Available Online at www.ijcsmc.com

International Journal of Computer Science and Mobile Computing

A Monthly Journal of Computer Science and Information Technology

ISSN 2320–088X
IMPACT FACTOR: 5.258

IJCSMC, Vol. 5, Issue. 5, May 2016, pg.17 – 21

Automatic Parallelization: A Review

Manreet Sohal[1], Rasmeet Kaur[2]

[1]Research Scholar, Department of Computer Engineering and Technology, GNDU, Amritsar, India

[2]Assistant Professor, Depart of Computer Science and Engineering, GNDEC, Ludhiana, India

[1] reetsohal@yahoo.com; [2] rasmeetrandhawa@yahoo.com

Abstract: In this paper a comparative study of past and present techniques for automatic parallelization has been presented. It comprises of techniques like scalar analysis, commutativity analysis, array analysis and other similar approaches. The motive of this paper is provide basic understanding of the techniques of automatic parallelization and how these techniques are currently being used to generate compilers that automatically develop parallelized applications. Further the challenges faced by automatic parallelization have also been discussed.

Keywords- Automatic Parallelization, Performance, Benchmarking, Parallel Programming, analysis

I. INTRODUCTION

For parallelization, identifying opportunities automatically is a significant step in developing efficient code for a diverse number of multithreading applications and is being under study for many years. Although some degree of fine grained parallelism is provided by hardware, the load of creating techniques for automatically parallelize applications from source of a program lies with compiler researchers. As distributed computing is increasingly saturating the field, demand will be placed by the software industry on their compiler technology for automatic parallelization of the software.

By past techniques solutions were provided for imperative languages like FORTRAN and C, but in the current scenario these might not be sufficient. Earlier compiler research on automatic parallelization dealt with parallelization of the parts of a source program with keeping specific system in mind. By such compilers, loops and other segments of code could be paralleled, and would introduce synchronization and message passing code to uphold correctness. Scalar and array analysis techniques were used by such compilers thus making it difficult to utilize language features like pointers in a portion of a program to be automatically parallelized, a less practical restriction [1].

There is a raised call for these compilers regardless of the languages and the development and refinement of reliable techniques is required. However, the requirement for automatic parallelization in compilers is increasing

with the popularity of clusters and other types of distributed computing like CPU technology is drifting towards higher degrees and coarser granularities of parallelism. Therefore static compiler techniques need to be developed that recognize coarse-grain tasks from source code of the program, splitting the source code into independent fragments, which can be executed across several processors in parallel.

There are five major challenges faced by an automatic parallelizing compiler when dealing with full applications: modularity, legacy optimizations, symbolic analysis, array reshaping, and issues arising from input/output operations [2].

Many researchers from academia and industry have been working towards such automatic parallelization tool that can automatically convert a serial code into a parallel version which is functionally equivalent to it, but this has been an open challenge from past one decade. With these tools legacy serial code could be transformed into a parallel code to execute on parallel architectures. Though there are some tools that carry out this task, the industry is still searching a tool that will provide best possible improvement in performance for variety of programs. The key challenges engaged in the design and implementation of such a tool consists of checking alias variables, statement dependencies, side-effects of function calls etc. In addition to that the tool has to tackle the variety in styles of coding, length, and number of files. Taking into consideration the amount of inherent parallelism provided by the applications is also important.

II. APPROACHES TO AUTOMATIC PARALLELIZATION

A. Scalar And Array Analysis:

These types of analysis are conventionally used in imperative languages like FORTRAN and C due to the nature of these analysis. Scalar analysis split a program to analyze the use of scalar variable and to check the dependencies between these variables. 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." Scalar analysis recognizes such cases that can be parallelized merely because of the complications in these dependencies. The segments of the programs that cannot be recognized as parallelizable are left to array analysis for their parallelization or else those will not be parallelized.

