Journal of Computational Science xxx (2016) xxx–xxx

Contents lists available at ScienceDirect

Journal of Computational Science

journal homepage: www.elsevier.com/locate/jocs

Controlling the granularity of automatic parallel programs

Alcides Fonseca*, Bruno Cabral

CISUC, Department of Informatics Engineering, University of Coimbra, Coimbra, Portugal

ARTICLE INFO

Article history:
Received 9 November 2015
Received in revised form 23 April 2016
Accepted 28 June 2016
Available online xxx

MSC:
00-01
99-00

Keywords:
Compiler
Parallel
Granularity

ABSTRACT

Programming for concurrent platforms, such as multicore cpus, is very time consuming and requires fine tuning of the final program in order to optimize the program parallel layout to the hardware architecture. Parallelization of programs is done by identification parts of code (tasks) that can be executed concurrently and execution in different threads.

Current approaches for automatic parallelization cannot achieve the same performance of manually parallelized programs. Current tools are limited and either parallelize everything possible, or are limited to parallelizing the outer loops, which may miss potential parallelism that could improve the program. Some approaches have controlled granularity during execution only, but without any relevant speedups. Automatic Parallelizing Compilers have shown little overall speedup without the manual guidance of programmers in terms of granularity. This work addresses the issue of achieving performant programs from a fully automated parallelization.

We propose a cost-model to decide between different parallelization alternatives. By performing static-analysis, we are able to estimate the time of tasks and parallelize them only if the time is larger than the overhead of task spawning. Because the information during compilation might not be enough to make that decision, we delay some of the decisions to runtime, when all variables are available. Thus, we use an hybrid approach that performs optimizations at compile-time and at runtime.

Although we apply our model in the Java language on top of the Æminium runtime, our approach is modular and can be applied to any programming language in any task-based runtime for shared-memory.

We have evaluated our approach in existing benchmark programs, in cases where a wrong granularity value would result in slowing down the programs. We were able to achieve speedups greater than versions without granularity control, or with runtime-based granularity control information. We were also able to generate programs with better performance than the state-of-the-art Java automatic parallelizing compiler. Finally, in some cases we were able to outperform the human programmer.

1. Introduction

Nowadays, parallelization is the most popular and cost-effective solution for improving the performance of computational intensive programs. This process is seldom straight-forward and is time consuming, requiring the identification of parallel tasks, the inclusion of synchronization logic in order to keep the same semantics and optimization of memory and CPU usage.

Optimizing a parallel program is very time consuming as it requires a trial-and-error approach. Different programs behave very differently when executing in parallel depending on the data and instruction layout. In order to optimize a program, different decisions have to be made.

One has to know how many threads a program will use. Often the parallelism extracted from a program is not enough to occupy all CPU resources and, in those cases, a smaller thread count should be used. For high-parallelism, a thread number equal to the number of cores or double the number of cores achieves the best result.

Another decision is how many tasks to create, or how large tasks should be (granularity). A parallel loop that iterates 1000 times can create 1000 tasks, or even threads. Few computers have that number of cores, so there would exist a high overhead in thread creation. The ideal distribution of iterations per tasks depends on several factors, being difficult to achieve even by an experienced programmer. This is very time consuming as it requires trial and error.

Another decision is which work-stealing or work-sharing approach to use. Some programs spawn several tasks in the beginning, while others spawn tasks evenly through time and require more load balancing.

* Corresponding author.
E-mail addresses: amaf@dei.uc.pt (A. Fonseca), bcabral@dei.uc.pt (B. Cabral).

http://dx.doi.org/10.1016/j.jocs.2016.06.005
1877-7503/

Finally, irregular programs need to decide whether to parallelize or not. For this, several cut-off mechanisms have been studied, but there has not been any recommendation of which to use, as it heavily depends on the program details.

All of these aspects colide with each other and studying the different combinations is very time consuming and requires executing the program before hand. In programs that have a very large execution time, trying different combinations may not be feasible.

In this paper we propose an automated approach for controlling the granularity of parallel programs. Our approach uses a cost-model that is applied during static analysis and can prevent the excessive creation of tasks during compilation and execution. This control is useful to avoid spending time and memory building boilerplate structures to hold the parallel execution of code.

Our model is based on the fact that some instructions are more expensive than others. We micro-benchmark different Java operations as well as the overhead in creating new tasks. We compose the values of terminal instructions in the Java language to model each AST node, and we analyze method invocations using runtime variables to delay the decision of task spawning to runtime. Our model works even in recursive calls by performing two passes per method. Furthermore, we also model the memory usage of each possible task to decide if the memory overhead would require swapping.

Granularity control has been tackled mostly during execution, where the internal runtime status is known. This is a novel approach that avoids overheads in dynamic scheduling is most cases by moving that decision to compilation time. If the decision depends on runtime information, the overhead also occurs, but the decision is made with a prediction of the task cost.

