Automatic Annotation of Tasks in Structured Code

Pedro Ramos, Gleison Souza
UFMG, Brazil
pedroramos@dcc.ufmg.br
gleison.mendonca@dcc.ufmg.br

Divino Soares, Guido Araújo
UNICAMP, Brazil
divcesar@gmail.com
guido@ic.unicamp.br

Fernando Magno Quintão Pereira*
UFMG, Brazil
fernando@dcc.ufmg.br

Abstract
This paper describes 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 approximate the ranges covered by memory regions, bound recursive tasks and estimate the profitability of tasks. We have used these ideas to implement a source-to-source compiler that inserts OpenMP pragmas into C/C++ programs without any human intervention. By building onto the static program analysis literature, 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. Furthermore, our techniques give us the means to discover opportunities of parallelism that remained buried in the syntax of well-known benchmarks for many years—sometimes leading to up to four-fold speedups on a 12-core machine at zero programming cost.

CCS Concepts → Computing methodologies → Parallel programming languages; Concurrent programming languages; Software and its engineering → Compilers;

Keywords Parallelism, Tasks, OpenMP

ACM Reference format:
DOI: https://doi.org/10.1145/3243176.3243200

1 Introduction
Annotation systems have risen to a place of prominence as a simple and effective way to write parallel programs. Examples of such systems include OpenMP [30], OpenACC [53], OpenHMPP [3], OpenMPC [35], OpenSs [37] and OmpSs [10, 19]. Annotations let developers grant parallel semantics to syntax originally written to execute sequentially. Combined with hardware accelerators such as GPUs and FPGAs, they have led to substantial performance gains [8, 40, 48]. Nevertheless, although convenient, the use of annotations is not straightforward. We still lack tools that help programmers to check if annotations are correct and/or effective. There exist tools that insert automatic annotations in programs [39, 45]. All these technologies explore data-parallelism—the possibility of running the same computation independently on different data. Task parallelism remains still uncharted land in what concerns automatic annotation. Such fact is unfortunate, since much of the power of current annotation systems lies in their ability to create tasks [5]. 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 [45]. The purpose of this work is to address this omission.

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. In Section 4, we show that we have been able to achieve almost four-fold speedups onto standard benchmarks (not-used in parallel programming), with zero programming cost. 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 [58], Bots [22] or the LLVM test suites, are large and complex. Yet, our automatically annotated programs not only approximate the execution times of the parallel versions of those benchmarks, but are in general faster than their sequential—unannotated—versions, as we report in Section 4.
2 Challenges

This paper presents a tool that annotates source C code with OpenMP task directives. This tool uses novel techniques to fulfill its purpose. Such techniques solve four challenges, which Section 2.2 illustrates with examples. However, before we dive into these challenges, the reader must understand the main benefits of using a runtime environment such as OpenMP’s, for such advantages have guided most of our design decisions. Section 2.1 introduces this discussion.

2.1 The OpenMP Runtime System

OpenMP task annotations let developers leave to the runtime the job of handling data dependences. The OpenMP runtime maintains a 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 [2, 49]. The OpenMP runtime shields us from this problem, because it already checks for dependencies among tasks, and dispatches them in a correct order [33].

Dependence tracking can be complex, involving checks over ranges of addresses (if the runtime system treats dependences across memory ranges). It can also include memory labelling and renaming (if the runtime system is able to remove false dependences). 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) [21]. 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 [47].

OpenMP’s runtime support allows the parallelization of irregular applications, such as programs that traverse data-structures formed by a mesh of pointers. In such programs, control constructs like if statements make the execution of some statements dependent on the program’s input. The runtime can capture such dependences, in contrast to static analysis tools. As an example, Figure 1 shows an application that finds patterns in lines of a book1. The book is given as an array of pointers. Each pointer leads to a string representing a potentially different line. Our parallelization of this program consists in firing off a task to process each line. Figure 1 has been annotated by TaskMiner. The OpenMP execution environment ensures the correct scheduling of the tasks created at lines 10 and 12, by reassuring that the annotated dependencies line and pattern are respected at runtime.

Figure 1. The benefits of the OpenMP runtime environment. Neither the runtime checks of Alves [2] or Rus [52], nor Whaley’s context and flow sensitive alias analysis [59] would be able to ensure the correctness of the automatic parallelization of this program.

2.2 Challenges

As seen in Section 2.1, the OpenMP runtime liberates us from the burden of having to track dependences between pointers statically. However, the automatic insertion of effective task annotations into programs still required us to deal with a number of challenges. If left unsolved, these challenges would restrict our interventions to trivial annotations, which would hardly be of any use. Our first challenge is inherent to any automatic parallelization system.

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

Figure 2. Challenges 2.1 and 2.2: identifying memory regions with symbolic limits, and using runtime information to estimate the profit of tasks.
Figure 2 illustrates Challenge 2.1, and shows how we solve it. That program receives an $M \times N$ 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 3.2, 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 3.1. 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 dependence graph (see Sec. 2.1). 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 [50], 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 $5 \times M$. 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.

**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" [20]. This observation is particularly true in the context of recursive, fine-grained tasks, as we analyze in Section 3.6. 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 – 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.