Scalar analysis is also used to find out dependencies on array elements by their indices. This form of analysis is known as "scalar symbolic analysis" [6], and is carried out by changing the indices into solvable affine equations by which the indices of the array are expressed. With this transformation the problem becomes more solvable integer-programming problem, that a large number of solutions are available that can solve the problem in a reasonable amount of time.

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

unluckily, this type of analysis can be applied to only some specific forms of program constructs.

The corresponding analysis to scalar analysis is array analysis. One approach of array analysis works to search for privatizable arrays on array data. Privatization is a technique that assigns a copy of the entire or working section of the array to each parallel instance that references it because for the segment in question no dependencies are carried out by data. An array access is analyzed for determining an equation of access into the array, then, if privatization is possible, then that ray is privatized. Particularly this is known ad array data-flow analysis[1].

B. Commutativity Analysis

Commutativity analysis is planned to work with operations that can be separated or operations that can be split into object section and an invocation section. The entire code identified by commutativity analysis 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. Commutativity analysis is the key for the parallelization of dynamic pointer based computations automatically. This approach of analysis has been designed for automatically recognizing and exploiting commuting operations. The simple mathematical definition for commutativity is two operations that can be carried out in any order and still get the same result. For assuring proper translation and execution, some restrictions have been proposed for using instance variables and on the separation of the operations [1]. There are restrictions on instance variables that nested object instance variables are not directly accessible and can only be accessed through methods containing object as a receiver. The separation restriction seem

to act as hindrance in the code development that calls an operation that reads the receiver for updating that receiver with the freshly computed value.

C. High Level Parallelization

Techniques have been proposed for putting parallelized output into an intermediate language. The major focus of this proposal is on the construction of a portable lightweight parallel run-time library to which parallelized programs are linked. This is reasonably different from other techniques as in this technique focus is on a platform independent build despite of selecting a specific test platform.

Several Steps are carried out by the front end to prepare the program for the automatic parallelization. The first step that is sketched in the process is known as scalarization. The second step is a transformation stage. The loops for single processor machines are optimized in this step [1]. Finally, there is an interprocedural analysis and inlining stage. This stage emphasizes on the parallelization optimizations. An interprocedural data flow analysis is conducted to improve the results of parallelization.

III. CHALLENGES IN THE AUTOMATIC PARALLELIZATION OF LARGE SCALE COMPUTATIONAL APPLICATIONS

Mostly it is supposed that in the presence of large realistic programs, automatic parallelization is not practicable. A benchmark suite is used that has been designed particularly to demonstrate the computing requirements existing in the industry. The High Performance Group of the Standard Performance Evaluation Corporation (SPEC) provides the benchmarks. Benchmarks comprises of a seismic processing application and a quantum level molecular simulation. Both applications are present in a serial and a parallel variant. Any programming language, compiler, computer architecture, and operating system will at last have to prove that functionality and performance of applications that have commercial value can be improved by it. Such types of applications are usually large in terms of lines of code and data sets and are widely-used, and are not generally available for free. The majority of the programs that are being used to force and assess the design of new computer systems technology do not fit this definition of commercial applications. A Systems research generally makes use of benchmarks having reasonably short times of execution and is available public.

Test applications for this type of research usually incorporates suites such as the SPEC CPU, Perfect, or Linpack benchmarks. Two applications from the SPEChpc benchmark suite, called SPECseis and SPECchem have been used. Both codes are large-scale computational applications that reflect problems faced in commercial settings. There are five challenges faced by an automatic parallelizing compiler while dealing with full applications: modularity, legacy optimizations, symbolic analysis, array reshaping, and issues arising from input/output operations. In today's mid-scale benchmarks of numerical applications, parallelizing compilers are able to gain success in about half of all applications. Using compiler tools on large-scale applications we may find that this success rate is significantly less.

