Automatic Exploration of Potential Parallelism in Sequential Applications

Vladimir Subotic[1], Eduard Ayguadé[1,2], Jesus Labarta[1,2], and Mateo Valero[1,2]

[1] Barcelona Supercomputing Center, Barcelona, Spain
`vladimir.subotic@bsc.es`
[2] Universitat Politecnica de Catalunya, Barcelona, Spain

Abstract. The multicore era has increased the need for highly parallel software. Since automatic parallelization turned out ineffective for many production codes, the community hopes for the development of tools that may assist parallelization, providing hints to drive the parallelization process. In our previous work, we had designed Tareador, a tool based on dynamic instrumentation that identifies potential task-based parallelism inherent in applications. Also, we showed how a programmer can use Tareador to explore the potential of different parallelization strategies. In this paper, we build up on our previous work by automating the process of exploring parallelism. We have designed an environment that, given a sequential code and configuration of the target parallel architecture, iteratively runs Tareador to find an efficient parallelization strategy. We propose an autonomous algorithm based on simple metrics and a cost function. The algorithm finds an efficient parallelization strategy and provides the programmer with sufficient information to turn that parallelization strategy into an actual parallel program.

Keywords: automatic parallelization, potential parallelism, OmpSs.

1 Introduction

For decades, microprocessors have been improving their performance following Moore's law without requiring major changes in the applications. In essence, the performance improvements relied on both architectural techniques that improve ILP (instruction-level parallelism) and compilers that optimize the code for each target architecture. Unfortunately, the improvements achieved by ILP had entered stagnation.

On the other hand, multicore processor architectures are now the norm in high-performance processors. Multicore processors have introduced the need to re-design applications in order to utilize the increasing number of available cores. Applications that before were running sequentially, now must be parallelized in order to efficiently harness the full potential of multicore processors. However, parallelizing existing applications is not an easy task. As the software community struggles to fulfill this demand, the gap between parallel hardware and sequential software keeps growing.

It is believed that neither the compiler nor the hardware itself will automatically detect and exploit the parallelism needed to feed current and future multi-/many-core architectures. Despite decades of research efforts [4,1,16] on auto-parallelization, and the inclusion of auto-parallelization features in some commercial compilers [2], the experience have shown that they have very limited applicability. In the current scenario, in which systems (from mobile to desktop/laptop and servers) are based mostly on parallel architectures, programmers must use explicit parallel programming techniques.

To help in the process of parallelization, the community has developed several tools to assist the parallelization process (see Section 7). These tools usually target a specific parallel programming model or language and/or impose constraints of the possible strategies to explore (e.g. loops).

In our prior work, we proposed Tareador [15] as a tool to analyze the potential parallelism inherent in applications. We also described an iterative top-down trial-and-error process to find suitable parallelization strategies. However, the presented process relied strongly on programmer's experience to guide the search.

In this paper, we propose an automatic exploration of parallelization strategies. Our goal is to formalize the programmers experience into an autonomous algorithm that can find an effective task decomposition of a sequential code. More specifically, our work provides the following contributions:

1. **Definition of a set of metrics and heuristics** that drive the automatic exploration of parallelization strategies. Our heuristics mandate the policy of refining decomposition in order to increase parallelism, as well as the end of the iterative exploration. The proposed metrics are parameterizable so they can be customized according to the targeted sequential code.
2. **Design of an environment** that leverages Tareador to automatically explore parallelism in sequential codes. The designed tool-chain iteratively tests various task decompositions, illustrating a reasonable exploration path for exposing parallelism inherent in the code. Furthermore, the environment offers visualization of the parallel execution of the tested decompositions.

The paper is organized as follows. Section 2 briefly summarizes the Tareador environment. Section 3 describes the proposed algorithm (metrics and heuristics) that autonomously explores task decomposition strategies and Section 4 presents the implemented environment. In Section 5 we present the results obtained for a set of simple applications, while in Section 6 we discuss how our approach could be applied to realistic workloads. Finally, Section 7 describes some related work and environments and Section 8 concludes the paper.

