COMPARATIVE SURVEY OF APPROACHES TO AUTOMATIC PARALLELIZATION

Nicholas DiPasquale, Vijay Gehlot and Thomas Way
Department of Computing Sciences
Villanova University
800 E. Lancaster Ave.
Villanova PA, 19085
nicholas.dipasquale@villanova.edu
vijay.gehlot@villanova.edu
thomas.way@villanova.edu

ABSTRACT

Automatic parallelization in a compiler is becoming more important as computer technologies expand to include more distributed computing. This paper focuses on a comparative study of past and present techniques for automatic parallelization. It includes techniques such as scalar analysis, array analysis, and commutativity analysis.

The need for automatic parallelization in compilers is growing as clusters and other forms of distributed computing are becoming more popular just as CPU technology is trending towards higher degrees and coarser granularities of parallelism. In this paper, we review known parallelization techniques for thread level identification in programs, and argue that these same techniques may also apply to generalized coarse-grain task identification.

INTRODUCTION

Compiler technology is facing a change in today's shifting market. As distributed computing increasingly permeates the field; the software industry will place a demand on their compiler technology to parallelize software automatically. Automatically identifying opportunities for parallelization is a critical step in generating efficient code for a variety of multithreading applications and has been studied for many years [2,3,4,6,8,10,11]. Although hardware can provide some degree of fine-grained parallelism [7,9], the burden falls to compiler researchers to develop techniques that will automatically parallelize applications from the source of a program. It is important to start with a definition of exactly what is meant by distributed system. We will use the definition that Bal et al. gave in their survey of parallel programming languages:

> "a distributed system consists of multiple autonomous processors that do not share primary memory, but cooperate by sending messages over a communications network"[1].

This paper focuses on a comparative study of past and present techniques for automatic parallelization. It includes techniques such as scalar analysis, array analysis, commutativity analysis, as well as other related approaches. Our purpose is for readers to get a basic understanding of the techniques presented in this article and how they are used currently to create compilers that automatically generate parallelized applications.

Past techniques provided solutions for imperative languages like FORTRAN and C; however, these might not be enough. Past compiler research on automatic parallelization dealt with parallelizing sections of a source program with a specific system in mind. Such

Proceedings of MASPLAS'05
Mid-Atlantic Student Workshop on
Programming Languages and Systems
University of Delaware, April 30, 2005

compilers would parallelize loops and other segments of code, and introduce synchronization and message passing code to maintain correctness. Such compilers used scalar and array analysis techniques that made it difficult to use language features like pointers in a section to be parallelized automatically, a restriction that has become less practical.

Recently, a shift in paradigm began and researchers started to look into other techniques for automatically generating parallelized programs, such as commutativity analysis applied over a broader window of the source code. This type of analysis worked quite well for subsets of newer languages like C++, which enables more flexibility in today's development environments.

Whatever the language, there is a growing need for these compilers and reliable techniques must be developed and refined. However, the need for automatic parallelization in compilers is growing as clusters and other forms of distributed computing are becoming more popular just as CPU technology is trending towards higher degrees and coarser granularities of parallelism. Thus, there is a need for static compiler techniques that identify coarse-grain tasks from program source, breaking the source into independent segments, which can be executed in parallel across multiple processors.

SCALAR AND ARRAY ANALYSIS

Scalar analysis and array analysis are two techniques often used in conjunction; thus they are presented together in this paper. Traditionally, these forms of analysis are used in imperative languages like FORTRAN and C because of the nature of the analyses. Scalar analysis breaks down a program to analyze scalar variable use and the dependencies that they have. A dependency as defined by Hall et al. is "when a memory location written on one iteration of a loop is accessed (read or write) on a different iteration."[6] Scalar analysis will identify such cases that can be parallelized simply due to these dependence complications. The sections that are not identified as parallelizable are left to array analysis to parallelize them; otherwise, they will not be parallelized. In addition, scalar analysis "determines [if] parallelization may be enabled by privatization or reduction transformations." [6]