A. Modularity

Large-scale applications, as expected are likely to be structured into various modules. There are many reasons for it. Modularity is a general software engineering tool. It is found that large programs are the result of additions of code modifications by many software engineers over many years. In addition to that, library modules may be taken in that carry out some of the desired functionality. The full applications have profound levels of hierarchy. These consist of abstractions with interfaces to the different computational routines. Modern engineering practices may not be reflected any longer by some of the code, which is often known as legacy code. Modular programs usually elevate the compiler issue of interprocedural analysis [2]. To complicate this problem, at compile time, it is not always known which of the functions will be called during a specific execution. Furthermore, a full application package often involves code written in some different programming languages, causing a considerable challenge to the compiler.

B. Legacy Optimizations

For low-level mathematical functionality, both SPECseis and SPECchem make use of legacy code. SPECseis consists of 35 IEEE library routines to carry outFast Fourier Transformations. SPECchem consists of 63 matrix routines, a few of which were obtained from Linpack code of 1978. These codes have a tendency to be optimized for performance, but may obstruct additional compiler optimizations.

Related to this more severe problem is that hand transformations in legacy codes may have been designed for previous generations of high-performance computer systems. For today's machines the transformation may no longer be advantageous, or may even degrade performance. SPECseis consists of a number of lower-level FFT routines that date back to an IEEE Press book of 1979. These routines are optimized to carry out Fourier transforms with negligible memory requirements by writing the output to the supplied input array. With such optimizations memory-related dependences are introduced, this limits the performance that can be obtained by a parallelizing compiler. Provided modest memory needs of an application, for low-level mathematical functionality and machine resources of today, such optimizations may no longer be satisfactory [2].
If the compiler is allowed to distinguish definite legacy optimizations then the earlier optimizations could be undone and the compiler could execute its own. Another advancement would be to authorize the compiler with the ability to manage all the functions and complexities added by legacy optimizations.

C. Symbolic Analysis

At the center of a parallelizing compiler lies in its ability to discover data accesses that either access the same location or does not access . This potential entails the array subscript expressions analysis. Some compilers in current use on high-performance systems can only analyze such expressions if they are affine. Affine subscript expressions consists of linear combinations of the iteration variables of enclosing loops, where all coefficients are known, compile- time constants. The symbolic and sometimes non-linear analysis is required for dealing with the expressions found in realistic codes. If subroutines are inlined, it further increases the complexity of these expressions. Moreover, substitution technique for the induction variable, which is an essential parallelization technique, have a tendency to form non-linear expressions in states of triangular loops or in case of coupling of induction variables. Similar patterns have been realizeed in the Perfect Benchmarks. SPECseis causes another challenge to symbolic analysis. Since SPECseis depends heavily on fast Fourier transforms [2].

D. Array Reshaping And Type Change

Interprocedural analysis is a very essential approach for dealing with modular programs. The subroutine inline expansion is used by Polaris compiler to obtain the same result. A problem faced by both these approaches is that different shapes may be assumed by arrays and may have different types in a subroutine and its caller.
Array reshaping may takes when the array is shaped by caller routine as a 2D array and is shaped by the callee as 1D. Fortran compilers suppose that no out-of-bounds indexing arises by default. Another instance of array reshaping is a condition where fractions of a large array, declared in the main program of SPECseis, are passed into several subroutines[2].

A similar issue takes place when an array's type changes between the caller and callee subroutine. An example lies in SPECseis where few arrays are declared real and utilized as complex within the callee subroutines. This is so, as it is a large work array. A portion of the work array is used by one set of routines as a smaller real array and another portion as a smaller complex array.

E. Io Statements And Loop Exits

The one main cause due to which it could not be determined by Polaris that one of the major loops of SPECchem is automatic parallelization was due to an abort statement. No doubt , the execution of abort statement takes place only in rare cases, but a conditional and an exit from the loop is seen by the compiler. In such scenrio, the abort statement was hidden deep inside of subroutine calls, loop nests, and conditionals [2]. The program exits should be ignored while looking for data dependencies as the exits arise only in cases of errors, cases in which correct program execution is not valid.

