Framework For Realizing Autoparallelism Through Automatic OpenMP Code Generation

A Project Report

submitted by

RAGHESH A

in partial fulfilment of the requirements
for the award of the degree of

MASTER OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY, MADRAS.
April 2011

THESIS CERTIFICATE

This is to certify that the thesis entitled **Framework For Realizing Autoparallelism Through Automatic OpenMP Code Generation**, submitted by **Raghesh A**, to the Indian Institute of Technology, Madras, for the award of the degree of **Master of Technology**, is a bona fide record of the research work carried out by him under my supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Dr. Shankar Balachandran
Research Guide
Assistant Professor
Dept. of Computer Science and Engineering
IIT-Madras, 600 036

Place: Chennai

Date:

ACKNOWLEDGEMENTS

I would like to thank everyone who helped me.

ABSTRACT

KEYWORDS: Markov Decision Processes, Symmetries, Abstraction

Current approaches to solving Markov Decision Processes (MDPs), the de-facto standard for modeling stochastic sequential decision problems, scale poorly with the size of the MDP. When used to model real-world problems though, MDPs exhibit considerable implicit redundancy, especially in the form of symmetries. However, existing model minimization approaches do not efficiently exploit this redundancy due to symmetries. These approaches involve constructing a reduced model first and then solving them. Hence we term these as "explicit minimization" approaches.

In the first part of this work, we address the question of finding symmetries of a given MDP. We show that the problem is *Isomorphism Complete*, that is, the problem is polynomially equivalent to verifying whether two graphs are isomorphic. Apart from the theoretical importance of this result it has an important practical application. The reduction presented can be used together with any off-the-shelf Graph Isomorphism solver, which performs well in the average case, to find symmetries of an MDP. In fact, we present results of using NAutY (the best Graph Isomorphism solver currently available), to find symmetries of MDPs. We next address the issue of exploiting the symmetries of a given MDP. We propose the use of an explicit model minimization algorithm called the G-reduced image algorithm that exploits symmetries in a time efficient manner. We present an analysis of the algorithm and corroborate it with empirical results on a probabilistic GridWorld domain and a single player GridWorld Soccer domain. We also present results of integrating the symmetry finding with the explicit model minimization approach to illustrate an end-to-end approach for “Abstraction using Symmetries in MDPs”.

We then note some of the problems associated with this explicit scheme and as a solution present an approach wherein we integrate the symmetries into the solution technique implicitly. However, we should select a suitable solution technique so that the overheads due to integration do not outweigh the improvements. We validate this approach by modifying the Real Time Dynamic Programming (RTDP) algorithm and empirically demonstrate significantly faster learning and reduced overall execution time on several domains.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

IITM	Indian Institute of Technology, Madras
RTFM	Read the Fine Manual

NOTATION

r	Radius, m
α	Angle of thesis in degrees
β	Flight path in degrees

CHAPTER 1

Background

1.1 Parallelism in Programs

These days it is hard to find somebody using a computer with single-core processor. With the help of multi-core and multi-processor machines it is possible to speed up the program by mapping the sections of the program to available processors(Remark - through out this document the term processor is used interchangeably with core). This is generally termed as parallelism in programs. It is very difficult to parallelize the entire program though. The degree of parallelism is limited by certain factors which is explained later in this section. In addition this section discusses various types of parallelism and make a comparison of various approaches towards parallelism which can be applied to programs.

1.1.1 Parallelism and locality

1.1.2 Types of parallelism

1.1.3 Realizing Parallelism

The various approaches to realize parallelism are explained in this section.

POSIX Threads/Pthreads: Pthreads provides a standard interface for performing mulithreaded computation. Threads are subprocesses running with in a process. We can find many applications such as a web browser which can take advantage of multithreading. The efficiency of an application improves when it is designed with threads because they have their own stack and status. The overhead of creating a separate process can be avoided here. Resources like files are shared among threads. Though Pthreads are good alternatives for having multiple processes in a single processor machine it is very difficult to scale it to multi-core processors. Another limitation of Pthreads is programmers are required to deal with a lot of thread-specific code. The number of threads required for a computation need to be hard corded which makes it less scalable.