Scalar analysis is also used to check dependencies on array elements by their indices. This type of analysis is called "scalar symbolic analysis" [6], and is preformed by transforming the indices into solvable affine equations that express the indices of the array. This transformation brings the problem to a more solvable integer-programming problem, that there are a multitude of solutions available to solve the problem in a reasonable amount of time.

```
for (i = 0; i < x.length; ++i) {
  for (j = 0; j < i; ++j) {
    x[i * x.length + j] = z[j];
  }
}
```

Figure 1. Loop by Array Analysis

Unfortunately, this analysis is applicable only to specific forms of program constructs. For example, if an array is accessed in a linear form via an equation that acts as if the array were multidimensional, then this form of analysis cannot parallelize the code. Figure 1 shows a loop that follows this pattern. The loop accesses the array in a standard linear fashion; however, the access is performed by a function of two variables. This type of situation is common in computer graphics as well as in scientific applications [6].

The counterpart to scalar analysis is array analysis. One method of array analysis works on array data to find privatizable arrays. Privatization is a method that allocates a copy of the complete or working portion of the array to each parallel instance that references it as the data carries no dependencies for the segment in question. Access to the array is analyzed to determine an equation of access into the array, then, if possible, that array can be privatized. Specifically this is called array data-flow analysis. Figure 2 shows a segment of code that has a data dependence and thus cannot be privatized.

```
for (i = 0; i < x.length; ++i) {
  for (j = 1; j < x[i].length;
++j) {
    x[i][j - 1] = f (x[i][j]);
  }
}
```

Figure 2. Non-privatizable Array

In order to parallelize the code segment the loop requires a transformation of the data, if a transformation cannot be applied then the array analysis will fail to parallelize this segment of code.

These two types of analysis offer very powerful tools that parallelize code based on scalar and array variables in the loop segments of the code. However, to maximize the potential of these forms of analysis, the compiler must be able to optimize inter-procedurally, thus allowing parallel optimizations to span across function boundaries. As the example in Figure 2 illustrates, many loops call functions inside their bounds and in order for the compiler to understand the data-flow completely it must be able to optimize across the function call.

However, there are some drawbacks to using these forms of analysis. We mentioned before that these techniques are most often used in imperative languages; however, they can only support a subset of language features of a language like C. Advanced features like pointers are not available to compilers that use these analyses heavily because it is almost impossible for them to detect the dependencies on dynamic pointer-based data. Another point is that these focus heavily on parallelizing loops in the source. In an imperative setting, this is quite powerful; however, with a richer language like C++ or Java these analysis techniques might not do the trick.

COMMUTATIVITY ANALYSIS

Commutativity analysis was originally used in other areas of computer science, but this technique adapts well to using languages with more advanced feature sets. In their article, Martin Rinard and Perdo Diniz proposed using commutativity analysis with a subset of the language C++. They claim that, “the key to automatically parallelizing dynamic pointer-based computations,” is commutativity analysis [5]. This is a technique that is, “designed to automatically recognize and exploit commuting operations” [5].

They use the basic mathematical definition for commutativity, that is two operations that can be preformed in any order and still obtain the same result. In order to assure proper translation and execution, they propose some restrictions to the use of instance variables and that operations must be separable. The restrictions on instance variables include nested object instance variables cannot be directly accessed and may only be accessed via methods that have the object as a receiver [5]. “Commutativity analysis is designed to work with separable operations,” or operations that, “can be decomposed into and object section and an invocation section” [5].

All of the code that commutativity analysis will identify must be separable into these two sections. The object section performs any access into the receiver. The invocation section makes calls to operations, the receiver is not accessible in this section, nor can it be. The separability restriction appears to hinder development of code that invokes an operation that reads the receiver to then update that receiver with the newly computed value.

