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),* 3 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 (?), OpenACC (?), OpenHMPP (?), OpenMPC (?), OpenSs (?), Cilk++ (?) and Tareador (?). Annotations work as a meta-language: they let developers grant parallel semantics to syntax originally written to execute sequentially. Combined with modern accelerators, they have led to substantial performance gains (?????). Nevertheless, although convenient, the use of annotations is not straightforward, and still lacks supporting tools.

Annotations such as OpenMP or OpenACC still require developers to worry about typical hassles of the parallel world, such as race conditions and deadlocks. Moreover, because they are used in tandem with imperative programming languages, such troubles are only made worse by pointer aliasing. There exist tools that insert automatic annotations in programs (?????). All these technologies explore data-parallelism – the possibility of running the same computation independently on different data. Thus, task parallelism 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 (?). This ability brings said systems closer to irregular programs such as those that implement graphs and worklist algorithms à la Pingali (???). The purpose of this work is to address this omission.

This paper describes TaskMiner, a compiler that mines task parallelism from programs. In order to fulfil this goal, TaskMiner solves many 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 all the memory blocks accessed within those regions (Section 3.2). Third, it extracts parameters from the code to estimate the profitability of tasks, given the cost to create them. 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). Additionally, TaskMiner determines which program variables need to be replicated at each new task, or shared among them (Section 3.5). Finally, it maps all this information back into source code, producing annotations that programmers can read (Section 3.6). This work demonstrates the design and the implementation of these technologies, as well as their benefit.

The major benefit provided by TaskMineris, essentially, performance and readability. TaskMiner receives as input a C program, and produces, as output, a C program annotated with OpenMP directives that enclose potential tasks. 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 (?) or Bots (?) 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 their sequential –unannotated– versions, as we report in Section 4.

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

2 Overview

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. In the rest of this section, we discuss the most important challenges, using this discussion as a motivation to the developments that follow in Section 3.

```
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 1. 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 1 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 1 also introduces the second challenge that we tackle:

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

The creation of tasks involves a non-negligible runtime cost due to allocation, scheduling of resources 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 (?), 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 1, 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 1-2 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 2. 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*" (?). This observation is particularly true in the context of recursive, fine-grain, tasks, as we analyze in Section 3.4. Figure 2 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 2. The guard in line 7 ensures that we never exceed DEPTH_CUTOFF, a predetermined threshold. This example, together with Figure 1, 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 1. 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 U) {
 2    for (int i = 0; i < N; i++) {
 3      if (L < i && i < U) {
 4        int j = i * N;
 5      #pragma task untied default(shared) firstprivate(j)
 6        while (j < (i + 1) * N) {
 7          sum += V[j];
 8          j++;
 9        }
10      }
11    }
12    return sum;
13  }
```

Figure 3. 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 3 must be privatized. In the absence of such action, that variable will be shared among all the tasks created at line 5 of Figure 3. 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

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

5 Related Work

6 Conclusion