MPI:

OpenMP: In view of the shortcomings of POSIX threads there was an urge to formulate a new threading interface. The major objective was to overcome the burden of learning different ways for programming threads in different operating systems with in different programming languages. OpenMP is able to deal with this by a great extend. As the framework is evolved rather than its APIs, support for pragmas became the distinguished feature of OpenMP. The user has to specify only the blocks of code that need to be run as parallel. The compiler does the rest. It will take care of making the pragma annotated blocks into threads. Necessary APIs are inserted to map those threads into different cores. The example below shows usage of pragma.

```
#pragma omp parallel for
for (i = 1; i <= N; i++)
    A[i] = B[i] + c[i]
```

Another characteristic of OpenMP is that by disabling support for OpenMP the same program can be treated as single threaded. This enables easy debugging and makes the programmer's life easier.

If the developer needs more fine-grained control a small set of APIs are available in OpenMP. But in this case Pthreads could be the right choice because it provides a greater number of primitive functions. So if in applications in which threads require individual attention the appropriate choice would be Pthreads.

Ample care should be take to ensure the correctness of the program while using OpenMP pragmas. The following example illustrates that.

```
for (i = 0; i < 10; i++) {
  #pragma omp parallel for private(k)
  for(j = 0; j < 10; j++) {
    k++;
    A[i] += k;
  }
}
```

We get incorrect result if the data sharing attribute for the variable *k* is *private*. It should be *shared* to get the intended result.

Intel TBB:

1.2 Auto Parallelization

We can take the advantage of hardware support for parallelism only if the compiler has support for generating the parallel code. There are interfaces like OpenMP for developing parallel applications. But the user has to manually provide the annotations for it in the source code. This becomes a tedious task for the user and he has to ensure the correctness of the code too. This prompted researchers to explore mechanisms for finding out the parallel portions of the code without manual intervention.

It can be noticed that most of the execution time of a program is spend inside some for loop. Parallelizing compiler tries to split up a loop so that its iterations can be executed on separate processors concurrently. A dependency analysis pass is performed on the code to determine whether it can be safely parallelized. The following example illustrates this.

```
for (i = 1; i <= N; i++)
    A[i] = B[i] + c[i]
```

The analysis detects that there is no dependency between two consecutive iterations and can be safely parallelized. Consider another example

```
for (i = 2; i <= N; i++)
    A[i] = A[i-1] * 2;
```

Here a particular iteration is dependent on previous one and so its not safe to parallelize. An intelligent compiler can convert this into parallel as follows.

```
for (i = 1; i <= N; i++)
    A[i] = A[1] * 2 ** (i - 1);
```

Detecting this kind of opportunities for parallelization and applying automatic transformation is a tedious task for existing compilers. A powerful mathematical model explained in the next section act as a helping hand for the compilers to do such transformations with some restrictions applied on the input.

1.3 The Polyhedral Model

In this model the program is transformed into an algebraic representation which can be used to detect data dependencies. This representation is then converted in such a way that the degree of parallelism is improved. Polyhedral optimizations are used for many kind of memory access optimization by looking into the memory access pattern of any piece of code. Any kind of classical loop optimization techniques like tiling can be used for this purpose. The model is explained in detail in Chapter 2.

1.4 LLVM

LLVM defines a common, low-level code representation in Static Single Assignment (SSA) form, with several novel features. The LLVM compiler framework and code representation together provide a combination of key capabilities that are important for practical, lifelong analysis and transformation of programs. One of the important features of LLVM is that the output of all the transformation passes have same intermediate representation(LLVM IR), which makes the programmer to analyze it with ease.

1.5 Polly

The framework for automatic OpenMP code generation which is the main topic discussed in this document is implemented using Polly[polly], an open source[licence] compiler optimization framework that uses a mathematical representation, the polyhedral model, to represent and transform loops and other control flow structures. It is an effort towards achieving autoparallelism in programs. The transformations are being implemented in LLVM(Low level virtual machine). Polly can detect parallel loops, issue vector instructions and generate OpenMP code(focus of this document) corresponding to those loops. Polly try to expose more parallelism with the help of polyhedral model. A loop which does not look parallel can be transformed to a parallel loop and these can be vectorized or parallelize using OpenMP.

More details on LLVM and Polly can be found at chapters 3 and 4 respectively.

1.6 Manual OpenMP Code Generation

1.7 SPEC2006 Benchmarks

CHAPTER 2

The polyhedral model

There are different types optimizations that can be performed on a program to improve its performance. The optimization can be made for finding data locality and hence extracting parallelism. Starting from the early history of programming languages the internal representation of program is done with Abstract Syntax Tree(AST). Though some elementary transformation can be performed on AST it is tough to carry out complex transformations like dependency analysis among statements inside a loop. Trees are very rigid data structures to do such transformations. In this chapter a extremely powerful mathematical model which puts together analysis power,expressiveness and flexibility is explained in detail.