**Figure 3.** Challenge 2.3: bounding the creation of recursive tasks. Example taken from [28, Fig.1].

```c
void sum_range(int* V, int N, int L, int* A) {
  int i=0;
  #pragma omp parallel
  #pragma omp single
  while (i < N) {
    int j = V[i];
    A[i] = 0;
    #pragma omp task default(shared) firstprivate(i,j)
    for (; j < L; j++) { A[i] += V[j]; } // A[i] += V[j];
  }
  return (i + j);
}
```

**Challenge 2.4.** *Identify private and shared variables.*

The two previous challenges are related to the performance of annotated code: 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 8 of Figure 4. Because $j$ is written to within these tasks, race conditions would ensue. Section 3.5 explains how we distinguish private from shared variables.

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

```c
# pragma omp critical
j = fib_recursive(n - 2);
```

```c
cutoff--;
#pragma omp taskwait
return (i + j);
```
3 Solutions

Figure 5 provides a high-level view of a source-to-source compiler that incorporates the techniques discussed in this paper. The many parts of this algorithm shall be explained throughout this section. This pseudo-code uses several concepts well-known in the compilers literature, such as control flow graphs [32] and dependence graphs [24]. In the rest of this section we explain how we have combined this previous knowledge to delimit and annotate tasks in structured programs. Our presentation focuses on the new elements that we had to add onto known techniques, in order to adapt them to our purposes.

Tasks: syntax and scope: Our goal is to identify tasks in structured programs. In the context of this work, a structured program is a program that can be partitioned in hammock regions, a concept introduced by Ferrante et al. [24] in the mid-1980’s. For completeness, we re-state it in Definition 3.1. Definition 3.2 formalizes the notion of task.

Definition 3.1 (Hammock Region [24]). A Control Flow Graph \( G \) is a directed graph with a starting node \( s \), and an exit node \( x \). A hammock region \( G’ \) is a subgraph of \( G \) with a node \( h \) that dominates\(^2\) the other nodes in \( G’ \). Additionally, there exists a node \( w \in G, w \notin G’ \), such that \( w \) post-dominates every node in \( G’ \). In this definition, \( h \) is the entry point, and \( w \) is the exit point of \( G’ \).

Definition 3.2 (Task). Given a program \( P \), a task \( T \) is a tuple \((G’, M_1, M_2)\) formed by a hammock region \( G’ \), plus a set \( M_1 \) of memory regions representing data that \( T \) reads, and a set \( M_2 \) of memory regions representing data that \( T \) writes.

Definition 3.2 uses the concept of memory region. Syntactically, memory regions are described by program variables

\(^2\)A node \( n_1 \in G \) dominates another node \( n_2 \in G \) if every path from \( S \) to \( n_2 \) goes across \( n_1 \). Inversely, \( n_1 \) post-dominates \( n_2 \) if every path from \( n_2 \) to \( x \) must go across \( n_1 \).

```python
1 def minetasks(Prog, CostModel):
2     CFG = ControlFlowGraph(Prog)
3     PDG = ProgramDependenceGraph(Prog)
4     Ranges = symbolicRangeAnalysis(CFG)
5     Vanes = findVanes(PDG)
6     Costs = profitabilityAnalysis(Tasks, Ranges, CostModel)
7     Tasks = []
8     for v in Vanes:
9         region = v.expand(Costs)
10        Tasks.append(region)
11        Privs = privatizationAnalysis(Tasks, PDG)
12        Prog.annotate(Tasks, Costs, Ranges, Privs)
```

Figure 5. The main steps of our code generator. Grey boxes show sections in this paper where each phase is discussed.

and/or pointers plus ranges of dereferenceable offsets. Given two tasks: \( T_1 = (G_1, M_{11}, M_{12}) \) and \( T_2 = (G_2, M_{21}, M_{22}) \), if \( M_{11} \cap M_{21} \neq \emptyset \), then we say that \( T_2 \) depends on \( T_1 \). If a program \( P \) is partitioned into a set \( T \) of \( n \) tasks, then this partition is said to be correct if: (i) \( T \) does not contain cyclic dependence relations; and (ii) the execution of the tasks in \( T \) in any ordering determined by dependence relations leads to the same results as the sequential execution of \( P \).

Example 3.3 (Memory Regions). Below we have two tasks, \( T_{foo} = (\text{foo}, [v[i-1], v[i]]) \) and \( T_{bar} = (\text{bar}, [v[i]], []) \). Dependencies are identified via the depend clause. In this example, each hammock region is formed by one function call:

```python
#pragma omp task depend(in: v[i-1]) depend(out: v[i])
v[i] = foo(&v[i-1], i);
#pragma omp task depend(in: v[i])
bar(&v[i], i);
```

The annotation zoo. Our equipment of choice to exploit task parallelism in programs is OpenMP 4.0. We describe below which annotations we are using, and explain, informally, their semantics. Full overviews of the syntax and semantics of these annotations are publicly available\(^3\); hence, we shall not dive into their details.

- parallel (clauses): forms a team of threads which will execute the marked program region in parallel.
- single (clauses): specifies that a program region must be executed by a single thread in the team.
- task (clauses): creates a new task.
- taskwait: defines a synchronization point where threads must wait for the completion of child tasks.