2 Background: Tareador

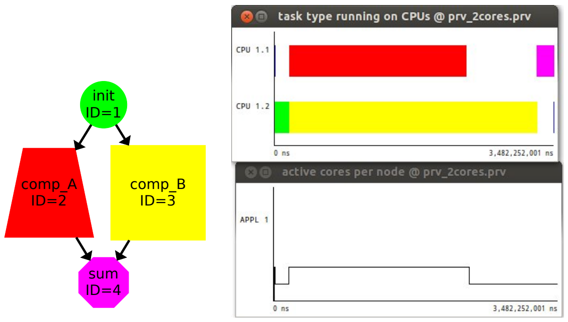
Tareador [15] is a tool for assisted parallelization of sequential applications. Using the Tareador API, the programmer annotates the sequential code to propose a task decomposition. Then, the tool (implemented as a Valgrind [12] plugin) dynamically instruments the code in order to collect all memory accesses within

```
tareador_start_task("init");
A = random_buff ();
B = reset_buf ();
tareador_end_task();

tareador_start_task("comp_A");
compute(A);
tareador_end_task();

tareador_start_task("comp_B");
compute(B);
tareador_end_task();

tareador_start_task("sum");
res += sum (A);
res += sum (B);
tareador_end_task();
```



(a) Input code (b) task graph (c) Paraver views

Fig. 1. Tareador instrumentation

each specified task. Based on the collected accesses, Tareador derives the data dependencies among the tasks and estimates the potential parallelism of the task decomposition. In our prior work [15] we demonstrated how a programmer can use Tareador to iteratively explore the task decomposition space and find a decomposition that exposes sufficient parallelism to efficiently deploy multi-core processors. Depending on the application (granularity of tasks, number of dependencies, ...), Valgrind instrumentation introduces the slowdown of $200x$-$1000x$ compared to the native sequential execution of the target application.

Tareador provides to a programmer a simple and flexible API to propose how a sequential code could be decomposed into tasks. Namely, the programmer invokes *tareador_start_task* to mark the beginning of a task, and *tareador_end_task* to mark the end of a task. The interface allows specification of any arbitrary task decomposition, even if the targeted code is badly structured or recursive (nesting of tasks is supported). No other refactoring of the targeted sequential code is needed. Figure 1a illustrates a simple code with Tareador annotations.

As an evaluation of the proposed decomposition, Tareador provides to the programmer two outputs: the dependency graph of all tasks; and the simulation of the potential parallel execution. Figure 1b shows the task graph for the previous example: a node represents a dynamic task instance and an edge represents a dependency between two task instances. In this example, the graph suggests to the programmer that there is potential concurrency only between task *comp_A* and task *comp_B*. Moreover, Figure 1c shows the timeline for the simulated parallel execution on a target processor with 2 cores. The upper timeline (horizontal axis is time) shows which task executes on each core and the lower timeline shows the number of active cores throughout the simulated execution. The same colors are used to represent matching tasks in the graph and in the timeline, helping the programmer to identify potential bottlenecks in the current task decomposition (for example, in this case to observe the load imbalance that occurs in the parallel execution of tasks *comp_A* and *comp_B*).

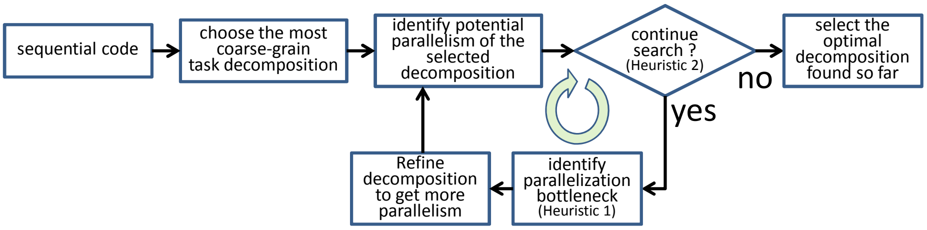

sequential code
choose the most coarse-grain task decomposition
identify potential parallelism of the selected decomposition
continue search ? (Heuristic 2)
no
select the optimal decomposition found so far
yes
identify parallelization bottleneck (Heuristic 1)
Refine decomposition to get more parallelism

