Automatic Identification and Annotation of Parallel Tasks in Structured Programs

Anonymous Author(s)

Abstract

This paper reports the design and implementation of a suit of static analyses and code generation techniques to annotate programs with OpenMP pragmas for task parallelism. These techniques deal with problems such as the discovery and use of program symbols to estimate the profit of tasks, bound their recursive depth and limit the memory regions onto which they operate. These techniques have been implemented into the first source-to-source compiler able to insert OpenMP pragmas into C/C++ programs without human intervention. By building onto the solid collection of static analyses available in LLVM, and relying on OpenMP's runtime ability to disambiguate pointers, we show that we can annotate large and convoluted programs, often replicating the performance gains of handmade annotation.

CCS Concepts •**Software and its engineering** → **Compilers;** *Procedures, functions and subroutines;* •**Theory of computation** → *Regular languages; Operational semantics;*

Keywords Parallelism, Tasks, OpenMP

ACM Reference format:
Anonymous Author(s). 2017. Automatic Identification and Annotation of Parallel Tasks in Structured Programs. In *Proceedings of IEEE, Planet Earh, 2018 (PL'17),* 6 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 Introduction

Annotation systems have risen to a place of prominence as a simple and effective means to write parallel programs. Examples of such systems include OpenMP (Jaeger et al. 2015), OpenACC (Standard 2013), OpenHMPP (Andión et al. 2016), OpenMPC (Lee and Eigenmann 2010), OpenSs (Meenderinck and Juurlink 2011), Cilk++ (Leiserson 2009) and Tareador (Ayguadé et al. 2015). Annotations work as a metalanguage: they let developers grant parallel semantics to syntax originally written to execute sequentially. Combined with modern hardware accelerators such as GPUs and FPGAs, they have led to substantial performance gains (Bertolli et al. 2014; Mendonça et al. 2017; Poesia et al. 2017; Reyes et al. 2012; Wienke et al. 2012). Nevertheless, although convenient, the use of annotations is not straightforward, and still lacks supporting tools that help programmers to check if annotations are correct and/or effective.

PL'17, Planet Earh
2017. 978-x-xxxx-xxxx-x/YY/MM…$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

There exist tools that insert automatic annotations in programs (Amini 2012; Guelton et al. 2012; Mendonça et al. 2016; Nugteren and Corporaal 2014; Pingali et al. 2011). All these technologies explore data-parallelism – the possibility of running the same computation independently on different data. Another form of parallelism, based on tasks, remains still uncharted land in what concerns automatic annotation. Such fact is unfortunate, since much of the power of current annotation systems lays on their ability to create tasks (Ayguadé et al. 2009). As we illustrate in Section 2.1, task parallelism – the power to run different routines simultaneously on independent data – brings annotation systems closer to irregular programs such as those that process graphs and worklists (Ben-Nun et al. 2017; Kulkarni et al. 2011; Pingali et al. 2011). The purpose of this work is to address this omission.

This paper describes TaskMiner, a source-to-source compiler that exposes task parallelism in C/C++ programs. To fulfil this goal, TaskMiner solves different challenges. First, it finds program regions, e.g., loops or functions, that can be effectively mapped onto tasks (Section 3.1). Second, it determines symbolic bounds to the memory blocks accessed within those regions (Section 3.2). Third, it extracts parameters from the code to estimate when it is profitable to create tasks. These parameters feed conditional checks, which, at runtime, enable or disable the creation of tasks (Section 3.3) and limit their recursion depth (Section 3.4). Fourth, TaskMiner determines which program variables need to be privatized in new tasks, or shared among them (Section 3.5). Finally, it maps all this information back into source code, producing readable annotations (Section 3.6).

In this paper, we defend the thesis that automatic task annotations are effective and useful. Our techniques enhance the productivity of developers, because they save them the time to annotate programs. TaskMiner receives as input C code, and produces, as output, a C program annotated with human-readable OpenMP task directives. We are currently able to annotate non-trivial programs, involving every sort of composite type available in the C language, e.g., arrays, structs, unions and pointers to these aggregates. Some of these programs, such as those taken from the Kastor (Virouleau et al. 2014) or Bots (Duran et al. 2009) benchmark suites, are large and complex; thus, their manual annotation is a time consuming and error prone task. Yet, our automatically annotated programs not only approximate the execution times of the parallel versions of those benchmarks, but are much faster than Guido: Numbers? their sequential –unannotated– versions, as we report in Section 4.