Some pragmas are associated with lists of clauses. There are several of these clauses available in OpenMP 4.0; however, we only use the following:

- default(shared): indicates that variables are shared by default among tasks.
- firstprivate(v): indicates that \( v \) must be replicated among tasks, and initialized with its value at the point where the annotation is executed.
- untied: if a task has this modifier, then its associated code can be executed by more than one thread; e.g., threads might alternate execution due to preemption and load balancing.
- depend(in|out|inout): determine if data is read (in), written (out) or both within a task region; thus, effectively setting dependences among tasks.
- if(condition): this clause leads to the creation of an undefined task, which must be executed immediately. We use it to enforce the cost model.
- final(condition) mergeable: this clause limits the creation of tasks, and reuses the data environment of the father task. We use it prune tasks considered too small, as discussed in Section 3.6.

\(^3\)For a quick overview, we refer the reader to the leaflet “Summary of OpenMP 4.0 C/C++ Syntax”, which is made available by the OpenMP ARB.
3.1 Finding Symbolic Bounds of Arrays

To produce the annotations that create a task \( T = (G, M_i, M_o) \), we need to determine the memory regions \( M_i \) that \( T \) reads, and the memory regions \( M_o \) that it writes. Memory regions consist of pointers, plus ranges of addresses that these pointers can dereference. Example 3.4 illustrates this notion. To determine precise bounds to memory regions, we resort to an old ally of compiler writers: the Symbolic Range Analysis, a concept that Definition 3.5 formalizes.

Example 3.4 (Memory Region). The statement \( U[i] += V[i*N + j] \), at line 9 of Figure 2 contains two memory accesses: \( U[i] \) and \( V[i*N + j] \). The first covers the memory region \([&U, &U + (M - 1) \times \text{sizeof}(\text{int})] \). The other covers the region \([&V+N*\text{sizeof}(\text{int}), &V+((i*M+j)-1)*\text{sizeof}(\text{int})]) \). There is no code in Figure 2 that depends on the loop in lines 8-10. Thus, a task annotation must only account for the input dependence, e.g., the access on \( V \). That is why the depend clause in line 6 contains a reference to this region.

Definition 3.5 (Symbolic Range Analysis). This is a form of abstract interpretation that associates an integer variable \( v \) with a Symbolic Interval \( R(v) = [l, u] \), where \( l \) and \( u \) are Symbolic Expressions. A symbolic expression \( E \) is recursively defined as one of the terms: \( s, c, E+E, E*E, \min(E, E), \max(E, E), -\infty, +\infty \), where \( s \) is a program symbol, and \( c \in \mathbb{Z} \).

The program symbols mentioned in Definition 3.5 are names that cannot be reconstructed as functions of other names. Examples include global variables, function arguments, and values returned by external functions. Symbolic range analysis is not a contribution of this paper: several implementations of it exist. We have adopted the one in the DawnCC compiler [40], which, itself, reuses an implementation by Nazaré et al. [42]. The only extensions that we have added into DawnCC’s implementation was the ability to handle C-like structs. Therefore, we shall not provide further details about this part of our work. To understand the rest of this paper, it suffices to know that this implementation is sufficiently solid to handle the entire C99 language, always terminates and runs in time linear on the size of the program’s dependence graph. Our implementation of Symbolic Range Analysis is interprocedural, to handle recursive functions, for instance. Thus, in the worst case, it is quadratic on the number of program variables. If a program contains an array access \( V[E] \), and we have that \( R(E) = [l, u] \), then the memory region covered by this access is, at least, \([&V+l, &V+u]\). The example below clarifies this observation.

Example 3.6. Symbolic range analysis, when applied onto Figure 2, gives us \( R(i) = [0, N - 1] \), and \( R(j) = [0, M - 1] \). The memory access \( V[i*N + j] \) at line 9, when combined with this information, yields the symbolic region that appears in the task annotation at line 6 of Figure 2.

3.2 Mapping Program Regions to Tasks

A task candidate is a set of program statements that can run in parallel with the rest of the program. To identify task candidates, we rely on the program’s Dependence Graph. For completeness, Definition 3.7 revisits this notion.

Definition 3.7 (Program Dependence Graph (PDG) [24]). The PDG of a program \( P \) contains one vertex for each statement \( s \in P \). There exists an edge from \( s_1 \) to \( s_2 \) if the latter depends on the former. Statement \( s_2 \) is data-dependent on \( s_1 \) if it reads data that \( s_1 \) writes. It is control-dependent on \( s_1 \) if \( s_1 \) controls a branch, whose result determines if \( s_2 \) executes.

Task candidates are vanes in windmills. Windmills are a family of graphs. The term was adopted by Rideau et al. [51] to describe structural relations between register copies. Our windmills exist as subgraphs of the program’s dependence graph. We use a slightly more general definition than Rideaus’s; however, the metaphor that gave origin to the name: the shape of the graphs, is still unmistakable:

Definition 3.8 (Windmill – [51]). A windmill is a graph \( G_{vw} = G_c \cup G_{c1} \cup \ldots \cup G_{cn} \) formed by a strongly connected component \( G_c \) (its center), plus \( n \) components, not necessarily strong, \( G_{c1}, \ldots, G_{cn} \), the vanes, such that:

1. For any topological ordering of \( G_{vw} \), and nodes \( n_c \in G_c \), \( n_{c1}, \ldots, n_{cn} \in G_{c1}, \ldots, G_{cn} \), we have that \( n_c \) ahead of \( n_{c1} \), \ldots, \( n_{cn} \).
2. For any \( i \) and \( j \), \( 1 \leq i < j \leq n \), \( G_{ci} \) and \( G_{cj} \) do not share vertices (thus, sharing edges is also impossible).