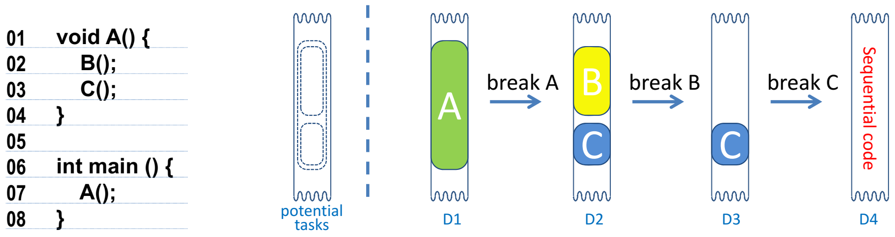
Fig. 2. Algorithm for exploring parallelization strategies

3 Automatic Exploration of Parallelization Strategies

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 Figure 2. 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).
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.

In the following sections, we further describe the design choices made in designing the mentioned heuristics. Nevertheless, first we must define a 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



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

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

Fig. 3. Iterative refinement of decompositions

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, Figure 3 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.

3.1 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 rest of this section further describes the operands from Equation 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:

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

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:

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

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:

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

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

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

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.

3.2 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:

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

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 parallelism is close to

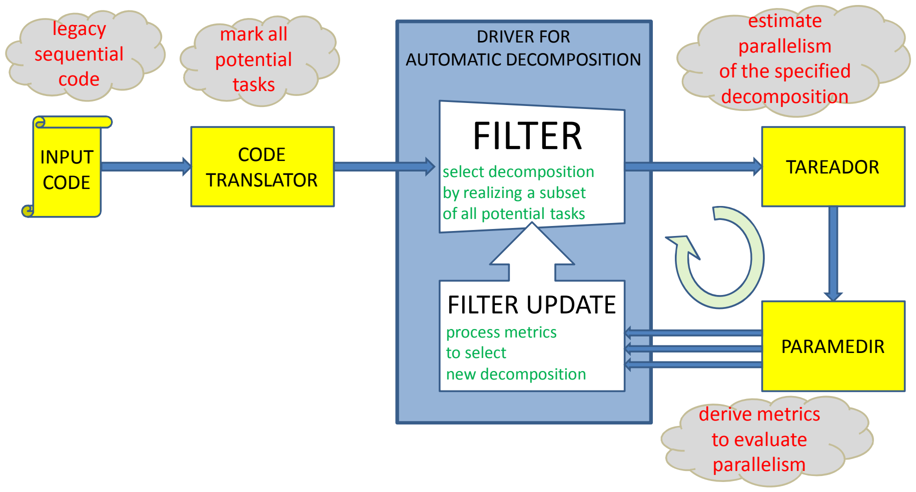

legacy sequential code
mark all potential tasks
DRIVER FOR AUTOMATIC DECOMPOSITION
estimate parallelism of the specified decomposition
INPUT CODE
CODE TRANSLATOR
FILTER
select decomposition by realizing a subset of all potential tasks
TAREADOR
FILTER UPDATE
process metrics to select new decomposition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

Fig. 4. Environment to automatically explore possible task decompositions

the theoretical maximum (number of cores in the target machine), finding a better decomposition is unlikely, so the algorithm should not tolerate high 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 Designed Environment

Our environment for automatic exploration of task decompositions (Figure 4) consists of: Tareador, Paramedir and the Driver. In addition, a source-to-source translator automatically annotates all potential tasks in the code (main function, each function call and each loop). Tareador [15] evaluates the potential parallel execution of a task decomposition. Paramedir [9], the non-graphical user interface to the Paraver [14] analysis tool, extracts performance metrics of the simulated parallel execution. Finally, the Driver is a glue that integrates all the mentioned tools in a common environment. Its important to stress that the most computation intensive processing (Valgrind instrumentation) is performed only once, and then the generated logs are used offline to test various task decompositions. The following paragraph describes this integration in more detail.