2 Advantages and Difficulties

The use of OpenMP pragmas as the concrete means to explore, without human intervention, task parallelism in programs presents difficulties and advantages. Such challenges and benefits have guided the design and implementation of the techniques described in Section 3. In the rest of this section we discuss such advantage and difficulties.

2.1 The Advantages

The main advantage of using OpenMP annotations to split a program into parallel tasks is the ability of leaving the job of handling dependences to a runtime. The OpenMP runtime maintains a task dependence graph which dynamically exposes more parallelism opportunities than those resulting from a conservative compile-time analysis. In particular, the runtime lets us circumvent the shortcomings that pointer aliasing impose on the automatic parallelization of code. Aliasing – the possibility of two pointers dereferencing the same memory region – either prevents parallelism altogether, or forces compilers to resort to complex runtime checks to ensure its correctness (Alves et al. 2015; Rus et al. 2002). The OpenMP runtime shields us from this problem, because it already checks for dependencies among tasks, and dispatches them in a correct order (LaGrone et al. 2011). To achieve that, OpenMP runtime uses task descriptors that contain Guido: The descriptor does not keep task status; this is done in the runtime. Removed: task's status (running, waiting) a task identifier and a list of memory addresses corresponding to the task input /output arguments.

The process of checking for dependencies can be as simple as checking if an incoming task accesses a memory address already in use by an in-flight task. It can also be complex, involving checks over long ranges of addresses (in case the runtime system supports dependence checking for memory ranges). This process can also include memory labelling and renaming (in case the runtime system is able to remove false dependencies). Regardless of its capacity – which is not the same across every OpenMP implementation – the runtime system will represent dependencies using a task dependence graph (TDG) (Duran et al. 2008b). In this directed acyclic graph, nodes denote tasks and edges represent dependences between them. Tasks are dispatched for execution according to a dynamic topological ordering of this graph (Planas et al. 2015).

Guido: Now it is clear the relevance for irregular programs. The OpenMP runtime support allows the parallelization of irregular applications, such as programs that traverse data-structures formed by a mesh of pointers. Guido: A litte bit more details: In such programs, control construcs (e.g. `if` statement) make the execution of certain pieces of code dependent on the program input. When using task parallelism the runtime can capture such dependences. This is not possible otherwise, since compile-time dependence analyzers must be conservative and eventually skip some parallelization opportunity.

```
1  struct LINE {char* line; size_t size;};
2  int str_contains_pattern(char* str, char* pattern);
3  char* copy_str(char* str);
4  void book_filter(struct LINE** bk_in,
5                   char* pattern, char** bk_out, int size) {
6    #pragma omp parallel
7    #pragma omp single
8    for (int i = 0; i < size; i++) {
9      #pragma omp task depend(in: line, pattern)
10     if (str_contains_pattern(bk_in[i]->line, pattern)) {
11       #pragma omp task depend(in: line)
12       bk_out[i] = copy_line(bk_in[i]->line);
13     }
14   }
15 }
```

Figure 1. The benefits of the OpenMP runtime environment. Neither the runtime checks of Alves (Alves et al. 2015) or Rus (Rus et al. 2002), nor Whaley's context and flow sensitive alias analysis would be able to ensure the correctness of the automatic parallelization of this program.

As an example, Figure 1 shows an application that finds patterns in lines of a book. The book is given as an array of pointers; each pointer leads to a string representing a –potentially– different line. The natural parallelization of this program consists in firing off a task to process each line. Figure 1 has been annotated automatically by Removed: the TaskMiner. The OpenMP execution environment ensures the correct scheduling of the tasks created at lines 9 and 11, by Guido: This is vague: Removed: runtime checks. assuring that the annotated input dependencies `line` and `pattern` are respected at runtime. Guido: Not clear. Need more details here. Adding: Pointer-based dependencies can address various memory regions. Hence, in order to enable the automatic synthesis of task annotations a system must be able to precisely define which memory region a pointer dependency is pointing to. Guido: Not clear why it is not important for correctness. Therefore, the ability to disambiguate memory regions statically is not essential for correctness, although it is important for efficiency reasons, as we will mention in Challenge 2.1.

