] is a Valgrind-based tool that estimates the potential speed-up for Cilk programs. On the other hand, Kremlin [16] identifies regions of a serial program that can be parallelized with OpenMP and proposes a parallelization planner for the user to parallelize the target program. Finally, Alchemist [17] identifies parts of code that are suitable for thread-level speculation. The major drawbacks of these tools are that they are limited to fork-join parallelism and that they offer very little qualitative information about the target program (no useful visualization support).

On the other side, the industry have also been recently developing their solutions for assisted parallelization. For example, Intel's Parallel Advisor [18] assists parallelization with Thread Building Blocks (TBB) [19]. Parallel Advisor provides timing profile that suggests to the programmer which loops should be parallelized. Critical Blue provides Prism [20], a tool to do "what-if" analysis that anticipates the potential benefits of parallelizing certain parts of the code. Vector Fabrics provides Pareon [21], another tool for "what-if" analysis to estimate the benefits of parallelizing loop iterations. All the three mentioned tools provide rich GUI and visualization of the potential parallelization. However, none of the tools offers automatic exploration of parallelization strategies. Moreover, they do not provide any API to automate the search for the optimal parallelization strategy as the one proposed in this paper.

7 Conclusion and Future work

The software community is facing a paramount task of parallelizing the existing body of sequential applications. Current mainstream hardware provides extremely high parallelism, but state-of-the-art sequential software cannot take advantage of this abundance of resources. To adapt current applications for the novel hardware, we must approach the challenge of parallelization.

In order to tackle this issue, in this paper we present Tareador – a tool for assisted parallelization. Tareador allows to the programmer to easily browse various parallelization strategies and choose the one that promises the highest parallelization potential. Furthermore, Tareador provides very rich visualization of the results, offering deeper insight into the potential parallelism and pinpointing the culprits for low performance. We showed how Tareador is used for teaching parallelism at the University courses, as well as for actually parallelizing sequential applications.

At the current stage of development, Tareador is useful for exploring the potential parallelism inherent in the applications. However, we describe our future development directions in order to upgrade Tareador into a tool for automatic parallelization of sequential applications. We also blueprint the parallelization process that the programmer should follow in order to use Tareador to port sequential application to OmpSs parallel programming model.

References

1. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis and transformation, San Jose, pp. 75–88 (2004)
2. Girona, S., Labarta, J., Badia, R.: Validation of dimemas communication model for MPI collective operations. In: EuroPVM/MPI'2000, Lake Balaton (2000)
3. Pillet, V., Labarta, J., Cortes, T., Girona, S.: PARAVER: a tool to visualize and analyze parallel code. In: WoTUG-18, Manchester (1995)
4. Gansner, E.R., North, S.C.: An open graph visualization system and its applications to software engineering. Software – Practice and Experience **30**(11), 1203–1233 (2000)
5. Subotic, V., Ferrer, R., Sancho, J.C., Labarta, J., Valero, M.: Quantifying the potential task-based dataflow parallelism in MPI applications. In: Euro-Par (1), Bordeaux, pp. 39–51 (2011)
6. Jost, G., Labarta, J., Gimenez, J.: Paramedir: a tool for programmable performance analysis. In: International Conference on Computational Science, Kraków, pp. 466–469 (2004)
7. Dagum, L., Menon, R.: OpenMP: an industry-standard API for shared-memory programming. Comput. Sci. Eng. **5**, 46–55 (1998)
8. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: an efficient multithreaded runtime system. J. Parallel Distrib. Comput. **37**, 55–69 (1996)
9. Duran, A., Ayguadé, E., Badia, R.M., Labarta, J., Martinell, L., Martorell, X., Planas, J.: Ompss: a proposal for programming heterogeneous multi-core architectures. Parallel Process. Lett. **21**(2), 173–193 (2011)
10. K. Fatahalian, Horn, D.R., Knight, T.J., Leem, L., Houston, M., Park, J.Y., Erez, M., Ren, M., Aiken, A., Dally, W.J., Hanrahan, P.: Memory – sequoia: programming the memory hierarchy. In: SC, New York, p. 83 (2006)

11. OpenMP Architecture Review Board: OpenMP Application Program Interface Version 4.0. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf. Active on July 2013
12. Pérez, J.M., Badia, R.M., Labarta, J.: A dependency-aware task-based programming environment for multi-core architectures. In: CLUSTER, Tsukuba, pp. 142–151 (2008)
13. Marjanovic, V., Labarta, J., Ayguadé, E., Valero, M.: Overlapping communication and computation by using a hybrid MPI/SMPSs approach. In: ICS, Tsukuba, pp. 5–16 (2010)
14. Nichols, B., Buttlar, D., Farrell, J.P.: Pthreads Programming. O'Reilly & Associates, Sebastopol (1996)
15. Mak, J., Faxén, K.-F., Janson, S., Mycroft, A.: Estimating and exploiting potential parallelism by source-level dependence profiling. In: Euro-Par (1), Ischia, pp. 26–37 (2010)
16. Garcia, S., Jeon, D., Louie, C.M., Taylor, M.B.: Kremlin: rethinking and rebooting gprof for the multicore age. In: PLDI, San Jose, pp. 458–469 (2011)
17. Zhang, X., Navabi, A., Jagannathan, S.: Alchemist: a transparent dependence distance profiling infrastructure. In: CGO '09, Seattle (2009)
18. Intel Corporation: Intel Parallel Advisor. http://software.intel.com/en-us/intel-advisor-xe. Active on 10.11.2014
19. Pheatt, C.: Intel threading building blocks. J. Comput. Sci. Coll. **23**, 298–298 (2008)
20. Critical Blue: Prism. http://www.criticalblue.com/. Active on 10.11.2014
21. Vector Fabrics: Pareon. http://www.vectorfabrics.com/products. Active on 10.11.2014