Proceedings of the First SUIF Compiler Workshop, pp. 148-152, January 11-13, 1996, Stanford University

Automatic Parallelization of Non-uniform Dependences*

Vipin Chaudhary, Jialin Ju, Laiwu Luo, Sumit Roy, Vikas Sinha and Cheng Zhong Xu
Parallel and Distributed Computing Laboratory
Wayne State University
Detroit, Michigan 48202

Venkat Konda
Mitsubishi Electric Research Laboratories
Sunnyvale, California

Abstract

This report summarizes our current experiences with Automatic Program Parallelization tools for converting sequential Fortran code for use on a multiprocessor computer. A number of such tools were evaluated, including *Parafrase*, *Adaptor*, *PAT*, *Petit* and the *SUIF* compiler package. We evaluated the suitability of such tools for parallelizing Computational Fluid Dynamics code supplied by the Army Research Laboratory, Aberdeen Proving Grounds. *SUIF* was found to be most suitable by carrying out extensive tests on a suite of test programs and a matrix multiplication program. As a result of these experiments we suggest some modifications to the existing *SUIF* toolkit for efficient parallelization of the *CFD* code. Although *SUIF* does efficient loop partitioning for uniform dependences, it cannot handle nested loops with irregular dependencies efficiently. Unlike the case of nested loops with *uniform dependencies* these will have a complicated dependence pattern which forms a *non-uniform dependence vector* set. We propose to incorporate additional passes in *SUIF* based on results from our previous research, to generate code which will handle applications with non-uniform dependences.

1 Introduction

The Parallel and Distributed Computing Laboratory at Wayne State University has been investigating issues in automatic program parallelization for the last couple of years. One phase of this research consisted of evaluating various relevant public domain tools including *Parafrase, Adaptor, PAT, Petit* and *SUIF*. Based on our experiences with these tools, we selected *SUIF* as the most suitable tool for automatically parallelizing Computational Fluid Dynamics code supplied by the Army Research Laboratory, Aberdeen Proving Grounds.

Another phase of our work is related to the problem of parallelizing loops with non-uniform or irregular dependence vectors. Unlike the case of nested loops with *uniform dependencies* these will have a complicated dependence pattern which forms a *non-uniform dependence vector* set. In [9], we have proposed a a simple way to compute minimum dependence distances. Using these minimum dependence distances the iteration space of the nested loops can be tiled. Iterations in a tile can be executed in parallel and the tiles can be executed with proper synchronization.

Hence we plan to add passes to the *SUIF* suite so that parallelism can be extracted from non-uniform dependences. These type of dependences are often found in various applications. We are also planning to construct a back-end for the *SUIF* suite to convert the intermediate format to **Fortran** instead of **C**. This

*This work was supported in part by NSF MIP-9309489, US Army Contract DAEA-32-93-D-004 and Ford Motor Company Grant #0000952185

would let application programmers get an overview of the transformations introduced by *SUIF* and allow them to add further directives as desired before passing the transformed code to the native compiler.

2 Parallelization Tools

There are a number of parallelization tools available in the public domain. We briefly evaluated the following tools for ease of use and suitability for the target code.