2.2 The Difficulties

The automatic insertion of effective OpenMP annotations into C/C++ programs required us to deal with a number of challenges. If left unsolved, these challenges would restrict our interventions in programs to trivial annotations, which

would hardly be of any use. Our first challenge is inherent to any automatic parallelization system.

```
1  int foo(int* U, int* V, int N, int M) {
2    int i, j;
3    #pragma omp parallel
4    #pragma omp single
5    for(i = 0; i < N; i++) {
6      #pragma omp task depend(in: V[i*N:i*N+M]) \
7        if (5 * M < WORK_CUTOFF)
8      for (j = 0; j < M; j++) {
9        U[i] += V[i*N + j];
10     }
11   }
12   return 1;
   }
```

```
movl   (%rsi,%rax,4), %r11d
movslq %r9d, %rbx
addl   %r11d, (%rdi,%rbx,4)
incl   %r10d
incl   %eax
```

Figure 2. Challenges 2.1 and 2.2: identifying memory regions with symbolic limits, and using runtime information to estimate the profit of tasks.

Challenge 2.1. *Identify the memory region covered by a task.*

Figure 2 illustrates Challenge 2.1, and shows how we solve it[1]. That program receives an M × N matrix V, in linearized format, and produces a vector U, so that U[i] contains the sum of all the elements in line i of matrix V. For reasons to be considered in Section 3.1, our static analysis determines that each iteration of the outermost loops could be made into a task. Thus, tasks comprise the innermost loop, and traverse the memory region between addresses &V + i * N and &V + i * N + M. The identification of such ranges involves the use of a symbolic algebra, which we have borrowed from the compiler-related literature, as we explain in Section 3.2. Figure 2 also introduces the second challenge that we tackle:

Challenge 2.2. *Estimate the profitability of tasks at runtime.*

The creation of tasks involves a heavy runtime cost due to allocation, scheduling and real-time management of the task dependence graph. Ideally, this cost should be paid only for tasks that perform an amount of work sufficiently large to pay for their management. Being an interesting program property, on Rice's sense (Rice 1953), the amount of work performed by a task cannot be discovered statically. As we show in Section 3.3, we can try to approximate this quantity, using, to this end, program symbols, which are replaced with actual values at runtime. For instance, in Figure 2, we know that the body of the innermost loop is formed by five instructions. Thus, we approximate the amount of work performed by a task with the expression 5 * M. We use the runtime value of M to determine, during the execution of the program, if we create a task, or not. Such test is carried out by the guard at line 7 of the figure, which is part of OpenMP's syntax. Also, we provide a reliable estimate on the *workload cutoff* from which a task can be safely spawned without producing performance overhead. This *cutoff* considers factors such as number of available cores and runtime information on the task dispatch cost in terms of machine instructions. This is further explained in Section 3.3.

[1]Figures 2-3 shows programs that we have annotated without any human intervention. Statements in black font are part of the original program. Annotations appear in grey.

```
1  long long fib (int n) {
2    static int taskminer_depth_cutoff;
3    taskminer_depth_cutoff++;
4    long long x, y;
5    if (n < 2) return n;
6    #pragma omp task untied default(shared) \
7      if(taskminer_depth_cutoff < DEPTH_CUTOFF)
8    x = fib(n - 1);
9    #pragma omp task untied default(shared) \
10     if(taskminer_depth_cutoff < DEPTH_CUTOFF)
11   y = fib(n - 2);
12   #pragma omp taskwait
13   taskminer_depth_cutoff--;
14   return x + y;
15 }
```

Figure 3. Challenge 2.3: bounding the creation of recursive tasks with counter associated with function calls.