Figure 6. Examples of program dependence graphs. Nodes that form centers of windmills are colored in gray. Data-dependence edges are solid; control-dependence edges are dashed. Vanes of windmills appear within dotted boxes. (a) PDG for Figure 1; (b) PDG for Figure 2.

Figure 6 shows two program dependence graphs, highlighting windmills and their vanes. The structure of windmills naturally leads to task candidates. Vanes correspond to program parts that are likely to be executed several times, because they sprout from a loop – the center of the windmill. Two different executions of the same vane can run in parallel. This is a property of the PDG, and one of the original motivations behind its design. These two different executions can
be seen as two instances of the same structure (the vane), coming out of the same windmill center. A topological ordering of the dependence graph does not impose any order among these two hypothetical replicas of the same structure. Tasks can then be synchronized at the post-dominator of vanes marked to run in parallel via the taskwait pragma.

Windmills give us structural properties to identify tasks. We can find them via a depth-first search traversal of the program’s dependence graph. Function findVanes, in Figure 5 implements this search. However, not every windmill leads us to profitable tasks. Furthermore, some windmills cannot be annotated, due to the inability of range analysis to find bounds to the memory they access. We deal with these shortcomings in the next two sections.

3.3 Estimating the Profitability of Tasks
The benefit of creating tasks depends on two factors: runtime cost, and parallelism. This tension leads to a sweet spot in the size of tasks, where size is measured as the amount of code they execute. The smaller the tasks are, the more parallel they tend to be, as a consequence of less dependencies. However, if tasks are too small, then the performance gained by the extra parallelism might not compensate the cost of managing them. On the other hand, if tasks are too big, then we might not obtain enough parallelism, because individual tasks still execute sequentially. In this paper, we mark as tasks the first maximal analyzable vane above the runtime cost. To imbue this last statement with meaning, in what follows, we define more formally the notions of task size. We leave the concept of maximal analyzable vane for Section 3.4.

Estimating the Size of Tasks. The size of a task is given by its workload, i.e., the number of instructions that such task executes. The same program region might lead to the creation of tasks with different sizes. Therefore, the actual size of a task is only known after it finishes execution. However, to judge if the creation of a task is worth its cost, we must be able to approximate its size statically. With this goal, we define the notion of Static Workload Estimate.

Definition 3.9 (Static Workload Estimate (SWE)). Let \( G = G_1 \cup G_2 \cup \ldots \cup G_n \) be a partition of a program’s control flow graph \( G \) into \( n \) disjoint hammock regions. We define the static workload estimate \( W(G) \) as a non-negative real number, such that \( W(G) = W(G_1) + W(G_2) + \ldots + W(G_n) \).

Definition 3.9 is too unconstrained: a function \( W(G) = 0 \) for any \( G \) satisfies it. However, we want \( W \) to approximate the dynamic behavior of programs. The literature contains heuristics to implement \( W \). Perhaps, the best-known among these heuristics is the static profiler of Wu and Larus [60]. In this paper, we adopted a different approach: we reuse the symbolic range analysis of Section 3.1 to augment static data with dynamic information. In other words, we use symbolic range analysis to construct expressions that represent the number of iterations of loops. A similar approach has already been used to decide when to migrate virtual memory pages in NUMA architectures [44]. OpenMP 4.0 contains syntax to enable the conditional creation of tasks. Our symbolic analysis lets us build predicates for such conditionals.

Example 3.10. The condition in line 7 of Fig. 2 determines that tasks are created if \( 5 \times M < WORK \_CUTOFF \). \( M \) is a program symbol, i.e., a variable passed as argument to function foo. The expression \( 5 \times M \) is an approximation for the size of the task that comprises the loop at lines 8-10. The constant \( WORK \_CUTOFF \) represents the cost of creating a thread in the OpenMP runtime, and it is determined empirically.

The declarative rules in Figure 7 sketch the heuristics that we use to compute the SWE for a hammock region \( S \). Rule LoopInv applies on loops whose number of iterations we cannot bound via some symbolic expression computed statically. We assume that loops execute 10 times, following previous statistics [60]. Rule LoopExp represents loops that we can analyze using our Symbolic Range Analysis. The auxiliary function \( \text{Iter}(S) \) returns a symbolic expression that represents the range of values covered by the code \( S \) that controls the number of iterations of the loop. In Figure 5, these rules are implemented by function profitabilityAnalysis.