IV. SOME AUTOMATIC PARALLELIZATION TOOLS

A. *PAR4ALL:* Par4All is an automatic compiler for parallelization and optimization that supports programs written in C and FORTRAN. It is based on PIPS (Parallelization Infrastructure for Parallel Systems) source-to-source compiler framework. The "p4a" is the fundamental script interface to generate parallel code from sources of the users. It obtains source files of C or FORTRAN and develops OpenMP or CUDA output to run on shared memory multicore processor or GPU respectively.

B. *CETUS:* Cetus is a tool that carries out source-to-source transformation of software programs. It also presents a basic infrastructure for writing automatic parallelization compilers or tools. Architecture of Cetus is similar to Polaris, the major difference is that the Polaris converts FORTRAN codes and Cetus is made for programs

written in C language. Cetus enables automatic parallelization by using data dependence analysis with the Banerjee-Wolfe inequalities, array and scalar privatization.

C. S2P: S2P is a tool for conversion from Sequential to Parallel developed by KPIT Cummins Info systems Ltd. India. It receives C source code as input which may contain numerous source and header files. The output code is a multi-threaded parallel code using pthreads functions and OpenMP constructs. Task parallelization as well as loop parallelization is the target of S2P tool [3].

V. CONCLUSION

As computer technologies are expanding to incorporate more and more of distributed computing, automatic parallelization in a compiler is becoming more significant. Automatic parallelization has become a vital step in developing well-organized code for a various multithreading applications. In this paper various methods of achieving automatic parallelization has been discussed. It is found that scalar and array analysis when used in combination act as a powerful tool for parallelization of applications. Commutativity analysis worked fairly well using the subset of C++.At present, there is no try of multiplying techniques of parallelization in a compiler. Many obstacles occur in the exhibition of automatic parallelization for large-scale computational applications. These problems are required to be resolved when automatic parallelization has to be demonstrated on large-scale computational applications. It is clear that parallelization is not fully automatic yet. There are obvious methods that need to be followed to allow compilers with automatic parallelization. There are many tools for demonstrating automatic parallelization. Though the parallel codes can be developed by CETUS and Par4All tools described in this paper, additional attempts are needed to optimize those codes in matters of performance. These tools should make an effort to omit the loops with smaller execution time.

REFERENCES

[1] DiPasquale, Nicholas, T. Way, and V. Gehlot. "Comparative survey of approaches to automatic parallelization." *MASPLAS'05* (2005).

[2] Armstrong, Brian, and Rudolf Eigenmann. "Challenges in the automatic parallelization of large-scale computational applications." In *ITCom 2001: International Symposium on the Convergence of IT and Communications*, pp. 50-60. International Society for Optics and Photonics, 2001.

[3] Athavale, Aditi, Priti Randive, and Abhishek Kambale. "Automatic parallelization of sequential codes using s2p tool and benchmarking of the generated parallel codes." *URL http://www. kpit. com/downloads/research-papers/automatic-parallelization-sequential-codes. pdf*.

[4] Tournavitis, Georgios, Zheng Wang, Björn Franke, and Michael FP O'Boyle. "Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping." In *ACM Sigplan Notices*, vol. 44, no. 6, pp. 177-187. ACM, 2009.

[5] Ferrandi, Fabrizio, Luca Fossati, Marco Lattuada, Gianluca Palermo, Donatella Sciuto, and Antonino Tumeo. "Automatic parallelization of sequential specifications for symmetric mpsocs." In *Embedded System Design: Topics, Techniques and Trends*, pp. 179-192. Springer US, 2007.

[6] Chan, Bryan, and Tarek S. Abdelrahman. "Run-time support for the automatic parallelization of Java programs." *The Journal of Supercomputing*28, no. 1 (2004): 91-117.

[7] Jones, Simon Peyton, and Satnam Singh. "A tutorial on parallel and concurrent programming in haskell." In *Advanced Functional Programming*, pp. 267-305. Springer Berlin Heidelberg, 2009.