Challenge 2.3. *Bound the creation of recursive tasks.*

We introduce Challenge 2.3 by quoting Duran *et al.*: "*In task parallel languages, an important factor for achieving a good performance is the use of a cut-off technique to reduce the number of tasks created*" (Duran et al. 2008a). This observation is particularly true in the context of recursive, fine-grain, tasks, as we analyze in Section 3.4. Figure 3 provides an example. To place a limit on the number of tasks simultaneously in flight, we associate the invocation of recursive functions annotated with task pragmas with a counter – taskminer_depth_cutoff in Figure 3. The guard in line 7 ensures that we never exceed DEPTH_CUTOFF, a predetermined threshold. This example, together with Figure 2, lets us emphasize that the code generation algorithms presented in this paper are parameterized by constants such as DEPTH_CUTOFF, or WORK_CUTOFF in Figure 2. Although these have default estimates, we provide them as parameters to be set at will.

```
1  int sum_range(int* V, int N, int L, int* A) {
2    int i=0, sum=0;
3    #pragma omp parallel
4    #pragma omp single
5    while (i < N) {
6      int j = V[i];
7      A[i] = 0;
8      #pragma omp task default(shared) firstprivate(i,j)
9      for (; j < L; j++) {
10       A[i] += V[j];
11       j++;
12     }
13     i++;
14   }
15   return sum;
16 }
```

Figure 4. Challenge 2.4: variable j must be replicated among tasks, to avoid the occurrence of data races.

Challenge 2.4. *Identify private and shared variables.*

The two previous challenges are related to the performance of annotated programs: if left unsolved, we shall have correct, albeit inefficient programs. Challenges 2.1, and 2.4, in turn, are related to correctness. Challenge 2.4 asks for the identification of variables that must be replicated among threads. This process of replication is called *privatization*. As an example, variable j in Figure 4 must be privatized. In the absence of such action, that variable will be shared among all the tasks created at line 5 of Figure 4. Because j is written within these tasks, race conditions would ensue. Section 3.5 explains how we distinguish private from shared variables.

3 Solution

3.1 Mapping Program Regions to Tasks

The algorithm to find tasks is illustrated in Figure 5. It uses information of two abstract program structures: the *Control Flow Graph (CFG)* and the *Program Dependence Graph (PDG)*. The key intuition behind finding potential tasks relies on a concept that we call *Helices*. A *Helix* is a set of nodes in the PDG that depend on a Strong Connected Component (SCC) but *do not integrate the SCC*. Strong Connected Components in the Dependence Graph represent statically detected iteration dependence, so it arises with loops or with recursive patterns (when the Graph is interprocedural). **??** shows what represents a *Helix*. The insight behind *Helices* is that they represent instructions that are deemed to repeat more than once

```
1  def minetasks(PROG):
2    CFG = ControlFlowGraph(PROG)
3    PDG = ProgramDependenceGraph(PROG)
4    HELICES = findHelices(PDG)
5    TASKREGIONS = []
6    for h in HELICES:
7      r = CFG.findMinimalCoveringRegion(h)
8      TASKREGIONS.append(r)
9    for r in TASKREGIONS:
10     r.expand()
11   NEW_PROG = annotate(PROG, TASKREGIONS)
12   return NEW_PROG
```

Figure 5. The main steps of our code generator.

```
1  def expand(REGION):
2    CanExpand = True
3    while (CanExpand):
4      PARENTREGION = REGION.getParent()
5      REGION.basicblocks.add(PARENTREGION)
6      REGION.resolveDependencies()
7      CanExpand = checkExpansion(REGION)
```

Figure 6. Discovery of tasks via expansion of hammock regions.

and, since they do not have a cycle, they can be executed in parallel.

Therefore, the first step in the algorithm to find tasks is to map these *Helices* into program regions. The algorithm goes through the Dependence Graph looking for *Helices* and then finds the minimal covering region for each *Helix*. These minimal regions are later going to be annotated as tasks in the original program, should they fit a cost model.

After finding potential task regions, next is the *expansion* step. We expand the region to a larger one as long as two requisites are met:

1. Input and output dependencies are never increased;
2. The potential parallelism is never reduced.

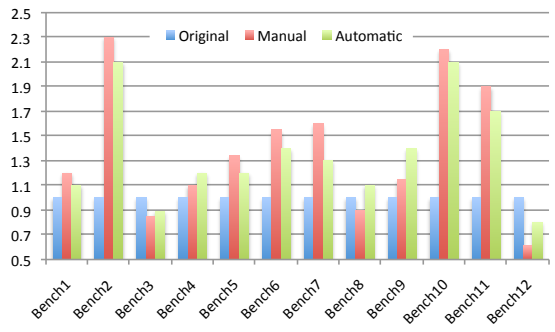

Figure 7. Speedup comparisons – automatic annotations compare favourably against manual interventions, and lead to great performance gains. The Y-axis shows the speedup of either manual intervention, or automatic annotation (this paper) onto the original programs.