However, Rinard and Diniz suggest two extensions to this method that alleviate the burdens and allow for courser grain parallelization. In order to test the commutability of operation the compiler uses two conditions. These conditions are as follows:

> "the new value of each instance variable of the receiver objects of A and B must be the same after the execution of the object section of A followed by the object section of B as after the execution of the object section of B followed by the object section of A" [5].

and

> "the multiset of operations directly invoked by either A or B under the execution order A followed by B must be the same as the multiset of operations directly invoked by either A or B under the execution order B followed by A" [5].

If the two operations execute the same method with the same receiver object and the same parameter values then the two operations are considered identical. These two tests determine the commutability of all of the operations in a program. Figure 3, taken from the article by Rinard and Diniz shows a source segment that can be determined to be commutative using these two tests.

```
class Node {
  private:
    bool marked;
    int value, sum;
    Node *left, *right;
  public:
    void visit (int);
};

void Node::visit (int p) {
  sum = sum + p;
  if (!marked) {
    marked = true;
    if (left != NULL)
      left->visit (value);
    if (right != NULL)
      right->visit (value);
    }
}
```

Figure 3. Commutativity Analysis Example [5]

The method *visit* is parallelizable because it fulfills all of the criteria set forth. Breaking down what is happening in the method *visit*, one can see that although there are two recursive calls within the *visit* method, the set of all calls to the method will remain over the same set. Also there is no dependence on the member *sum*, meaning that regardless of the execution order, the *sum* member will retain the same exact value.

Using the above restrictions one can see that the member instance variables *sum*, *marked*, *left*, and *right* are all accessed within the method sum, this conforms to the restriction that none of them are directly accessed. This also conforms to the other restriction that the instance variables are only accessed by a method that has the object as a receiver. These restrictions make the language a bit more difficult to program in. For instance, Figure 4 shows a segment of source code that cannot be automatically parallelized by commutativity analysis. Note the use of instance variables, the variable *empty* is accessed from outside of its object.

```
void Calculator::calculate
(Stack s) {
  if (!s.empty) {
    value = this->operate (s,
value);
  }
}
```

Figure 4. A method that cannot be parallelized by Commutativity Analysis

Beyond that, the assignment of the instance variable value is done in a manner that is not separable. The statement:

```
value = this->operate (s, value)
```

breaks the rules of separability. The invocation section is entwined with the object section in fact the object section depends on the invocation section to perform its calculations. In today's programming environment, this can be a significant restriction.

HIGH LEVEL PARALLELIZATION

The techniques discussed so far are techniques that focus primarily on the backend of the compiler; to generate binary output that is parallelized. However, in their paper, Chow et al. make a proposal of techniques that put parallelized output into an intermediate language. Their proposal focuses on building a “portable lightweight parallel run-time library,” that parallelized programs link to [2]. This is quite different from other approaches in that they focused on a platform independent build despite that they had selected a specific test platform. They used the IBM ASTI high-level optimizer to form “the foundation” of their compiler [2]. This compiler uses the FORTRAN 90 and C languages as input.

The front end performs several steps to prepare the program for the automatic parallelization. The first step that is outlined in the process is called scalarization. This process converts FORTRAN 90 specific array statements into equivalent loops, preserving the semantic of the statements.

The second step outlined is a transformation stage. This transformation stage optimizes loops for single processor machines.

Finally, there is an interprocedural analysis and inlining stage. This stage focuses on the parallelization optimizations. An interprocedural data flow analysis is performed to enhance the parallelization results. Currently however, the interprocedural data flow module has been removed from the compiler.

This technique has been improved, with enhancements made to the optimizer for better parallelization results. Such enhancements include locality optimizations, “select iteration-reordering transformations”, and outlining [2]. They claim that locality optimization “is a fundamental step for ... SMP parallelization,” because this identifies the loops on which parallelization is attempted.

The second enhancement they added was outlining, a method that can be comparatively described as the opposite of inlining. The process includes defining regions of the program and combining them into a procedure. The claim is that the outlining process “simplifies storage management, because each thread participating in execution of the loop gets a separate copy of local variables” [2].