The Cost Model. A static workload estimate is only meaningful in the context of a cost model. A cost model is a collection of parameters that determine the impact of the underlying computer architecture onto the creation and management of tasks. The literature contains examples of techniques used to build cost models, be it analytically [6], or empirically [48]. The automatic construction of a cost model is not the goal of this paper. Instead, the reader shall notice that Figure 5 receives a cost model as a parameter. For the experiments in Section 4, we have determined a small collection of constants related to the creation of threads in our target architecture. As an example, the value of \( WORK \_CUTOFF \), seen in Example 3.10, in our setting, is roughly 500 cycles.
3.4 Task Expansion

Task expansion consists in finding the smallest task that is large enough to pay for the thread creation cost. Figure 8 shows the algorithm that performs this activity. We start this algorithm by setting REGION to be a hammock graph \( H \) that corresponds to a vane within a windmill \( W \). The hammock decomposition of a structured program forms a tree [24], such that vertices in this tree represent hammock graphs. \( H_1 \) is a child of \( H_2 \) in this tree if two conditions apply: (i) \( H_1 \subset H_2 \), and (ii) for any other node \( H_x \), if \( H_1 \subset H_x \) then \( H_2 \subset H_x \). In this context, we call \( H_2 \) the parent of \( H_1 \).

```
1 def expand(VANE, COST):
2     CanExpand = True
3     while (CanExpand and SWE(VANE) < COST):
4         PARENTREGION = VANE.getParent()
5         VANE.basicblocks.add(PARENTREGION)
6         VANE.resolveDependencies()
7         CanExpand = checkExpansion(VANE)
```

**Figure 8.** Task discovery via expansion of hammock regions. COST is the overhead of creating and scheduling threads.

The routine checkExpansion sets the boolean CanExpand in Figure 8. This function is true as long as the following conditions apply: (i) VANE has the structural properties of a windmill’s vane, i.e., it is contained within a windmill \( W \); (ii) we can analyze the memory regions within VANE using the symbolic range analysis of Section 3.1; (iii) VANE does not depend on another vane inside the same windmill \( W \).

```
1 for (i = 0; i < N; i++) {
2     for (j = 1; j < M-1; j++) {
3         if (V[i] = U[j]) {
4             V[i] = U[j];
5         }
6         U[i] = 0;
7         }
```

**Figure 9.** (a) Example of doubly nested loop that shall be analyzed by the TaskMiner. (b) The dependence graph of the program. (c) The decomposition of this loop into windmills and vanes. \( D \) is the set of variables the vane depends on.

The program in Figure 9 (a), a slight variation of the code seen in Figure 2, contains two windmills. The first consists of the for loop at line 1. The second, nested in the first, is the for loop at line 2. The vanes that sprout from these windmills are highlighted in the program dependence graph in Figure 9 (b).

**Rational:** We expand tasks up to the COST threshold, which is determined by the cost model seen in Section 3.3. If the profitability of a task is smaller than this constant, then parallelism might not pay off the cost of managing threads. However, if we expand tasks too much, then we might lose parallelism, as code in a task runs sequentially. Thus, determining COST is essential to the performance of our optimization. In this work, we have set this parameter empirically for the runtime environment to be described in Section 4.

3.5 Privatization

TaskMiner bestows a parallel semantics on code that has been originally conceived to run sequentially. This semantic gap might lead to race conditions. Race conditions happen when annotations cause two or more threads to update values that, in the sequential program, are denoted by the same name. To avoid such situations, the insertion of annotations requires us to replicate some values, making them private to each task. We have named such replication privatization.

**Example 3.11** (Privatization). Variables \( i \) and \( j \) in Figure 4 need to be replicated among the tasks that the annotation at line 8 creates. The need to replicate \( j \) is more apparent: this variable is updated in the body of the task pragma, at line 9. In the absence of replication, we have a race condition. Variable \( i \) must also be replicated. Each iteration of the loop at line 9 reads a different value of \( j \). Thus, each task should receive a different value in this variable. In the absence of replication, all the threads would read the same value; hence, parallelization would change the semantics of the program.

**Definition 3.12** (The Private Requirement). If a variable name has scalar type and contains at least one def-use chain that enters the frontier of a task, then this name is said to bear the private requirement. The frontier of a task is determined by the expand function seen in Figure 8.

**Example 3.13.** Variable \( i \) must be privatized because it is defined at lines 2 and 10 of Figure 4, and is used at line 9 – the body of a new task. Moreover, this variable has a scalar type, e.g., int. Similarly, variable \( j \) is defined at line 6, and is used at line 9. This property – the existence of def-use chains entering the task region – leads to the insertion of the first private clause at line 8 of Figure 4.
firstprivate vs depend: according to Definition 3.12, only names representing scalar values are privatized. These names have value semantics, i.e., they are “passed by copy” if used as actual arguments of functions. Memory regions, that is to say, regions whose access happens through pointers, are not privatized. These regions are, instead, marked as dependences of the task, through the depend clause. Memory regions are discovered via the symbolic region analysis of Section 3.1. In addition to pointers, we do not privatize values defined within a task region, and used outside it. That is to say: we privatize incoming def-use chains, but do not privatize outgoing chains. Values having this "outgoing chain property" are shared by default. We indicate this semantics via the default(shared) pragmas, as seen in line 8 of Figure 4.