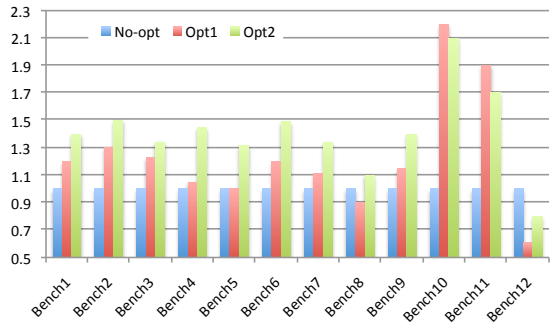

Figure 8. The benefit of our optimizations. Each optimization (profitability analysis and recursion bounds) lead to non-trivial speedups.

3.2 Finding Symbolic Bounds of Arrays

3.3 Estimating the Profitability of Tasks

3.4 Limiting the Creation of Recursive Tasks

3.5 Separating Private from Shared Variables

3.6 Mapping IR onto Source Code

4 Evaluation

We performed a thorough evaluation on TaskMiner's applicability in real programs. Our test framework is divided in five aspects:

- Performance
- Optimizations
- Versatility
- Applicability
- Practicality

Performance.
Optimizations.
Versatility.
Versatility.
Practicality.

5 Related Work

6 Conclusion

References

Péricles Alves, Fabian Gruber, Johannes Doerfert, Alexandros Lamprineas, Tobias Grosser, Fabrice Rastello, and Fernando Magno Quintão Pereira. 2015. Runtime Pointer Disambiguation. In *OOPSLA*. ACM, New York, NY, USA, 589–606.

Mehdi Amini. 2012. *Source-to-Source Automatic Program Transformations for GPU-like Hardware Accelerators. (Transformations de programme automatiques et source-à-source pour accélérateurs matériels de type GPU).* Ph.D. Dissertation. Mines ParisTech.

José M. Andión, Manuel Arenaz, François Bodin, Gabriel Rodríguez, and Juan Touri no. 2016. Locality-Aware Automatic Parallelization for GPGPU with OpenHMPP Directives. *Inter. Journal of Parallel Programming* 44, 3 (2016), 620–643.

Eduard Ayguadé, Rosa M. Badia, Daniel Jiménez, José R. Herrero, Jesús Labarta, Vladimir Subotic, and Gladys Utrera. 2015. Tareador: A Tool to Unveil Parallelization Strategies at Undergraduate Level. In *WCAE*. ACM, New York, NY, USA, 1:1–1:8.

Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. 2009. The Design of OpenMP Tasks. *Trans. Parallel Distrib. Syst.* 20, 3 (2009), 404–418.

Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. 2017. Groute: An Asynchronous Multi-GPU Programming Model for Irregular Computations. In *PPoPP*. ACM, New York, NY, USA, 235–248.

Carlo Bertolli, Samuel F. Antao, Alexandre E. Eichenberger, Kevin O'Brien, Zehra Sura, Arpith C. Jacob, Tong Chen, and Olivier Sallenave. 2014. Coordinating GPU Threads for OpenMP 4.0 in LLVM. In *LLVM-HPC*. IEEE, New York, NY, USA, 12–21.

Alejandro Duran, Julita Corbalán, and Eduard Ayguadé. 2008a. An Adaptive Cut-off for Task Parallelism. In *SC*. IEEE, Piscataway, NJ, USA, 36:1–36:11.

Alejandro Duran, Josep M. Perez, Eduard Ayguadé, Rosa M. Badia, and Jesus Labarta. 2008b. Extending the OpenMP Tasking Model to Allow Dependent Tasks. In *IWOMP*. Springer, Heidelberg, Germany, 111–122.

Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguade. 2009. Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. In *ICPP*. IEEE, Washington, DC, USA, 124–131.

