Automatic Identification and Annotation of Parallel Tasks in Structured Programs

Anonymous Author(s)

Abstract

We need to write our abstract here.

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 ACM SIGPLAN Conference on Programming Languages, New York, NY, USA, January 01–03, 2017 (PL'17),* 2 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

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 (**?**), OpenACC (**?**), OpenHMPP (**?**), OpenMPC (**?**) and OpenSs (**?**). Annotations work as a meta-language: they let developers grant parallel semantics to syntax originally written to execute sequentially. The compiler-related literature contains a generous load of success stories describing the use of annotation systems to parallelize programs. Combined with modern accelerators, they have led to substantial performance gains (**?????**). Yet, even if convenient, the use of annotations is not straightforward, and 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 or 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. Task parallelism remains still uncharted land, in what concerns automatic annotation. Such fact is unfortunate, because much of the power of current annotation systems lays on their ability to create tasks We need to find references!. This ability brings said systems closer to irregular programs such as those that implement graphs and worklist algorithms à la Pingali (**???**). The goal of this work is to address this omission.

PL'17, New York, NY, USA
2017. 978-x-xxxx-xxxx-x/YY/MM…$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

This paper describes TaskMiner, the first compiler that mines task parallelism from programs. To fulfill this goal TaskMiner solves three challenges. First, it find 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 maps such information back into source code, producing annotations that programmers can read (Section 3.3). The goal of this paper is to describe the design and the implementation of these three technologies, and to demonstrate their benefit.

Such benefit is performance and readability. TaskMiner receives as input a C program, and produces, as output, a C program annotated with OpenMP directives that identify 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.

2 Overview

3 Solution

3.1 Mapping Program Regions to Tasks

3.2 Find Symbolic Bounds of Arrays

3.3 From IR to Source Code

4 Evaluation

We aimed to show that our tool exceeded in three main aspects that surround parallel programming. Thus our work evaluation stands on these 3 pillars:

- Practicality
- Applicability
- Performance

Practicality. Programming in a parallel pattern is deemed difficult by programmers mainly because of the complexity of its data structures and algorithms. We show that our tool is simple and practical, for it does not involve changing the code and data structures in order to fit in a parallel pattern. The code is simply annotated by compiler directives.

Applicability.

Performance. When it comes to parallel computing, it is undeniable that the main achievable goal should be performance. There's no reason why programmers would opt for parallelism other than the desire for faster programs. Therefore, we ought to show that the programs annotated with TaskMiner are faster than their serial counterparts.

5 Related Work

6 Conclusion