3.6 Mapping IR onto Source Code

The analyses described in Sections 3.1-3.5 have been implemented at the level of the LLVM intermediate representation, and can be used through an online interface\footnote{TaskMiner can be used at http://cuda.dcc.ufmg.br/taskminer/}. We chose to implement those techniques at that level to benefit from the support of several data-flow analyses already in place in the LLVM infra-structure, such as scalar evolution and symbolic execution. This toolbox gave us the necessary means to disambiguate pointers and estimate safe limits to memory regions. Yet, our annotations are still inserted in C programs. Having them in a high-level programming language has two advantages: readability and portability across compilers. However, this modus operandi also brings two challenges. First, we cannot annotate partial code: to produce LLVM bitcodes, we need all the type definitions used in the target program. Second, we need to map information from LLVM’s intermediate representation back into C. A type inference engine for C let us deal with the first challenge \cite{38}. In this section we explain how we deal with the second.

The Scope Tree: To recover high-level information from the low-level IR, we have designed a supporting data-structure henceforth named the scope tree. The scope tree maps hammock regions into C constructions, such as while and if-then-else blocks. Each node of this graph represents a hammock region, augmented with meta-information, such as the program part that it represents. We keep track of these program parts via debugging information, which is appended into the LLVM IR via the \texttt{–g} flag passed to the clang compiler. We have an edge from node $s_1$ to node $s_2$ if region $s_2$ is nested within region $s_1$. In the absence of control-flow optimizations such as loop unrolling or dead-code elimination, each hammock region corresponds to some structured code block (code region delimitable by braces). Thus, we can find line numbers for each of the regions that we have marked as tasks after the expansion seen in Section 3.4.

Simplification: before we annotate programs, we proceed to simplify these annotations. Simplification happens via a system of rewriting rules, which explores identities. Typical identifies include, for instance, $x + 0 = x$, $\max(x, x) = x$ and $c \times \max(x, y) = \max(c \times x, c \times y)$. Identities are applied iteratively, until a fixed-point is reached. Because we do not support commutativity, a fixed-point is guaranteed to be always reached. Notice that the source code that we produce is not totally equivalent to the original program augmented with annotations. To be able to insert annotations, we format the original code. This operation involves, for instance, breaking lines containing multiple statements, and inserting delimiting braces within every block in the program, even one-liners. This said, it is still possible that a task, after expansion, maps to a single line that contains multiple statements, such as nested function calls. In this case, we do not annotate the target program. Section 4 provides data about TaskMiner’s capacity to annotate real-world programs.

Bounding Recursive Tasks During the evaluation of the TaskMiner, we observed that recursive programs experienced performance regressions due to the excessive creation of tasks. To avoid this kind of slowdown, we currently give users the possibility to bound the number of threads ever in flight, via a command line option, e.g., ./Taskminer \texttt{-r} 12 will limit the number of tasks to 12. We implement this feature directly at the source code level, as part of the final annotation of code. Our solution is simple, yet, as the reader shall perceive in Section 4, it brings non-negligible benefits onto recursive benchmarks. Task bounding is implemented via a global variable, statically linked in the program. This variable is cutoff in Figure 3. We insert code to increment it at the beginning of recursive functions that are invoked within task regions, and we insert code to decrement it at each return point of said functions.

Example 3.14 (Task Bounding). Figure 3 illustrates the strategy that we use to limit the number of tasks in flight due to recursive function invocation. The parameter DEPTH\_CUTOFF is determined by TaskMiner’s users.

There are more involved ways to bound the number of tasks. We believe that the state-of-the-art in the field today is the work of Iwasaki and Taura \cite{28, 29}. These authors propose different techniques to limit the creation of tasks, be it through the replication of code, be it through the estimation of work, given function inputs. Pragmas for the conditional creation of tasks, such as the one seen in lines 8 and 11 of Figure 3, lets us obtain much of the benefit of code versioning, as proposed by Iwasaki and Taura. However, we found work estimation of recursive functions a task too difficult to accomplish in general. Quoting \cite[p.355]{28}: “there are tasks which essentially do not have simple termination conditions (e.g., tasks traversing pointer-based trees)”. Thus, although simple, our recursion counters are general enough, handling even such tasks that are hard to bound symbolically. Nevertheless, we still would like to explore further ways to cut-off extremely fine-grained tasks in the future.
4 Evaluation

Runtime Environment: We have implemented the techniques described in this paper in LLVM 3.9 [34]. All our experiments were performed in a 64-bit 12-core Intel(R) Xeon(R) CPU E5-2620 at 2.00GHz, with 32K of L1 cache, 256K of L2 cache, 15M of L3 cache and 15Gb of main memory. We use OpenMP 4.0, from November of 2015.

Research Questions: In this section, we evaluate this implementation, focusing on four research questions:

- [Performance]: how do our automatically annotated programs compare against their sequential counterparts, or against manually annotated versions?
- [Optimizations]: what is the impact of the cost model (Section 3.3) and recursion bounding (Section 3.6) onto the programs that we annotate?
- [Versatility]: how effective is TaskMiner in finding opportunities to annotate general benchmarks?
- [Scalability]: what is the runtime complexity of our implementation of TaskMiner?

