An Overview of OpenMP based Automatic Parallelization Tools

Ramya Badthody Shenoy
Computer Science Department
San Jose State University
San Jose, CA 95192
408-594-5009
ramya@shenoy.me

ABSTRACT

In the fast paced world today, computer programmers are required to know everything about speeding up and optimizing an application. Since the barriers for speeding up sequential code have been reached, we now have to find other options. And this is where parallel programs come into picture. A programmer thus has to learn how to parallelize code and would require a comprehensive understanding of the workings of the entire system from both a hardware and a software perspective. Automatic parallelization is the process of converting sequential code into parallel code in order to efficiently utilize resources of a shared memory multi-core systems. Automatic parallelization tools thus present an easy alternative to the above. This paper aims at understanding the way automatic parallelization works and covers the basic techniques that OpenMP based automatic parallelization tools use. It covers a brief study of the workings of two such tools -Pluto and Par4all. These tools are analyzed using a sample program and the speedups obtained on utilizing them is compared against the sequential execution.

1. INTRODUCTION

The focus in designing every application is to make it faster and smarter at the same time. Earlier, this was achieved by improving the CPU performance and by improving hardware designs. The increases in CPU clock speed cannot be advanced any further due to heat constraints in the CPU. This has given rise to the advent of multiple processors and many-core architectures. The focus is now on packaging multiple processors on a single chip. Eventually, all devices like mobile phones, embedded and portable devices will support multi core processors.

Software developers would thus have to spend more effort in understanding the various parallel processing platforms, architectures and in addition learn how to parallelize the sequential code. The developer would need to have an in depth understanding of both software and hardware parallelism. Many times the sequential code in question would not be written by the developer himself but would be legacy code. Therefore, it becomes very difficult to manually parallelize such applications.

In such a case, the idea of software that parallelizes other software seems to be a more easier alternative to manual parallelization. Automatic parallelization aims at parallelizing sequential code with least intervention from software developers. Parallelization tools analyze the program for areas which can be parallelized and insert parallel directives into the source so that they can be run on a parallel machine.

Due to the number of different parallel architectures and machines, parallel programs can be written using a variety of parallel programming paradigms like message passing, shared memory, mixed-mode, bulk synchronous and so on. The shared memory and message passing architectures are the two most important ones[1]. The steps that a developer could follow to parallelize an application can be 1) decide on the parallel machine that will be used 2) choose the right parallel programming paradigm to parallelize the code, 3) choose the good automatic parallelization tool[1].

The rest of paper is organized as follows. In Section 2, I provide a basic explanation of OpenMP, a shared memory parallel processing paradigm. The techniques in automatic parallelization like induction variable substitution, vectorization and polyhedral transformations are discussed in Section 3. Section 4 discusses the working of the automatic parallelization tools-Pluto and Par4all. Finally, the experiments carried out using these tools and the results are discussed in Section 5.

2. OPENMP

OpenMP means Open Multiprocessing, which is a API that supports shared memory multiprocessing. It supports programming languages C, C++ and Fortran. It supports multiple Operating Systems like Windows and Linux. It consists of three distinct components - Compiler directives, Runtime Library Routines and environment variables. The directives allow the user to mark sections of the program, such as do, while or for loops, which have been recognized for parallelizing. The directives appear as comments so that the original program can be compiled and run seamlessly in a serial manner.

The onus is on the application developer to use these components. The application developer examines code and determines areas to parallelize. OpenMP itself does not carry out any check for data dependencies, data conflicts, race conditions, or deadlocks. It does not add or generate parallelization directives automatically or assist the compiler in this regard. OpenMP relies totally on user assistance. An example C program is illustrated Figure 1 in which we make use of the OMP compiler directives.

```
#include<stdlib.h>
#include<omp.h>
int main(){
#pragma omp parallel
        {
        printf("Hello World\n");
        }
        return 0;
}
```

Figure 1. OpenMP directives in C program.