2.1 Program Transformations with polyhedral model

In this section some of the common program tranformations which can be realized with the assistance of polyhedral model are explained. The polyhedral model is not a normal representation of progarms when compared to the classical structure of programs(like AST) that every programmer is familiar with. But it is easier to do transformations smoothly in this model.

2.1.1 Transformation for improving data docality

The polyhedral model can detect common array accesses which improves the data locality. It is illustrated with a simple example.

```
for(i = 1; i <= 10; i++)
  A[i] = 10;


for(j = 6; j <= 15; j++)
  A[j] = 15;
```

The two loops will be represented by two polyhedrons and it can find the common array accesses starting from index 6 to 10 and the code can be transformed as follows.

```
for(i = 1; i <= 5; i++)
  A[i] = 10;


for(j = 6; j <= 15; j++)
  A[j] = 15;
```

2.2 Constant propagation through arrays

2.3 Eliminate dead loop iterations

2.4 Automatic parallelization

2.5 Vectorization

CHAPTER 3

Introduction to LLVM

3.1 What Is LLVM?

LLVM is a virtual machine infrastructure that doesnt provide any of the high-level features youd find in something like the Java or .NET virtual machines, including garbage collection and an object model.

The basic design of LLVM is an unlimited register machine (URM), familiar to most computer scientists as a universal model of computation. It differs from most URMs in two ways:

- Registers are single-assignment. Once a value for a register has been set, it cant be modified. This is a common representation in a lot of compilers, and has been since the idea was invented by an IBM researcher in 1985.
- Each register has a type associated with it.

LLVM programs are assembled from basic blocks. A basic block is a sequence of instructions with no branches.

3.2 The Intermediate Representation

The core of LLVM is the intermediate representation (IR). Front ends compile code from a source language to the IR, optimization passes transform the IR, and code generators turn the IR into native code. LLVM provides three isomorphic representations of the IR. The most common one used in examples is the assembly format, which looks roughly like an assembly language for a real machine (although with a few significant differences). A ″Hello, world″ program might look something like this:

```
@.str = internal constant [12 x i8] c"hello world\00"

define i32 @main() nounwind {
entry:
  %tmp1 = getelementptr ([12 x i8]* @.str, i32 0, i32 0)
  %tmp2 = call i32 (i8*, ...)* @printf( i8* %tmp1 ) nounwind
  ret i32 0
}
```

First is a constant string, @.str. This has two qualifiers, internal and constant, which are the equivalent of static const in C. It then has a type. The square brackets signal that its an array; in this case, an array of 12 8-bit integers. The main function doesnt contain any branches, so its a single basic block. The label entry: indicates the start of the basic block, and the final instruction, ret, indicates the end. Every basic block is terminated with some kind of flow-control instruction. The ret instruction means return; in this case, returning 0 as a 32-bit integer. The type specified by the ret instruction and the return type specified in the function definition must match, or the IR will fail to validate. Above the return instruction is a call to printf. Again, note the type signatures everywhere. The printf functions return and argument types are given explicitly, and the types of the arguments are also listed. nounwind on the end indicates that this function is guaranteed not to throw an exception, which can be used in optimization later.

The first instruction in this basic block is one that most difficult one to grasp. Most programming languages (certainly, all Algol-family languages) contain some data structures that are accessed via offsets from their starts. A lot of CPUs include complex addressing modes for dealing with them. The getelementptr instruction (often referred to as GEP) provides something that can easily map to both. The first argument is a complex type, in this case our global string variable. Note that, although the string is declared as an array type, when you reference it you actually get a pointer to that array. Our printf statement wants a pointer to an i8, but we have a pointer to an array of i8s. The remaining arguments to our GEP instruction are element offsets. The first dereferences the pointer, to give an array. The second then gets a pointer to the 0th element in the array. This instruction can get pointers to any element in an arbitrarily-complex data structure. The GEP instruction does not dereferences the pointer. The GEP instruction just calculates offsets. When given all zero arguments, as in this example, all its really doing is casting a pointer to another type, which will emit no code in the final code-generation phase. This instruction could be replaced by a cast instruction that would simply change the pointer types. Both are semantically valid in this instance, but the GEP instruction is safer because it will validate the types.

While this representation of the IR is the one youre likely to see most often, its not the most commonly used format. When generating IR, its common to use a set of C++ classes that represent it and provide convenience methods for constructing it. Intermediate values then are referenced simply as pointers to llvm::Value objects, rather than by name. Most of the time, the IR is used; but when being generated, transformed, or emitted, the C++ representation is used.