The main functionality of the Driver is to guide the iterative decomposition refinement. In each iteration, Driver specifies a *list of tasks* that compose the current task decomposition. Initially the list contains only the main function of the program. The Driver automates the process of exploring potential decompositions by guiding the environment through the following steps:

1. **Generate execution logs:** use Valgrind to dynamically instrument the application and derive memory usage logs.

2. **Select the starting decomposition:** put the whole main into one task.
3. **Estimate the parallelism of the current decomposition:** run Tareador to generate traces that describe 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.

5 Experiments

Our experiments explore possible parallelization strategies for four well-known applications (Jacobi, HM transpose, 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 sufficient parallelism.

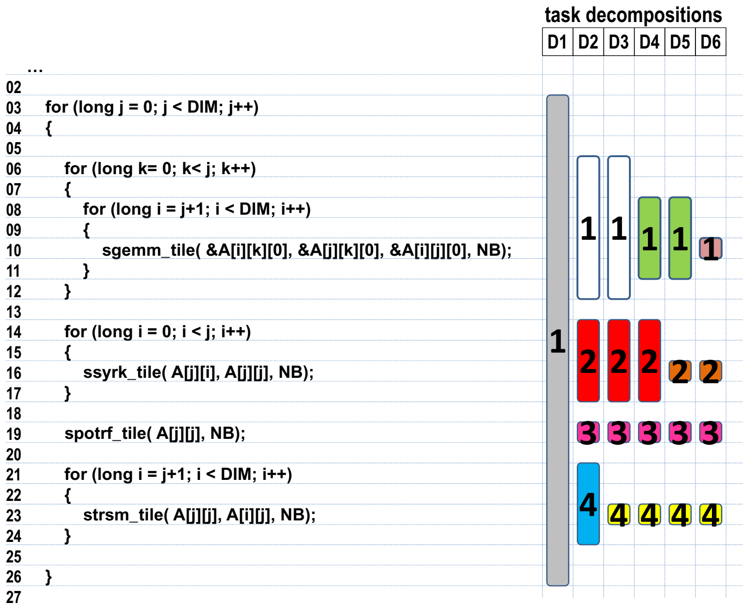
Table 1. Empirically identified parameters of the automatic search

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

Table 1 lists the empirically identified values for the parameters defined in Section 3. As already mentioned, the total cost function is a sum of length, dependency and concurrency cost (Equation 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 Equation 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 speedup by a factor of 1.5 . Finally, in Equation 9, parameter $Q_{threshold}$ was set empirically to allow sufficient quality degradation for a flexible search.

5.1 Illustration of the Iterative Search

This subsection illustrates our algorithm on a couple of small examples. We start from the example presented in Section 2. Table 2 enlists the tasks costs for example from Figure 1. Since tasks *comp_B* is the longest (Figure 1c), it gets the highest length cost. On the other hand, because of their very short length, tasks *init* and *sum* get very low length costs. The Table also shows that all tasks have the same dependency cost. This is because, for each task, the sum of incoming



```
...

for (long j = 0; j < DIM; j++)
{

  for (long k= 0; k< j; k++)
  {
    for (long i = j+1; i < DIM; i++)
    {
      sgemm_tile( &A[i][k][0], &A[j][k][0], &A[i][j][0], NB);
    }
  }

  for (long i = 0; i < j; i++)
  {
    ssyrk_tile( A[j][i], A[j][j], NB);
  }

  spotrf_tile( A[j][j], NB);

  for (long i = j+1; i < DIM; i++)
  {
    strsm_tile( A[j][j], A[i][j], NB);
  }

}

```

Fig. 5. Cholesky: decomposition of the code into tasks

and outgoing dependencies is equal to 2 (Figure 1b). Figure 1c also explains the tasks' concurrency costs. Since during the whole execution of task *comp_A* both cores are utilized, this task has the lowest concurrency cost. On the other hand, during the execution of tasks *init* and *sum* there is only one core active, so these tasks have the highest concurrency cost. Finally, task *comp_B* obtains the highest total cost of 0.90. Thus, in this example, our algorithm would identify *comp_B* as the bottleneck task, dominantly following the length criterion.