The two highlighted lines are OpenMP Compiler directives. The line #pragma omp parallel implies that the segment block following it needs to be executed in parallel. We also set the environment variable OMP_NUM_THREADS to indicate how many threads should be considered by the compiler during parallel execution. The compilation and the output for the above program based on the number of threads is as shown in Figure 2.

```
$ gcc -openmp helloworld.c helloworld.out
$ export OMP_NUM_THREADS=2
$./helloworld.out
Hello World
$ export OMP_NUM_THREADS=4
Hello World
Hello World
Hello World
Hello World
```

Figure 2. Execution of parallel C program.

3. AUTOMATIC PARALLELIZATION TECHNIQUES

3.1 General Algorithm

The process of automatic parallelization consists of the following steps:

1) Scan and Identify: In this stage the source code is scanned and matched against a pre defined grammar in order to recognize tokens. The source code is segregated and loops, control statements and variables are identified.

2) Program analysis: This mainly involves data dependency analysis. The code is examined to separate out statements that are independent of each other and do not share data.

3) Schedule: The scheduler will line up all processes optimally based upon their dependencies.

4) Code Generation: After the scheduler lists all the tasks the code generator will insert constructs to be read by the scheduler upon execution. The constructs indicate which tasks are to be executed on which cores.

3.2 Dependency Analysis

In order to parallelize an application, an extensive dependency analysis needs to be carried out. Dependency analysis indicates how different parts of the program are interdependent and how changes in one may affect the other. There are two different types of dependencies.

3.2.1 Control Dependency:

It is a situation where the execution of a statement in the program depends on the output of the execution of a statement before it.

```
S1: IF R1> 3 goto L1
S2:        X=Y
S3: L1: R2=R2+5
```

Figure 3. Example of control Dependency.

3.2.2 Data Dependency:

There are 4 types of data dependency.

(i) Flow(true) data dependence: When data is written at a particular location, and then later read from that location at a later time.

```
S1: R1 = 2
S2: R2 = R1 + A
```

Figure 4. Example of Flow Dependency.

(ii) Anti dependency: When data is read at a particular location, and then later written to that location at a later time.

```
S1: R1 = R2 + A
S2: R1 = 20
```

Figure 5. Example of Anti Dependency.

(iii) Output dependence: When data is written to an array element, and then overwritten with a different value at a later time.

```
S1: R1 = 2
S2: R1 = 5
```

Figure 6. Example of Output Dependency.

(iv) Input Dependence: When data is read from an array element, and then read again at a later time.

```
S1: R1 = R1 + 2
S2: R3 = R1 + 5
```

Figure 7. Example of Input Dependency.

3.3 Loop Parallelization

Loops are the main areas where a program spends most of its computing and execution time. Thus, the automatic compiler needs to recognize the loops in the program after a careful program analysis. Loop parallelization can be carried out by the parallelizing tools only if the loops are independent i.e. the actions within the loop iterations are not referencing memory

locations across iterations and there are no data flow dependency. These are known as loop-carried dependences. Usually the outermost loop is selected to be parallelized in order to maximize amount of work done and to minimize the number of parallel instances of the loop. Inner loops are selected when the loop carried dependencies are within both outer and inner loop.

3.4 Vectorization

This is a technique where the compiler converts a scalar representation of the program into vector representations. For example, if a single operation is carried out on a set of operands one at a time, we convert this to carry out the same operations on multiple data operands at the same time in parallel.

```
R1 = A1 + B1
R2 = A2 + B2
R3 = A3 + B3
R4 = A4 + B4
```

Figure 8. Example of Vectorization.

Sometimes, these vectorization possibilities can also be recognized among loops.

3.5 Reduction Recognition

A reduction variable is a location which during each loop iteration gets updated after some commutative or associative operation is applied to the previous contents of the location along with some data. The location is updated with this new values. Consider the code segment given below. '

```
Do I = 1, N
   ...
   sum = sum + A(I)
   ...
EndDo
```

Figure 9. Example of reduction recognition[2].

Such a loop cannot be executed in parallel because there are loop carried dependency in addition with anti dependence and output dependence on the variable sum. This loop can be safely parallelized by executing this explicit loop statement in the critical section as follows.

```
DOALL I = 1, N
   ...
   begin critical section
      sum = sum + A(I)
   end critical section
   ...
EndDo
```

Figure 10. Reduction variable optimization[2].