The final representation is the bitcode, a very dense binary Format used to transfer LLVM IR between components in different address spaces. When using LLVM tools connected by pipes, the bitcode is sent between them. It can also be serialized to disk and loaded later.

CHAPTER 4

Polly - Pollyhedral optmizations in LLVM

CHAPTER 5

OpenMP Code generation in Polly

Optimizations in Polly should be able to create loops that are executed in parallel, as if the user would have added some OpenMP pragmas. To achieve this, code generation needs to emit code that calls an OpenMP library to be executed in parallel. The GNU OpenMP Library(libgomp) is used for this purpose. The dependency analysis module of Polly automatically detects parallel loops(SCoPs) and are given to OpenMP code generation module. Here we generate the required libgomp library calls. The code generated is similar the code generated if the user have added OpenMP pragmas[2]. The following sections explain the steps taken towards generating the OpenMP code. The generated code is in LLVM IR format.

5.1 Generating OpenMP Library Calls

Consider the for loop below to have a basic understanding about what is to be done.

```
for (int i = 0; i <= N; i++)
  A[i] = 1 ;
```

The above for loop is detected as a parallel loop and given for OpenMP code generation. Here the following sequence of GOMP library calls with proper arguments and return types(signature) has to be generated in LLVM IR format.

- GOMP parallel loop runtime start
- subfunction
- GOMP parallel end

The code for body of the for loop is generated inside the subfunction which has the following GOMP library calls to achieve the necessary parallelism.

- GOMP loop runtime next
- GOMP loop end nowait

The signature and descriptions of each of the above functions can be found in in libgomp manual[3].

5.2 Support for inner loops

So far OpenMP code created apply only for outer loops, which is detected as SCoP. Next step is to do it for inner loops. Due to dependency issues the outer loop is not detected as SCoP, but innerloop can be safely parallelized as in the following example.

```
for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
    A[j] += i;
```

Those loops need the values of the surrounding induction variables in the OpenMP subfunction. We need to pass the values of the outer induction variables in a structure to the subfunction. This step is almost completed with some minor issues to be fixed.

CHAPTER 6

Testng with pollybench

6.1 Polybench

Intel Core 2 Duo(32 Bit OS) - Black

Intel Core 2 Duo(64 Bit OS) - Red

Intel Core i5(64 Bit OS) - Blue

AMD Engineering Sample(24 Core)(64 Bit OS) - Green

	Serial Execution	**Automatic Parallelization(Polly)**	**Manual Parallelization(GCC)**
2mm			
3mm			
atax			
lu			

Table 6.1: Performance Comparison

APPENDIX A

A SAMPLE APPENDIX

Just put in text as you would into any chapter with sections and whatnot. Thats the end of it.

Publications

1. S. M. Narayanamurthy and B. Ravindran (2007). Efficiently Exploiting Symmetries in Real Time Dynamic Programming. *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 2556–2561.

REFERENCES

Amarel, S., On representations of problems of reasoning about actions. *In* **D. Michie** (ed.), *Machine Intelligence 3*, volume 3. Elsevier/North-Holland, Amsterdam, London, New York, 1968, 131–171.

Barto, A. G., **S. J. Bradtke**, and **S. P. Singh** (1995). Learning to act using real-time dynamic programming. *Artificial Intelligence*, **72**, 81–138.

Bellman, R. E., *Dynamic Programming*. Princeton University Press, 1957.

Crawford, J. (1992). A theoretical analysis of reasoning by symmetry in first-order logic. URL `citeseer.ist.psu.edu/crawford92theoretical.html`.

Griffiths, D. F. and **D. J. Higham**, *Learning LaTeX*. SIAM, 1997.

Knoblock, C. A., Learning abstraction hierarchies for problem solving. *In* **T. Dietterich** and **W. Swartout** (eds.), *Proceedings of the Eighth National Conference on Artificial Intelligence*. AAAI Press, Menlo Park, California, 1990. URL `citeseer.ist.psu.edu/knoblock90learning.html`.

Kopka, H. and **P. W. Daly**, *Guide to LaTeX (4th Edition)*. Addison-Wesley Professional, 2003.

Lamport, L., *LaTeX: A Document Preparation System (2nd Edition)*. Addison-Wesley Professional, 1994.

Manning, J. B. (1990). *Geometric symmetry in graphs*. Ph.D. thesis, Purdue University.

Ravindran, B. and **A. G. Barto** (2001). Symmetries and model minimization of markov decision processes. Technical report, University of Massachusetts, Amherst.