Guido: Not sure if we should dive into a discussion of what is a regular or an irregular program; to be even more precise we are actually mining task parallelism (for this moment) only within loops; hence I propose this new title Pedro: I think we should say "Automatic Mining of Task Parallelism in Programs" because we might mine task parallelism in recursive functions as well.

Removed: Task Mining in Irregular Programs.Automatic Mining of Task Parallelism in Programs

Abstract

We need to write our abstract here.

1. Introduction

It is undeniable that parallel computing is nowadays the most pursued alternative by developers when striving for more application performance [3]. Two main reasons stand behind the actual urge for parallel programming. First, the ever increasing industry of multi-cores and many-cores architectures propitiates the development of software with such behaviour [2, 9]. Guido: Fine, but it seems to me a very convoluted way to say that this was due to the emergence of multicore/manycore architectures as the alternative to maintain transistor power-density under control. Second, programmers often code in simple parallel patterns [8], which evidences their awareness and will for parallelism. Guido: Not sure if programming using patterns is an evidence that supports the urge for interest in parallel programming; on the contrary, it seems to me that it is more a consequence to tackle the complexity of the problem.

Although extensive work has been done to scavenge parallelism in regular applications [4], finding good parallel opportunities in irregular code remains a grueling task. Guido: We need to re-phrase this if we want to use this statement to introduce the motivation to our work; a number of the examples that we are using do not have pointers, and thus could not be classified as irregular code....

Programming parallel applications is a hard task, as programmers ought to take care of many aspects of the job such as identifying and keeping variable dependences correct and assuring synchronization and scalability of threads/processes whilst avoiding race conditions. However, finding parallelism in programs that were not coded in a parallel paradigm Removed: annotated sequential code is not a simple task, as compiler static analyses are not precise enough to identify parallel regions [5]. For example, due to the conservative trait of static analyses, loops are frequently marked as iteration-dependent when, in reality, their iterations might run in parallel most of the time. As a matter of fact, many dependences pointed by the compiler might never occur during runtime, resulting in several missing opportunities for parallelization.

A way to approach this problem is to use a programming model that can capture and enforce dependences between iterations at runtime whenever they occur. In such model, those iterations that do not have *loop-carried* dependencies are free to run in parallel, while iterations that dynamically create loop-carried dependences are serialized and dispatched in dependence order by a system runtime. In recent years, task-based execution models have proven itself to be a scalable and quite flexible approach to extract regular and irregular parallelism from sequential code [**?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**]. In this model, the programmer uses a *task* directive to mark code regions in the program that are potential tasks and lists their corresponding dependences. The OpenMP task scheduling [1] is an example of such model. Although OpenMP task simplifies the mechanics of dispatching and running tasks, the burden of finding code regions to be annotated with task directives still lies upon the programmer.

Work has been carried out to generate OpenMP annotations automatically given a set of instructions that define a region [6, 7, 10]. Guido: Need to provide a short paragraph here to give an idea why these works do not solve the problem of automatically extracting parallelism from loops with loop carried-depedenciss.

This paper presents an automatic technique that extracts task parallelism from loops which cannot be proved by static analysis to be loop-carried independent.

Guido: I HAVE STOPPED HERE; WILL CONTINUE LATER

The static analysis finds irregular loops (i.e., loops with loop-carried dependences) and then speculates whether these loops can be safely executed in parallel or not. These loops are annotated with compiler directives, to point out the possibility of task-parallelization within that sequence of code. Then a runtime able to compile these directives runs the tasks for the annotated code. During execution, the dependences are resolved by the runtime and the code is executed in parallel.

The key insight of the tool is that it does not rely on the programmer to find possible parallel sites. It does so automatically, and the runtime is solely responsible for unraveling the dependences between tasks. The final product of this combination is a high-performant framework that lets us run algorithms that are traditionally difficult to parallelize.

We have implemented this technique in the LLVM compiler infrastructure. We tested our technique on three large benchmarks (CASTOR, LG Graphical Application, ???) and compared it to the manually annotated version. We show that XX% of the manually annotated regions are found by our technique. Our technique also finds XX% more possible task regions than what is manually annotated. Regarding performance, the automatically annotated code can be up to X times faster than the manually annotated one.

2. Overview

3. Solution

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

References

[1] E. Ayguadé *et al.*, "The design of openmp tasks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 3, pp. 404–418, 2009.

[2] B. A. Haugen, "Performance analysis and modeling of task-based runtimes," 2016.

[3] W.-m. Hwu, "What is ahead for parallel computing," *Journal of Parallel and Distributed Computing*, vol. 74, no. 7, pp. 2574–2581, 2014.

[4] M. Kulkarni *et al.*, "How much parallelism is there in irregular applications?" in *ACM sigplan notices*, vol. 44, no. 4. ACM, 2009, pp. 3–14.

[5] M. Kulkarni *et al.*, "Optimistic parallelism requires abstractions," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 211–222, 2007.

[6] G. S. D. Mendonça *et al.*, "Automatic insertion of copy annotation in data-parallel programs," in *Computer Architecture and High Performance Computing (SBAC-PAD), 2016 28th International Symposium on*. IEEE, 2016, pp. 34–41.

[7] K. Pingali *et al.*, "The tao of parallelism in algorithms," in *ACM Sigplan Notices*, vol. 46, no. 6. ACM, 2011, pp. 12–25.

[8] G. Pinto *et al.*, "A large-scale study on the usage of java's concurrent programming constructs," *Journal of Systems and Software*, vol. 106, pp. 59–81, 2015.

[9] H. Vandierendonck, G. Tzenakis, and D. S. Nikolopoulos, "Analysis of dependence tracking algorithms for task dataflow execution," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 10, no. 4, p. 61, 2013.

[10] C.-K. Wang and P.-S. Chen, "Generating task clauses for openmp programs."