- *Parafrase*[1] is a parallelization tool consisting of a number of passes, including **constant propagation**, **loop interchanging**, detection of possible **doall** loops and **code generation**. If the input code is in Fortran, the final code generated is in **Cedar Fortran**, for a hierarchical-memory multiprocessor machine. Analysis is also supported for C code but there is no code generation. It was found that this tool cannot recognize independence in *linearized* subscripts.(Which formed part of the CFD kernel test suite).It is also not designed to be extended easily and the transformed code is not very portable. However a strong point of the tool is the graphic display of the flow graph, the dependence vectors and the function call graph.
- *ADAPTOR* [2] is a source-to-source translator, which takes programs written in Fortran with array extensions, parallel loops and layout directives and converts them to code with calls to an explicit message passing library. Support is provided for using PVM [5] on a cluster of workstations. However it does not accept "C" code at all.
- *PAT* [3] is a source analysis tool providing support for interactive transformation of serial Fortran code. The tool is under development and is being rewritten to adopt an object oriented approach.
- *Petit* [4] is a dependence analysis and program transformation tool based on Michael Wolfe's *Tiny* tool. It requires transformation of source programs into a Fortran like language before use.
- *SUIF* [6] is a parallel compiler research tool from Monica Lam's group at Stanford University. It is not designed to be a particularly efficient compiler in itself, rather it forms the basis for conducting research in parallelizing compilers. It offers a standard intermediate format, which can be passed through various compiler stages for doing dependence analysis, loop transformations etc. It is easily extendible and has good support. Though the current distribution has support for a limited number of machines, we were able to port most of the tool to a 4 processor SPARCserver 630MP and we wrote a runtime library to make use of Solaris kernel threads. A small FORTRAN preprocessor was also written to convert newer constructs, like the "DO"- "ENDDO" pair, to traditional Fortran 77 code ("DO line-number" - "line-number CONTINUE").

3 SUIF: Results and Analysis

The *SUIF* compiler tool-kit was chosen for this project based on the evaluations in the previous section. For this stage, we wanted to concentrate our investigations on the efficiency of the *SUIF* compiler for the code at hand. Since the performance yardstick would be the speed-up due to the compiler, we had to add some extensions to the compiler package.

After porting *SUIF* to the 4 processor SUN, we performed various timing measurements. The codes used were three test routines from ARL and a matrix multiplication program. The chief difference in the test programs was in the format of their array declarations. The matrix multiplication program was used as a basis to determine performance losses. Different problem sizes were considered to investigate scaling issues. The number of processors used was changed from one to four, by increasing the number of concurrent threads appropriately. (Individual threads will always attempt to run on every available processor.)

The matrix multiplication program consisted of two nested loops, one for initializing the elements, the next for multiplying them. The first loop was doubly nested, and the second loop was triply nested. We compared the execution times for the following versions of the program:

- A single-threaded **Fortran** version using the native compiler.

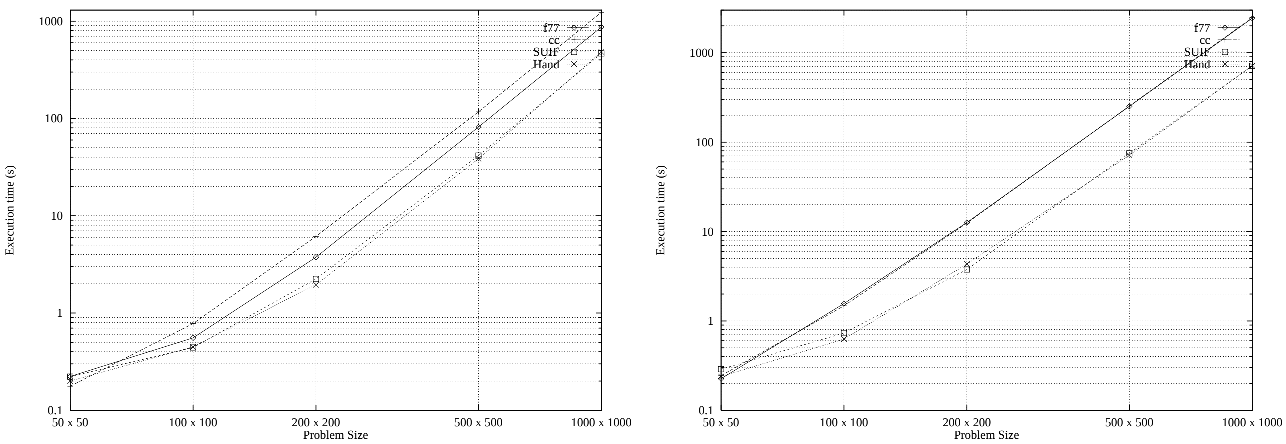
	Real/Float				Integer			
Array Size	F77	CC	SUIF	Hand	F77	CC	SUIF	Hand
50 × 50	0.223	0.177	0.222	0.200	0.266	0.244	0.289	0.239
100 × 100	0.556	0.780	0.442	0.449	1.566	1.477	0.733	0.626
200 × 200	3.752	6.134	2.241	1.954	12.625	12.416	3.766	4.329
500 × 500	81.718	116.909	41.502	38.515	251.399	253.193	74.828	71.689
1000 × 1000	869.368	1231.317	466.334	482.201	2434.053	2468.806	717.615	714.940