Serge Guelton, Mehdi Amini, and Béatrice Creusillet. 2012. Beyond Do Loops: Data Transfer Generation with Convex Array Regions. In *LCPC*. Springer, Heidelberg, Germany, 249–263.

Julien Jaeger, Patrick Carribault, and Marc Pérache. 2015. Fine-grain Data Management Directory for OpenMP 4.0 and OpenACC. *Concurr. Comput. : Pract. Exper.* 27, 6 (2015), 1528–1539.

Milind Kulkarni, Donald Nguyen, Dimitrios Prountzos, Xin Sui, and Keshav Pingali. 2011. Exploiting the Commutativity Lattice. In *PLDI*. ACM, New York, NY, USA, 542–555.

James LaGrone, Ayodunni Aribuki, Cody Addison, and Barbara Chapman. 2011. A Runtime Implementation of OpenMP Tasks. In *IWOMP*. Springer, Heidelberg, Germany, 165–178.

S. Lee and R. Eigenmann. 2010. OpenMPC: Extended OpenMP Programming and Tuning for GPUs. In *SC*. IEEE, Washington, DC, USA, 1–11.

Charles E. Leiserson. 2009. The Cilk++ Concurrency Platform. In *DAC*. ACM, New York, NY, USA, 522–527.

Cor Meenderinck and Ben Juurlink. 2011. Nexus: Hardware Support for Task-Based Programming. In *DSD*. Springer, New York, NY, USA, 442–445.

Gleison Souza Diniz Mendonça, Breno Campos Ferreira Guimarães, Péricles Rafael Oliveira Alves, Fernando Magno Quintão Pereira, Márcio Machado Pereira, and Guido Araújo. 2016. Automatic Insertion of Copy Annotation in Data-Parallel Programs. In *SBAC-PAD*. IEEE, New York, 34–41.

Gleison Mendonça, Breno Guimarães, Péricles Alves, Márcio Pereira, Guido Araújo, and Fernando Magno Quintão Pereira. 2017. DawnCC: Automatic Annotation for Data Parallelism and Offloading. *ACM Trans. Archit. Code Optim.* 14, 2 (2017), 13:1–13:25.

Cedric Nugteren and Henk Corporaal. 2014. Bones: An Automatic Skeleton-Based C-to-CUDA Compiler for GPUs. *TACO* 11, 4 (2014), 35:1–35:25.

Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. 2011. The Tao of Parallelism in Algorithms. In *PLDI*. ACM, New York, NY, USA, 12–25.

Judit Planas, Rosa M. Badia, Eduard Ayguadé, and Jesús Labarta. 2015. SSMART: Smart Scheduling of Multi-architecture Tasks on Heterogeneous Systems. In *WACCPD*. ACM, New York, NY, USA, 1:1–1:11.

Gabriel Poesia, Breno Guimarães, Fabrício Ferracioli, and Fernando Magno Quintão Pereira. 2017. Static Placement of Computation on Heterogeneous Devices. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 50:1–50:28.

R. Reyes, I. López-Rodríguez, J. Fumero, and F. Sande. 2012. AccULL: An OpenACC Implementation with CUDA and OpenCL Support. In *Euro-Par*. Springer, New York, NY, USA, 871–882.

H. G. Rice. 1953. Classes of Recursively Enumerable Sets and Their Decision Problems. *Trans. Amer. Math. Soc.* 74, 2 (1953), 358–366.

Silvius Rus, Lawrence Rauchwerger, and Jay Hoeflinger. 2002. Hybrid Analysis: Static and Dynamic Memory Reference Analysis. In *ICS*. IEEE, Piscataway, NJ, USA, 251–283.

OpenACC Standard. 2013. *The OpenACC Programming Interface*. Technical Report. CAPs.

Philippe Virouleau, Pierrick BRUNET, François Broquedis, Nathalie Furmento, Samuel Thibault, Olivier Aumage, and Thierry Gautier. 2014. Evaluation of OpenMP Dependent Tasks with the KASTORS Benchmark Suite. In *IWOMP*. Springer, Heidelberg, Germany, 16 – 29.

Sandra Wienke, Paul L. Springer, Christian Terboven, and Dieter an Mey. 2012. OpenACC - First Experiences with Real-World Applications. In *Euro-Par*. Springer, New York, NY, USA, 859–870.