The second illustration of the algorithm uses the example of parallelizing Cholesky sequential code on a simulated machine with 4 cores. Figure 5 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 the code outside of marked tasks belongs to the *master task* (sequential part of execution that spawns worker tasks). Table 3 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

Table 2. Tasks costs for the example from Figure 1

init				comp_A				comp_B				sum			
$\bar{l}_i(1)$	$\bar{d}_i(1)$	$\bar{c}_i(3)$	$\bar{t}_i$	$\bar{l}_i(1)$	$\bar{d}_i(1)$	$\bar{c}_i(3)$	$\bar{t}_i$	$\bar{l}_i(1)$	$\bar{d}_i(1)$	$\bar{c}_i(3)$	$\bar{t}_i$	$\bar{l}_i(1)$	$\bar{d}_i(1)$	$\bar{c}_i(3)$	$\bar{t}_i$
0.03	0.25	0.42	0.70	0.39	0.25	0.05	0.69	0.54	0.25	0.11	0.90	0.04	0.25	0.42	0.71

Table 3. 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 whole execution into one task. There is only one task (#1, lines 3-26), which is automatically the critical task that needs to be broken. Refining $D1$ generates decomposition $D2$ that achieves the speedup of 1.30 (Table 3) and consists of 4 different task types (Figure 5): #1 that covers the first loop; #2 that covers the second loop; #3 that covers function *spotrf_tile*; and #4 that covers the third loop. Heuristic 1 identifies task #4 (lines 21-24) 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 (lines 6-12) 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.

5.2 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 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 OmpSs[6] 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, often finding the decomposition manually selected by an expert programmer. The algorithm finds the reference decomposition for Jacobi and HM transpose (Figures 6 and 7) in iterations 4 and 5, respectively. In these two applications, the algorithm bases its decisions mainly on the length criterion. The algorithm also finds the reference decomposition for Cholesky in iteration 7 (Figure 8). However, in order to get to this decomposition, the algorithm refines decompositions based on the concurrency criterion in iterations 3 and 5. In all three applications, soon after finding the reference decomposition, the algorithm passes through decompositions that activate the mechanism for stopping the search (Heuristic 2).