Table 1: Matrix Multiplication Timings

- A single-threaded C version compiled with the native **CC** compiler.
- A multi-threaded C version, coded by **hand**.
- A Fortran version passed through **SUIF** with calls to the multi-threaded runtime.

All compilations were done with maximum optimization setting for the native compiler. *Integer* and *real/float* were used to isolate language or hardware dependent behavior. We investigated scaling issues by using various array sizes.[Table 1]. We also studied the effect of increasing the number of available threads, beyond the actual number of processors.

[Fig. 1(a)] shows the relative performance of the differently compiled programs for floating point numbers and [Fig. 1(b)] for integers.



(a) Floating Point Timings

(b) Integer Timings

Figure 1: Execution timings with various compilers

As can be seen from the data, there is a performance penalty of approximately 100 % when using **C** for floating point operations as compared to **Fortran** on the SPARCserver 630 MP.

The multiprocessor machine has a very efficient floating point unit. In case of **Fortran**, the floating point version is 3 times faster than the integer code [Table 1]. The integer and float sizes on this machine were identically 32 bits, hence there is no performance loss due to differing memory requirements for the data. It is to be noted that the **Fortran** and **C** versions of the program have identical execution times when using integers.

Comparing the results of the hand coded parallel program and that of the *SUIF* version, it is seen that there is very little overhead due to calls to the *SUIF* runtime library. Allowing for statistical variation in the execution times, the results lie within 10 % of their mean. One would expect very little performance gain by manually embedding parallel routines in the target code.

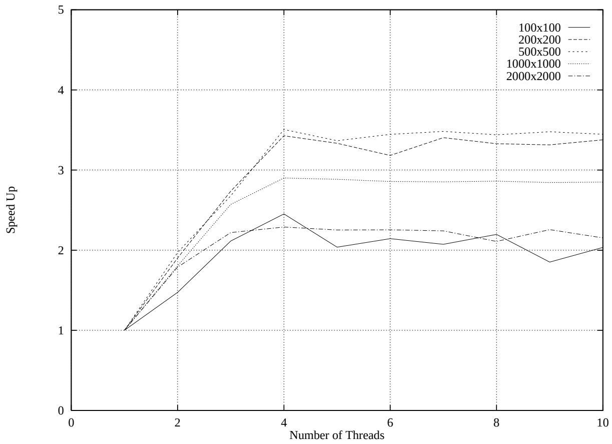


Figure 2: Speed-Ups vs Number of Threads

It is also apparent that there was a memory bottle neck on our machine, since the performance dropped for a matrix size of 1000×1000.[Fig. 2]

4 Non-uniform Dependences

According to an empirical study reported by Shen et. al. [7], subscripts that are linear functions of loop indices or coupled subscripts appear quite frequently in real programs. They observed that nearly 45% of two-dimensional array references are coupled. Coupled array subscripts in nested loops generate non-uniform dependence vectors.