Another point they make is that the outlining process is used as a basis to call their library routines. The outlining process determines the parameters of the newly created procedure. The core to their work lies in the parallel run-time library. This library, “employs a join/fork”, system that controls the processes.

CONCLUSIONS AND FUTURE WORK

Each of the techniques we have presented has merit in their respective applications. Scalar and array analyses when used together provide a powerful toolkit to parallelize applications. Though in general the techniques work best for imperative languages, it may be that extensions can be made to the techniques to allow them to work for other paradigms. Beyond that, a more dynamic data based extension might be possible to allow for pointer-based computations.

Commutativity analysis worked quite well using the subset of C++; however, Rinard and Diniz indicate that commutativity analysis is quite similar, in principle, to array reduction techniques used in other parallelizing compilers. This is an area that needs to be explored more, the commonality between commutativity analysis and array reduction techniques.

Another question that arises is can these techniques be used in conjunction with each other. Currently there is no attempt to use multiple parallelizing techniques in a compiler. However, the complexity of such a task, as well as possible incompatibilities, might make it nigh impossible. Commutativity analysis imposes restrictions on the programmer, can these restrictions be removed and still maintain the ability to parallelize complex pointer-based operations. If these restrictions are removed will others be imposed in their place due to the limits of parallel computing.

Each of these techniques are applied only to a subset of all types of distributed systems, how will other types of distributed systems affect each of these algorithms. Are these techniques even valid for other types of distributed computing and what is the limit that they might have? Bal et. al discuss “workstation-LAN” and “workstation-WAN” in their article, but it remains an open question whether these techniques valid for such systems, and if they are what changes might need to be made to adapt them [1].

Whatever the future of parallelizing techniques, there definitely is a future for automatically parallelizing compilers. With continuing advancements to ILP, VLIW and multiprocessor architectures, and new manufacturing techniques such as nanotechnology which promises dramatic changes to processor design, parallel computing may be entering a new era of availability and utility. And as technology broadens and applications become more distributed, the importance of systems being able to compute in parallel at ever coarser granularities will be crucial.

REFERENCES

[1] H. Bal, J. Steiner, and A. Tanenbaum. Programming Languages for Distributed Computing Systems. In *ACM Computing Surveys*, Vol. 21. No. 3. 1989.

[2] J.-H. Chow, L. E. Lyon, and V. Sarkar. "Automatic Parallelization for Symmetric Shared-Memory Multiprocessors," in *Proceedings of CASCON*: 76-89, Toronto, ON, November 12-14, 1996.

[3] M. Cintra and, D. R. Llanos. Toward Efficient and Robust Software Speculative Parallelization in Multiprocessors. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 2003.

[4] R. CYTRON. DoAcross: Beyond vectorization for multiprocessors. In Int’l. Conf. on Parallel Processing, Aug. 1986.

[5] M. Diniz and P. Diniz. Commutativity Analysis: A New Analysis Technique for Parallelizing Compilers. In *ACM Transactions on Programming Languages and Systems*, Vol. 19, No. 6, pages 1-47, 1997.

[6] M. W. Hall, S. Amarsinghe, B. R. Murphy, S. Liao, and M. Lam. Detecting Course-Grain Parallelism using an Interprocedural Parallelizing Compiler. In *Proceedings Supercomputing ’95*, December 1995.

[7] Intel Corporation, Intel® Architecture Software Developer's Manual with Preliminary Willamette Architecture Information, manual available at http://developer.intel.com/.

[8] J. G. Steffan and T. C. Mowry. The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, Las Vegas, Nevada, February 2-4, 1998.

[9] S. Thakkar and T. Huff. "Internet Streaming SIMD Extensions," IEEE Computer: 32:26-34, 1999.

[10] M. J. Wolfe, High Performance Compilers for Parallel Computer, Addison-Wesley Publishing Company, Redwood City, California, 1996.

[11] Hans Zima, Supercompilers for Parallel and Vector Computers, ACM Press, New York, NY, 1990.