The contributions of this work are as follows:

- Definition of a novel approach and cost-model for automatically controlling the granularity and spawning of new tasks in parallel programs.
- Implementation of the approach in a automatic parallel compiler for the Java Language.
- Evaluation of the solution by comparing the performance of programs parallelized using this model to programs without granularity control and other state-of-the-art approaches.

The remaining of the paper is as follows: Section 2 describes the state of the art in parallel programming and granularity control. Section 3 describes our approach and the model used. Section 4 provides an evaluation of our solution compared to other approaches. Finally, Section 5 draws conclusions and presents future work.

2. State of the art

In the last decade, scientific computing has been using parallel programming as a default for large-scale processing. In order to improve the performance of scientific programs, parallel programs must execute as fast as possible. This means optimizing programs to take the maximum advantage of the hardware. In order to achieve this goal, the granularity of tasks has to be statically or dynamically adjusted in order to balance the load between processors.

Traditionally, parallel programming has been achieved through either manual parallelization or using annotation-based languages. Manual parallelization has been done on top of threads, requiring a manual granularity definition. Cilk [1] and OpenMP [2] are two popular language extensions to easily express parallelism on top of sequential code that support task and data parallelism. Task granularity is controlled by manually identifying tasks around code, or by defining the size of each chunk of parallel loops.

More recently, new parallel-by-default languages have been developed to simplify the writing of parallel programs. X10 [3], Fortress [4] and Chapel [5] are examples of those languages, which contain parallel constructs in the language (such as `parallelfor`). By alternating between parallel and sequential constructs is also possible to define the granularity of each task, leaving that burden to the programmer. Æminium [6] automatically generates parallel programs from sequential code, based on access-permissions. In this language, programmers define the contracts for variable accesses inside code blocks and the compiler will parallelize as much as possible. Granularity control is not possible in this case, as the programmer does not even need to know which parts will be executed in parallel or not. However, from a performance standpoint, Æminium programs rarely have speedup without granularity control. The existing work-around was to annotate standard library functions with a `@Cheap` annotation [7]. This work addresses the issue of automatic control of the granularity of tasks.

Another approach for generating parallel programs is to use automatic parallelization compilers such as SUIF [8], Cetus [9] and Par4All [10]. These compilers apply the Polyhedral Model in order to extract loop parallelism. SESAM [11] has controlled the granularity of automatic parallel programs for asymmetric Multiprocessor System-on-Chip by executing Par4All programs on the simulator during compilation. Cetus performs parallelization only of the outer loops, which may limit the parallelization on larger inner loops inside small outer loops. Loop parallelization has been done during runtime [12], but without any relevant speedups. Overall, these automatic compilers show little performance without any assistance from the programmer on thread-level parallelism identification or granularity control. This paper addresses that issue by providing a granularity control mechanism for automatic parallelization.

zJava [13] and OoOJava [14] are compilers for the Java language that perform task-based automatic parallelization. zJava is based on a runtime system that manages data accesses, while OoOJava performs Thread-Level-Speculation on annotated blocks of code, thus being limited in the type of operations that can be parallelized, and requiring identification of parallel code. JPar [15] is another automatic parallelization compiler for Java based on inferred access annotations, similarly to Æminium. Since parallelization is very fine-grained, we reach a point where the overhead of scheduling a task is greater that the time obtained by parallelizing the task, resulting in an overall slowdown. We have used an heuristic of only parallelizing methods with 10 instructions, but this approach is very limited.

Other approach to granularity control is ZettaBricks [16], which improves the configuration of parallel programs on each run, based on profiling. This approach only works for programs that are executing several times over the time. ARTA [17] uses a runtime policy to adaptively change the granularity of a STM engine. Multi-versioning [18] has been used to generate several versions of loops, which different unrolling levels and dynamically picking the version based on runtime information. This approach has showed speedups over loop-based applications.

Another alternative to manage the granularity of parallel programs is decide whether to schedule a task or not during execution. Lazy Task Creation [19] is an approach that decides to create a task or not depending on a the amount of tasks in the system. The two most common factors are a threshold of maximum tasks in the system and a maximum depth in the task graph [20], a mixed approach or whether at least one processor has no work [21]. In Oracle, the user defines the asymptotic complexity of a function and that is used at runtime whether to spawn a new task or not. This approach requires a heavy human interaction and has not gained traction.

Finally, we have used a simpler Cost-Model for deciding between the GPU or CPU for Java code execution [22], also using micro-benchmarks to support several GPU and CPU architectures. A Cost-Model framework has also been used in [23] for

Please cite this article in press as: A. Fonseca, B. Cabral, Controlling the granularity of automatic parallel programs, J. Comput. Sci. (2016), http://dx.doi.org/10.1016/j.jocs.2016.06.005