3.6 Induction Variable Substitution

An induction variable is a location which gets updated by increasing or decreasing its previous value during each iteration of the loop. Its values can also be a linear function of another variable. For example, in the loop below j is an induction variable.

```
for( i=0; i<n; i++)
        i = j * 2;
```

Figure 11.Example of Induction variable.

The compiler recognizes such expression and replaces the relevant expressions formed from the loop indices as follows.

```
for( i=0; i<n; i++)
        i + = 2;
        ... = 40 * i
```

Figure 12. Induction variable substitution.

3.7 Polyhedral transformations

Sometimes simple loop parallelization is not sufficient to parallelize programs and since a lot of time is spent on nested loops we require a more powerful abstraction. The polyhedral model is one such technique where we view a dynamic instance of each statement in a loop as an integer point in a well defined space called the statements polyhedron. Representation in such a form allows the compiler to identify code sections which can be transformed and can optimize it further. For example, consider the following two loops.

```
for(i = 1; i <= 10; i ++)
        A[i] = 10;
for(j = 5; j <= 15; j ++)
        A[j] = 15;
```

Figure 13. Example of redundant loops.

The two loops will be represented by two polyhedrons and the this can help the compiler find the common array regions accessed. The resulting code will be as follows:

```
for(i = 1; i <= 4; i ++)
        A[i] = 10;
for(j = 5; j <= 15; j ++)
        A[j] = 15;
```

Figure 14. Optimization on loop.

The polyhedral model represents Static Control Parts(SCoP) in a program. These static ScoP's are regions of code containing affine loop nests and conditionals that are evaluated statistically. Affine loops nests are those wherein the loop bounds, the conditionals and the data access functions are defined as some affine functions

on the enclosing loop indices[3].The representation of a polyhedral model consists of three parts.

1) Iteration Domain: Each statement within a loop has several statement instances for each value of the loop. Each statement instance is associated with a iteration vector. The iteration vector of a statement S1 in figure can be written as (i,j). The set of all iteration vectors for a given statement is called the iteration domain of that statement.

```
for(i=1;i<=n; i++)
        for(j=5;j<=n;j++)
                A[i]=15; //S1
```

Figure 15. Statement Instance.

2) Schedule: This is a method to represent the order of execution of the statement since the iteration domain does not indicate itself an ordering of statements. It us a function that maps the each point in the iteration domain to a vector of time stamps. The schedule function is mainly so that we can reorder the points in the iteration domain. The loop iterations will be scheduled in the lexicographic order of their associated timestamps. [3]

3) Access function: these function returns the coordinates of each point in the iteration domain. Supposing a statement has an array access as A[i+j][i], then the access function is written as

$$F_A(i,j) = (i+j,i)$$

Figure 16. Access function.

Thus, for loop transformations, the automatic parallelization tools that use the polyhedral model follow this three step process. First the code is analyzed for dependency analysis and then represented in the polyhedral model through these three parts. The parallelized code is then generated from this model.

4. LITERATURE SURVEY OF PARALLELIZATION TOOLS

There are a number of automatic parallelization software's available for the C programming language. Some OpenMP based parallelization tools are Cetus, Par4all, Pluto and SUIF compiler which are freely available. We choose two such tools called Par4all and Pluto and study their architectures and what parallelization techniques they cover.

4.1 Pluto

Pluto is an automatic parallelization tool based on the polyhedral model. It is a source to source compiler that transforms C programs into OpenMP based compiled C code. It makes use of Loopo, a prototype implementation of loop parallelization methods and CLooG, a code generation code for compilers which use the polyhedral model. This tools recognizes complex loop next optimizations and loops where there are affine dependencies. The architectural diagram of Pluto is as shown below.