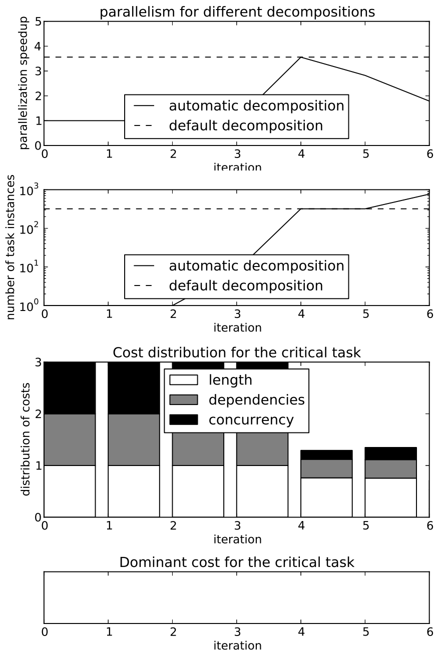


Fig. 6. Jacobi on 4 cores

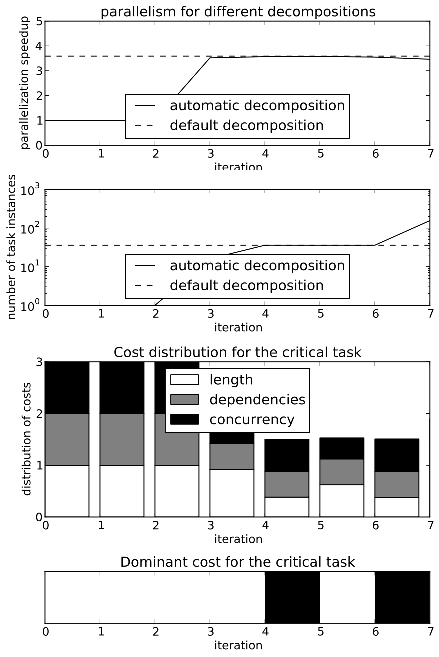


Fig. 7. HM transpose on 4 cores

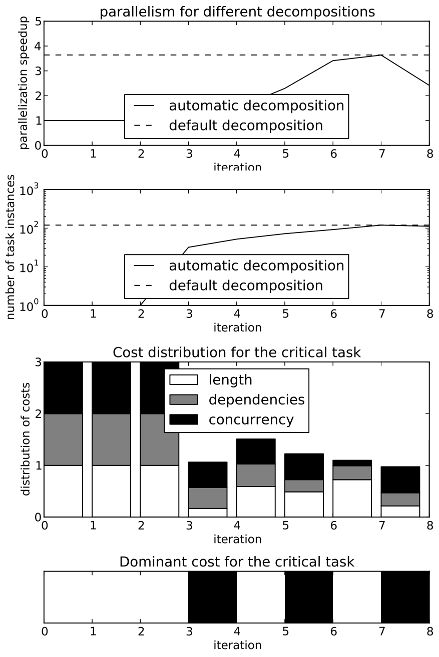


Fig. 8. Cholesky on 4 cores

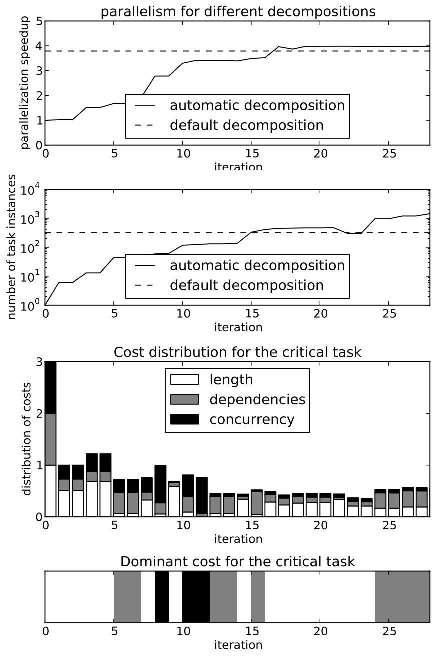


Fig. 9. Sparse LU on 4 cores

Sparse LU (Figure 9), as the most complex of the studied applications, demonstrates the power of our search. Compared to the previous codes, 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 other hand, qualitative reasoning suggests that, within a set of decompositions that provide a 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 programmers 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

