Extracting Coarse-Grain Parallelism in General-Purpose Programs

Sean Rul Hans Vandierendonck Koen De Bosschere
Ghent University
{sean.rul,hans.vandierendonck,koen.debosschere}@elis.ugent.be

Abstract

While the chip multiprocessor (CMP) has quickly become the predominant processor architecture, its continuing success largely depends on the parallelizability of complex programs. In the early 1990s great successes were obtained to extract parallelism from the inner loops of scientific computations. In this paper we show that significant amounts of coarse-grain parallelism exists in the outer program loops, even in general-purpose programs. This coarse-grain parallelism can be exploited efficiently on CMPs without additional hardware support.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features—Concurrent programming structures

General Terms Design, Performance

Keywords thread-level parallelism, coarse-grain parallelism, DO-ACROSS

1. Introduction

In the recent past the mainstream processor was one big monolithic processor core. Due to power and performance issues, however, the industry moved to chips filled with multiple simpler cores, the chip multiprocessors (CMP). In order to tap the full processing power of a CMP one needs parallel programs. Since rewriting or developing programs with threads and explicit synchronization is an intricate and time consuming job, a more attractive course is automatic parallelization of sequential code. A lot of research has gone into finding loop-level parallelism operating on array data structures. Significant advances have been made in programs from the scientific computing community, whose inner loops exhibit large loop bodies, high loop trip counts and predictable data dependences. In contrast, this work focusses on extracting DO-ACROSS parallelism in the outermost toplevel loops of general-purpose programs, which corresponds closely to pipeline-like operations on a data set.

2. Framework

This section describes the different stages of our framework (Rul et al. 2006) that aid the programmer in a partially automatic way to parallelize a program. The goal is to detect coarse-grain parallelism for which thread creation and startup overheads become marginal. Consequently, our interest goes to the outer loops of a program.

Defining code regions As we are interested in coarse-grain parallelism, we record dependences only between large regions of code. We consider three types of code regions: *functions*, *loops* and *snippets*. Snippets are code fragments that contain memory references, but do not contain loops or function calls. Thus, snippets are potentially data dependent on other code fragments. The different code regions in a program are determined by a static analysis and passed on to the profiler in the next stage.

Profiling dependences Instead of striving for exact (but conservative) static analysis of programs, we employ dynamic control and data dependence analysis. This dynamic analysis turns out to correctly identify coarse-grain parallelism in practice. Indeed we have observed that the coarse-grain dependencies between large code regions are very consistent across program inputs. Yet, verification is still required.

The control dependences are tracked in order to respect the sequential semantics when parallelizing a sequential program. Hereto, we structure the program as a tree where each node in the tree corresponds to a previously defined code region. In case a code regions has multiple children, the children are sorted in sequential order. The actual control flow corresponds to tracing a path through the tree, where control flow can move down in the tree (entering a sub-region of the current region) or it can move up in the tree (exit a region).

Data dependences indicate which code region (consumer) is reading a memory location that was last written by a code region (producer). For each data object we record a *memory dependence matrix* which scales dynamically based on the number of producers and consumers for the data object in question. In a matrix the entry at row f and column g records the number of times that code region g consumed a value produced by code region f in that data object. Statically allocated objects are retrieved from the symbol table of the program, while dynamically allocated objects are identified during program execution by monitoring calls to memory allocation routines (`malloc`, `calloc`, `realloc`, `alloca` and `free`). To know the code region that was the last one to produce a value at a particular memory address we use the *last producer table*. Note that the producers are tracked at the smallest writable quantity (i.e. bytes) in order to correctly identify dependences. This is necessary since each field of a data object can have a different producer.

Analyzing dependences Control and data dependences between code regions are recorded in a *joint control data dependence graph* (JCDDG) (Figure 1). This graph contains rectangular nodes which represents the code regions and elliptic nodes which represent the data structures. Code regions are linked by edges according to

PPoPP'08, February 20–23, 2008, Salt Lake City, Utah, USA.
ACM 978-1-59593-960-9/08/0002.

control flow (solid lines). Data flow edges (dotted lines) indicate which functions access a data structure. The data flow edges point from a function to a data structure when the function writes the data structure. The data flow edge points from the data structure to a function if that function reads the data structure. Data flow edges that are marked with "`lc`" are loop-carried data dependences.