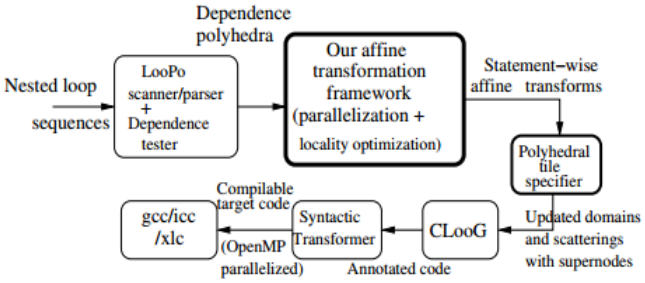

Figure 17. Architecture of Pluto[5].

Pluto produces a separate source file for a C program which has to be compiled with gcc or icc again to produce the parallelized executable.

4.2 Par4all

Par4all is an automatic parallelization and optimizing compiler for sequential programs written in C and Fortran. It is a source to source compiler which produces parallel programs capable of running on multi core systems, high performance systems and graphical processor units. It supports OpenMP, CUDA and OpenCL. It is based on PIPS (Parallelization Infrastructure for Parallel Systems) which is a framework for source-to-source for program analysis, optimization and parallelization. Par4all does array privatization, reduction variable recognition and induction variable substitution. The only limitation is that Par4all can be installed only on debian or Ubuntu platforms. The basic architecture is as shown in Figure 18. This tool directly produces a parallelized executable file.

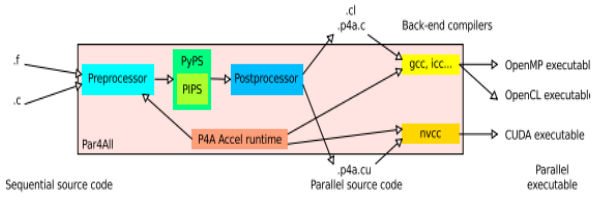

Figure 18. Architecture of Par4all[9].

5. EXPERIMENTS

The Par4all and Pluto tool were installed on a Ubuntu 13.10 Linux Operating System. A sample program for calculating the multiplication of matrices was considered and the program was run on a machine with two cores and four logical cores. The total memory available to the programs was 8GB. The matrix multiplication operation c[n][n] += a[n][n] * b[n][n] was carried out for large array values possible. The results for both the tools were compared against those of sequential execution. The resulting graphs are as shown in Figure 19 and Figure 20.

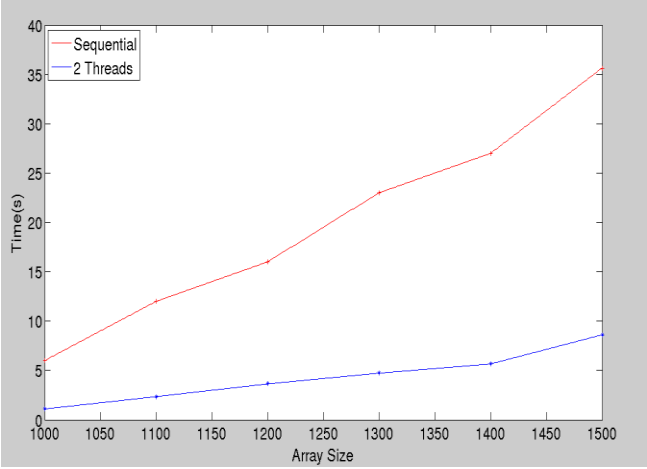

Figure 19. Graph of execution times of Par4all with sequential execution times.

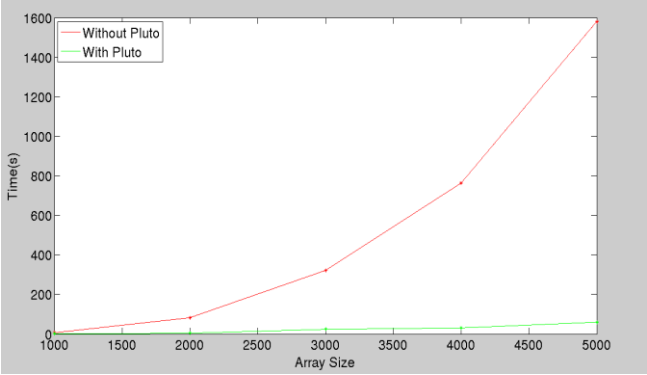

Figure 20. Graph of execution times of Pluto with sequential execution times.