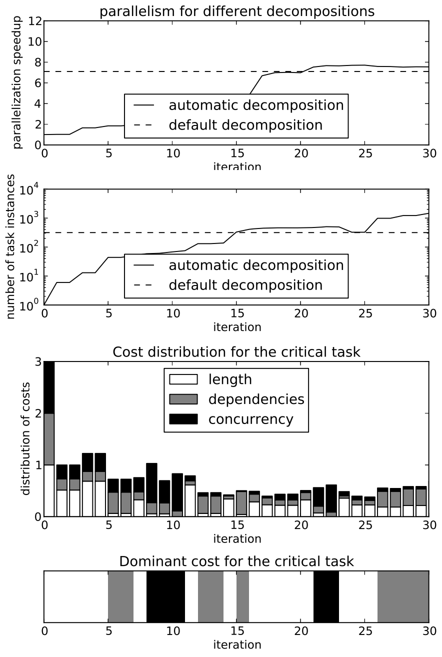


Fig. 10. Sparse LU on 8 cores

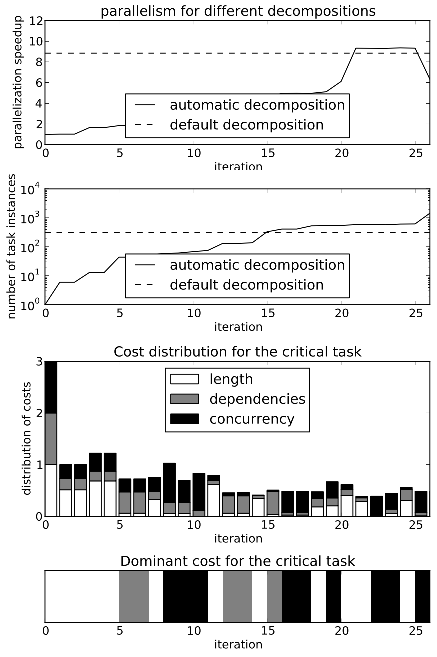


Fig. 11. Sparse LU on 16 cores

of the parallel execution of the tested decomposition. Thus, changes the normalized concurrency cost, while dependency and length cost remain the same. Figures 10 and 11 illustrate potential decompositions for Sparse LU for executing on machines with 8 and 16 cores. In the experiments with 8-core target machine (Figures 10), the reference OmpSs 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 (Figures 11), 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).

6 Discussion: Biting the Bullet of Real Workloads

This paper demonstrates that our automatic technique can find significant parallelism in a few small applications. In this section, we discuss techniques to extend scalability and applicability of our approach, and therefore allow processing realistic workloads.

Scalability of our approach concerns the execution time of the automatic search. Valgrind dynamic instrumentation presents the most computation intensive part of our technique. However, we already significantly reduced the impact of dynamic instrumentation, by implementing the workflow in which dynamic instrumentation is done only once, and then the generated logs are used offline to browse various task decompositions. Currently, we are further tackling this overhead by porting the dynamic instrumentation from Valgrind to LLVM [10]. LLVM allows us more optimized dynamic instrumentation, as well as bypassing some part of instrumentation from run-time to compile-time. Our initial studies estimate that LLVM migration could accelerate the dynamic instrumentation by a factor of 5-10x. Also, in the part of offline decomposition exploration, we are studying various divide-and-conquer techniques. These techniques let us evaluate parallelism of smaller sections of execution (with lower number of task instances), and then combine these partial results to reconstruct the total execution.

On the other hand, applicability of our approach concerns analyzing codes that are **not parallelization friendly**. In some applications, parallelism cannot be exposed just by decomposing the application into loops and functions. In these cases, our approach first must identify memory objects whose access patterns impede automatic parallelization. The tool should also pinpoint the culprit code sections, and advice the programmer how to change problematic memory access patterns. Once the programmer changes the problematic access patterns in the sequential application, the application should become more parallelization friendly, and automatic parallelization should achieve better results.

7 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 [11] is a Valgrind-based tool that estimates the potential speed-up for Cilk programs. On the other hand, Kremlin [8] 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 [5] assists parallelization with Thread Building Blocks (TBB) [13]. Parallel Advisor provides timing profile that suggests to the programmer which loops should be parallelized. Critical Blue provides Prism [3], a tool to do "what-if" analysis that anticipates the potential benefits of parallelizing certain parts of the code. Vector Fabrics provides Pareon [7], 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.

8 Conclusions and Future Work

In this paper, we have proposed a technique to automate the exploration of parallelization strategies based on a task decomposition approach. We have presented an effective search algorithm that aims to find an efficient task decomposition of codes. We have defined a set of key metrics and heuristics that lead the iterative process of refining task decomposition in order to increase the parallelism of the code. The metrics collect information such as the length (duration) of the tasks, the dependencies among tasks and tasks' concurrency level. A cost function that takes into account these metrics has been proposed to guide the parallelization process. In our experiments, we demonstrate that our search algorithm is able to find task decompositions that provide enough parallelism in the application to fully utilize the target multicore processor architecture.