Benchmarking: In Sections 4.1 and 4.2 we use BSC-Bots (https://github.com/bsc-pm/bots/tree/master/serial) [22] and Swan (http://cuda.dcc.ufmg.br/swan/) [41]. In Section 4.3 we use the LLVM test suite, and in Section 4.4 we use random programs produced with CSmith [61]. Runtimes are averages of five executions.

4.1 Performance

Figure 10 compares the runtime of programs produced by TaskMiner against either the original program or versions of said programs annotated manually. In this experiment we use the benchmarks BSC-Bots [22] (fft to jacobi in Fig. 10) and Swan [41] (dfs to private). Both these benchmarks come with sequential and parallel (manually annotated) versions. Baseline and annotated sources are compiled with gcc-6 -O3.

The programs annotated by TaskMiner were faster than their sequential counterparts in 13 of the 16 samples. Most of the programs in these benchmarks were classic Divide and Conquer algorithms, such as Strassen’s matrix multiplication, knapsack and fft. In 6 cases, automatically annotated programs were close or slightly faster than the manually annotated versions. In three examples where TaskMiner fell behind the original samples, e.g., fib, floorplan and bellmanford (bell-ford), it produced code faster than the manually annotated competitor. The version of bellmanford produced by TaskMiner is slower than its sequential baseline. The slowdown is not due to the cost model. This algorithm traverses a graph implemented as an array of arrays, and data dependencies cause serialization at runtime. We emphasize that the human-annotated versions of our benchmarks have not been tested originally in our architecture. For instance, BSC-Bots was evaluated on an SGI Altix 4700 with 128 processors [22]. Thus, hardware differences might be accountable for the slowdown in some of the manually annotated samples. Nevertheless, this experiment demonstrates that TaskMiner can deliver non-negligible speedups comparable with sequential and manually annotated programs.

4.2 Optimizations

This paper describes two ways to optimize task placement. Both these techniques are based on the idea of “task pruning”: we avoid creating tasks if we deem them unprofitable. The first technique prunes tasks judged unprofitable by the cost-model of Section 3.3; the second prunes tasks that are too deep in the recursion stack, as described in Section 3.6. Figure 11 illustrates the benefits of these two techniques. We
show the gains — in terms of speedup over the baseline — of the three benchmarks that benefit the most from each type of task pruning, among those earlier seen in Figure 10.

In three benchmarks, private, boruvka and bellman-ford, the cost model avoids creating tasks around loops that initialize data-structures. For instance, the following loop, in line 75 of bellmanford.c, would be parallelized by TaskMiner, were it not for the cost-model marking it as unprofitable:

```c
for (long unsigned i = 0; i < N; i++)
    *(G + i * N + j) = rand();
```

Similarly, recursion bounding (Sec. 3.6), although simple, is effective in eliminating the excessive number of tasks. Figure 11 shows the effect of this optimization upon three BSC-Bots benchmarks: health, nqueens and strassen. These programs were designed to illustrate the parallelization of divide-and-conquer algorithms [22], and contain a large number of recursive calls (see RFC) in Figure 11. Pruning at higher levels of the recursion tree lets them deliver non-negligible speedups onto the baseline programs. Were pruning absent, then we would observe slowdowns in health and nqueens. This experiment shows that our optimizations are effective to improve the quality of the code that we generate.

4.3 Versatility

The benchmarks used in Section 4.1 have been coded to demonstrate the power of parallel systems; hence, they have been written in a way that simplifies the discovery of parallelism in programs. In this section, we address the following question: “can TaskMiner” find parallelism in general programs? To answer it, we have applied TaskMiner onto the 219 C programs available in the LLVM test suite, and have compared the annotated version against the sequential version compiled. TaskMiner annotates a benchmark if: (i) it can find symbolic bounds to every memory access used in a vane; and (ii) the vane is large enough to pay for the cost of creating threads. Under these constraints, we have discovered tasks in 63 benchmarks. Figure 12 relates the number of tasks and the number of instructions in the 30 largest benchmarks that we have used. To avoid counting multiple C files for the same benchmark, Figure 12 contains only benchmarks that consists of a single file (present in LLVM’s SingleSource folder). As this experiment demonstrates, TaskMiner can annotate a non-trivial number of real-world benchmarks.

Most of the benchmarks, e.g., 27, that have been automatically annotated did not give us speedups, because the regions marked as tasks were too small to influence the program’s runtime. We have observed slowdowns in 17 benchmarks. In this case, interactions between data-structures end up forcing the OpenMP runtime to serialize execution of tasks. Most of these slowdowns were inferior to 10%. In one case, MiBench/office-stringsearch, our program was 11x slower. Nevertheless, we have measured speedups above 5% in 19 benchmarks. In Misc/lowercase, the speedup is above 3.5x, and in three cases, it is above 1.5x. Figure 13 shows our 10 largest speedups. We emphasize that this experiment did not involve any human intervention. Hence, TaskMiner enables the discovery of parallelism at zero programming cost.