To parallelize the nested loops with irregular dependence vectors, we adopt the following approach. A set of *Diophantine equations* is formed from the array subscripts of the nested loops. These Diophantine equations are solved for integer solutions. The loop bounds are applied to these solutions to obtain a set of inequalities. These inequalities are then used to form a *dependence convex hull* (DCH) as an intersection of a set of half-spaces. We use the algorithm presented by Tzen and Ni [8] to construct this dependence convex hull. Every integer point in the convex hull corresponds to a dependence vector of the iteration space. If there are no integer points within the convex hull, then there are no cross-iteration dependencies among the nested loop iterations. The corner points of this convex hull form the set of extreme points for the convex hull. These extreme points have the property that any point in the convex hull can be represented as a convex combination of these extreme points. Since the extreme points of a DCH are formed by intersections of a set of hyper-planes, they might have real coordinates. Therefore these extreme points need not be valid iterations.

We have developed an algorithm to convert these extreme points with real coordinates to extreme points with integer coordinates [9]. The main reason for doing this is that we use the dependence vectors of these extreme points to compute the minimum and maximum dependence distances. This information is otherwise not available for non-uniform dependence vector sets. Also, it can be easily proven that the dependence vectors of these extreme points form a set of extreme vectors for the dependence vector set [10]. We refer to the convex hull with all integer extreme points as *Integer Dependence Convex Hull* (IDCH). No useful dependence information is lost while changing the DCH to IDCH [10].

The dependence vectors of the extreme points form a set of extreme vectors for the dependence vector set. The minimum and maximum values of the dependence distance can be found from these extreme vectors. In [9] we have proposed a number of theorems and schemes to compute the minimum dependence distance. Using the minimum dependence distances , we can tile the iteration space.

5 Conclusions

Based on the work done so far, the *SUIF* compiler system seems to be the ideal candidate for further development into an automatic parallelization system. We have tested the current implementation on a 4 processor SPARCserver and have had some excellent results. *SUIF* is able to efficiently recognize parallelism in the code of interest and has a minimal overhead in its runtime library. Some performance penalty was however experienced due to the use of "C' as the final output language. When compared to the **Fortran** version, the maximum speed-up was never more than 2, using 4 processors. This seems to be the effect of the more efficient floating point library in case of **Fortran**. The direction for future work would be to construct a back-end for the *SUIF* suite to convert the intermediate format to **Fortran** instead of **C**. We are also planning to incorporate our work on irregular dependence analysis into the *SUIF* compiler passes.

References

[1] C. Polychronopoulos et al, "Parafrase-2 Home Page",http://www.csrd.uiuc.edu/parafrase2/index.

[2] T. Brandes, "Adaptor (HPF Compilation System)", http://www.gmd.de/SCAI/lab/adaptor/adaptor_home.

[3] W. Applebe, "Parallelization Assistant (PAT)", http://www.tc.cornell.edu/UserDoc/Software/PTools/pat.

[4] W. Pugh et al, "Petit", http://www.cs.umd.edu/projects/omega/petit.html.

[5] J. Dongarra et al, "Parallel Virtual Machine (PVM) Version 3", http://www.epm.ornl.gov/pvm/pvm_home.html

[6] M. Lam et al, "The Stanford SUIF Compiler", http://suif.stanford.edu.

[7] Z. Shen, Z. Li, and P.-C. Yew, "An empirical study on array subscripts and data dependencies," in *Proceedings of the International Conference on Parallel Processing*, pp. II–145 to II–152, 1989.

[8] T. H. Tzen and L. M. Ni, "Dependence uniformization: A loop parallelization technique," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, pp. 547 to 558, May 1993.

[9] S. Punyamurtula and V. Chaudhary, "Minimum dependence distance tiling of nested loops with non-uniform dependences," in *Proceedings of the Sixth IEEE Symposium on Parallel and Distributed Processing*, (Dallas, Texas), IEEE Computer Society Press, October 1994, pp. 74 to 81.

[10] S. Punyamurtula and V. Chaudhary, "On tiling nested loop iteration spaces with irregular dependence vectors," Tech. Rep. TR-94-02-22, Parallel and Distributed Computing Laboratory, Wayne State University, Detroit, Mar 1994.