As future work, we have identified the need to include a new metric that evaluates the cost of expressing a decomposition using the syntax (and constraints) offered by the target parallel programming model (for example, traditional fork-join, dataflow, ...). The automatic search should be able to quantify how expressible (or viable) a decomposition could be and to use this information to guide the search. The result would be an efficient task decomposition that can be straightforwardly expressed using a specific programming model.

References

1. Benkner, S.: Vfc: The vienna fortran compiler. Scientific Programming 7(1), 67–81 (1999)
2. Bik, A., Girkar, M., Grey, P., Tian, X.: Efficient exploitation of parallelism on pentium iii and pentium 4 processor-based systems (2001)
3. Critical Blue. Prism, `http://www.criticalblue.com/` (active on June 27, 2013)
4. Blume, W., Doallo, R., Eigenmann, R., Grout, J., Hoeflinger, J., Lawrence, T., Lee, J., Padua, D.A., Paek, Y., Pottenger, W.M., Rauchwerger, L., Tu, P.: Parallel programming with polaris. IEEE Computer 29(12), 78–82 (1996)
5. Intel Corporation. Intel Parallel Advisor, `http://software.intel.com/en-us/intel-advisor-xe` (active on June 27, 2013)
6. 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 Processing Letters 21(2), 173–193 (2011)
7. Vector Fabrics. Pareon, `http://www.vectorfabrics.com/products` (active on June 27, 2013)
8. Garcia, S., Jeon, D., Louie, C.M., Taylor, M.B.: Kremlin: rethinking and rebooting gprof for the multicore age. In: PLDI, pp. 458–469 (2011)
9. Jost, G., Labarta, J., Gimenez, J.: Paramedir: A tool for programmable performance analysis. In: Bubak, M., van Albada, G.D., Sloot, P.M.A., Dongarra, J. (eds.) ICCS 2004. LNCS, vol. 3036, pp. 466–469. Springer, Heidelberg (2004)
10. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis and transformation, San Jose, CA, USA, pp. 75–88 (March 2004)
11. Mak, J., Faxén, K.-F., Janson, S., Mycroft, A.: Estimating and Exploiting Potential Parallelism by Source-Level Dependence Profiling. In: D'Ambra, P., Guarracino, M., Talia, D. (eds.) Euro-Par 2010, Part I. LNCS, vol. 6271, pp. 26–37. Springer, Heidelberg (2010)
12. Nethercote, N., Seward, J.: Valgrind, `http://valgrind.org/` (active on June 27, 2013)
13. Pheatt, C.: Intel threading building blocks. J. Comput. Sci. Coll. 23(4), 298–298 (2008)
14. Pillet, V., Labarta, J., Cortes, T., Girona, S.: PARAVER: A Tool to Visualize and Analyze Parallel Code. In: WoTUG-18 (1995)
15. Subotic, V., Ferrer, R., Sancho, J.C., Labarta, J., Valero, M.: Quantifying the Potential Task-Based Dataflow Parallelism in MPI Applications. In: Jeannot, E., Namyst, R., Roman, J. (eds.) Euro-Par 2011, Part I. LNCS, vol. 6852, pp. 39–51. Springer, Heidelberg (2011)
16. Wilson, R.P., French, R.S., Wilson, C.S., Amarasinghe, S.P., Anderson, J.-A.M., Tjiang, S.W.K., Liao, S.-W., Tseng, C.-W., Hall, M.W., Lam, M.S., Hennessy, J.L.: Suif: An infrastructure for research on parallelizing and optimizing compilers. SIGPLAN Notices 29(12), 31–37 (1994)
17. Zhang, X., Navabi, A., Jagannathan, S.: Alchemist: A transparent dependence distance profiling infrastructure. In: CGO 2009 (2009)