4.4 Scalability

The analyses described in Section 3 have a worst-case quadratic time. This is the worst-case scenario of the symbolic range analysis of Section 3.1 and the task-expansion algorithm of Section 3.4. Nevertheless, our implementation is fast in practice. To support this statement, we have used CSmith [61], a random code generator, to produce 100 programs of varying sizes, which we then fed to the TaskMiner. Figure 14 shows the result of this experiment. Our largest program had 4,999 lines of C, and TaskMiner could process it in 1.2 seconds. The $R^2$ of polynomials of degree 1, 2, 3 and 4 is, respectively, 0.795, 0.890, 0.909 and 0.911, suggesting very small differences between polynomials of degree 2 or more.
Each dot represents one of 100 programs produced by CSmith

\[ y = 4 \times 10^{-5} x^2 + 0.047 x - 25.15 \]

Figure 14. Runtime of TaskMiner vs size of input programs. We have fit a degree-2 polynomial on this dataset. We cannot control the size of programs produced by CSmith; hence, it is difficult to push the limit above 5,000 LoC.

5 Related Work

Mainstream compilers have only recently added support to OpenMP 4.0’s task parallelism. An implementation of clang supporting tasks was released in March 14th, 2014. A few weeks later, in April 22nd, 2014, such support was also announced in gcc 4.9. Because the necessary infrastructure for the implementation of tasks is new, currently there are no tools, other than TaskMiner, that annotate programs with these directives. Nevertheless, there exist much research aiming at the automatic parallelization of programs. This section examines elements in this list related to our work.

Directive-based code annotation standards provide programmers with an easy-to-use parallel programming model. Task-based extensions to these standards have resulted in OpenMP 4.X, StarSs [7, 43, 46, 55] and OmpSs [10, 19]. Such programming models come with tools that help programmers to find the best annotations for their code. For example, Tareador [4] enables a programmer, by means of a graphical interface, to annotate sequential code, thus allowing the identification of potential task parallelization opportunities. Contrary to Tareador, this paper proposes an approach that enables the automatic insertion of OpenMP task annotations to relevant fragments of a sequential program.

The techniques that we use to map program regions to tasks is similar to analyses previously used in the generation of code for data-flow machines. Data-flow programming was originally proposed as a candidate to enable task-based parallelism [1, 15, 25]. Agrawal et al. [1] extended data-flow programming with task input/output specifications in Cilk++. Vandierendonck et al. [56] further extended Cilk++ with dependency clauses to facilitate the design of complex parallelization patterns. Both groups presented a unified scheduler based on fork-join parallelism [57] that enabled the execution of task-based applications. Other approaches have also used data-flow graphs to exploit parallelism, like Data-Driven Tasks [54], where the programmer can use put/await clauses to determine task arguments before execution. Function-based task parallelism was proposed by Gupta et al. [25] to use function arguments as a way to specify task dependencies. Notice that such research efforts focused on giving to programmers the tools to construct parallel programs. The automatic annotation of ordinary programs with data-flow constructs was not among their goals.

Many systems have been developed to extract task parallelism from sequential programs (semi) automatically. Examples include OSCAR (Optimally Scheduled Advanced multiprocessor), Multigrain Parallelizing Compiler [27, 31], MAPS (MPSoC Application Programming Studio) [11, 12], and DiscoPoP (Discovery of Potential Parallelism) [16, 36]. Besides parallelization systems, tools like Paraver [13, 14], Aftermath [18], DAGvis [26], and TEMANEJO [9] were designed to enable performance analysis and visualization of task-based programs. Although these tools help the programmer in adapting code to run in parallel, they are not fully automatic. For instance, the work that is the closest to ours, in reach and effectiveness, in our opinion, is the suite of techniques proposed by Ravishankar et al. [49]. The type of irregular loops that they handle is impressive; however, the lack of the runtime support à la OpenMP 4.X still requires them to modify code before parallelization. In their words: “For all benchmarks and applications, all functions were inlined, and arrays of structures were converted to structures of arrays for use with our prototype compiler.”

6 Conclusion

This paper has described a methodology to annotate programs with task parallel pragmas, which we could demonstrate to be effective on general programs. This methodology does not introduce any fundamentally new static analysis or code optimization; in this field, we claim no contribution. Instead, our contributions lay into the overall design that eventually emerged from a two-years long effort to combine existing compilation techniques towards the goal of profiling from the powerful runtime system that OpenMP’s task parallelism brings forward. During this period, we went through many dead-ends; too many, indeed, for a 11-pages report. Nevertheless, this experience lets us single out a few elements from the compiler literature, such as symbolic range analysis and windmills, which we could use to solve challenges related to automatic code annotation. Much work is still left to be done, until we can reach a stage in which automatic annotations can beat consistently manual interventions. In particular, our methodology asks for more aggressive tuning strategies. Techniques such as Trancoso’s [17], Iwasaki’s [29] or Emani’s [23] have already been shown to be effective in this domain, and we want to explore them further.

Software: TaskMiner can be used directly through an online interface available at http://cuda.dcc.ufmg.br/taskminer/
References


[39] Gleison Souza Diniz Mendonça, Bruno Campos Ferreira Guimarães, Péricles Rafael Oliveira Alves, Fernando Magno Quintão Pereira,
Automatic Annotation of Tasks in Structured Code

PACT’18, November 1-4, 2018, Limassol, Cyprus


Philippe Virouleau, Pierrick BRUNET, Francois Broquedis, Nathalie Furmento, Samuel Thibault, Olivier Aumage, and Thierry Gautier.