5.1 OBSERVATIONS

- The Par4all tool gave significant increases over matrix sizes ranging from 1000x1000 to 1500x1500. For matrices greater than this the Par4all took more than 15s to multiply.
- For an matrices of 1100X1100, the multiplication took about 12.45s when executed sequentially. On conversion of the code to parallel code using Par4all, the multiplication took just 2.35s. The speedup achieved was 78.9412% and the reduction in time was approximately 10s.
- Pluto on the other hand gave a much better performance in comparison to Par4all. The resulting code of Pluto was tested on matrices who dimensions ranging between 1000X1000 and 5000X5000.
- On multiplying matrixes of size 5000X5000, sequential execution took approximately 26.33 minutes(1579.96s). The matrix multiplication of a 5000X5000 matrix after code parallelization using Pluto took just approximately 1 minute (59.53s). The speedup achieved was about 96.23%.
- On matrices of 1000 X 1000 elements, the Pluto's parallelized code took just 0.51s to multiply compared to the sequential execution where it took 6.21s. The resulting code from Par4all on the other hand multiplied the matrices in 1.13s.

6. CONCLUSION

Automatic parallelization provides an easy alternative to manual parallelization. It thus serves to aid developers by analyzing the various parts of the sequential code that can be parallelized. There are many techniques that like dependency analysis, induction variable substitution, reduction variable recognition and polyhedral loop transformations that are carried out by the automatic parallelization tools.

The workings of two tools Pluto and Par4all are discussed in this paper. It was found that these tools give an immense speedup in comparison to sequential programs when used to parallelize the code. Pluto was found to give a better speedup in comparison to Par4all.

7. REFERENCES

[1] Ying Qian, Automatic Parallelization Tools, Proceedings of the World Congress on Engineering and Computer Science, October 2012. Retrieved from http://www.iaeng.org/publication/WCECS2012/WCECS2012_pp97-101.pdf.

[2] William Morton Pottenger, Induction Variable substitution and Reduction Recognition in the Polaris Parallelizing Compiler, University of Illinois at Urbana-Champaign, 1995. Retrieved from http://www.dimacs.rutgers.edu/~billp/pubs/MSThesis.pdf.

[3] Kong, Veras, Stock, Frachetti, Puchet, Sadayappan, When Polyhedral Transformations Meet SIMD Code Generation, PLDI'13. Retrieved from http://www.cs.rutgers.edu/~zz124/cs671_fall2013/lectures/adarsh_simdpoly.pdf.

[4] Raghesh A, A Framework for Automatic OpenMP Code Generation, Indian Institute Of Technology, Madras. Retrieved from http://polly.llvm.org/publications/raghesh-a-masters-thesis.pdf.

[5] Uday Bondhugula, Albert Hartono, J. Ramanujam, P. Sadayappan, A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. Retrieved from http://drona.csa.iisc.ernet.in/~uday/publications/uday-pldi08.pdf.

[6] Loop Parallelism , Retrieved from http://parlab.eecs.berkeley.edu/wiki/_media/patterns/loop_parallelism.pdf.

[7] Midkiff, S. P. (2012). Automatic parallelization: An overview of fundamental compiler techniques. San Rafael,Calif.: Morgan & Claypool Publishers.

[8] PLUTO - An automatic parallelizer and locality optimizer for multicores, http://pluto-compiler.sourceforge.net/

[9] Par4all, http://www.par4all.org/

[10] Alexandra Jimborean, Adapting the Polytope Model for Dynamic and Speculative Parallelization. Retrieved from http://tel.archivesouvertes.fr/docs/00/73/38/50/PDF/thesis.pdf

[11] Open MP Architecture Review Board (2005). OpenMP Application Program Interface Version 2.5. Retrieved from http://www.openmp.org/mp-documents/spec25.pdf

[12] Automatic Parallelization Tool, Wikipedia. Retrieved from http://en.wikipedia.org/wiki/Automatic_parallelization_tool.

[13] Novillo, D. (n.d.). OpenMP and automatic parallelization in GCC, 1-3. Retrieved from http://www.airs.com/dnovillo/Papers/gcc2006.pdf