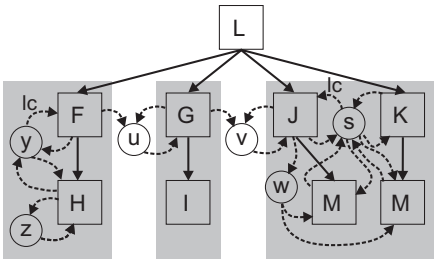

Figure 1. Joint control data dependence graph

The data dependences are also represented in an *inter-procedural data flow graph* (IDFG). The nodes in this graph represent code regions and an edge from a node A to a node B indicates that code region B is consuming data produced by code region A. Note that the IDFG contains information that is irreducible from the JCDDG. If a data structure has multiple producers, one cannot deduce from the JCDDG which producer a consumer is reading data from.

The format of parallelism we are looking for is depicted in Figure 2. A loop contains multiple code regions that are data dependent on each other, but has no interdependent clusters. If there are loop-carried dependences, then they point from a code region to itself. To detect this code template every loop in the JCDDG containing function calls is analyzed. We match the code template to every loop in the program. Initially, every code region is a separate cluster, so it is unlikely that the template matches, due to control flow or data dependences. To improve this match, we iteratively merge clusters until the template matches. Each cluster is treated as an indivisible unit of work, either because of control dependences or because of data dependences in that cluster.

In the first step we cluster code regions based on control dependences. We recurse through each of the child nodes and mark every node recursively as a member of the corresponding cluster. If a node is reached that is already assigned to a cluster, then the clusters are merged. The second step is to cluster code regions by data dependences. Hereto we start by mapping the clustering of the JCDDG on the IDFG. The goal of this step is to remove data dependent loops between clusters by merging the involved clusters. The algorithm to perform the clustering by data dependences uses topological sorting of clusters and detection of connected components to simultaneously sort the clusters by data dependences and merge mutually dependent clusters. The result of the final clustering is depicted by the grey background in Figure 1, resulting in the targeted code template of Figure 2.

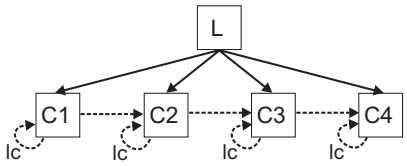

Figure 2. Code template

Parallelization The parallelism in a loop of the format of Figure 2 is the most general form of DO-ACROSS parallelism. The program is parallelized by executing instances of the clusters in different threads. Depending on the occurence of loop carried dependences, the parallelization is a form of loop unfolding, software pipelining or a combination of the first two. In case of loop unfolding each thread runs an iteration of the loop and synchronization happens between different iterations of the same cluster. In case of software pipelining each thread executes a different cluster and the necessary synchronization between different clusters is added.

3. Evaluation

Figure 3 shows for several benchmarks the outer parallel loops that were detected. Each part of the bars correspond to the execution time of the sequential part of the program or one of the parallelizable loops (pipeline n). We show for each loop its original execution time (S: sequential version) and its estimated reduced execution time (P: parallel version). For DO-ACROSS loops $t_{par} = t_{1iter} + (n-1)t_{lcmax}$, where n is the number of iterations and t_{lcmax} the execution time of the most time-consuming cluster with loop-carried dependencies.

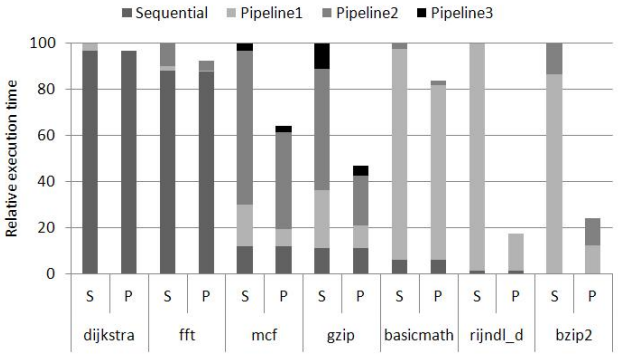

Figure 3. Relative execution time of detected pipelines in sequential execution (S) and in parallel execution (P)

The benchmarks on the right-hand side contain more coarse-grain parallelism and are more suitable for parallelization. On an 8-core Sun UltraSPARC T1 processor we have obtained speedup results of factor 5 to 12 for that group of programs.

4. Conclusion

We presented a framework for extracting coarse-grain parallelism from a sequential program using dynamic analysis techniques for obtaining precise control and data dependence information. Interdependent code regions in loops are recursively merged until the overall loop structure matches a preset template which can be parallelized.

Acknowledgments

Sean Rul is supported by a grant from the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen). Hans Vandierendonck is a post-doctoral researcher of the Fund for Scientific Research-Flanders (FWO). This research is also funded by Ghent University and HiPEAC.

References

Sean Rul, Hans Vandierendonck, and Koen De Bosschere. Function level parallelism lead by data dependencies. In *dasCMP: Workshop on Design, Architecture and Simulation of Chip Multi-Processors*, 2006.