

Code Study: Automatic Parallelism Detection in Dyfesm

Yuan Lin and David Padua
Department of Computer Science
University of Illinois at Urbana-Champaign
{yuanlin,padua}@uiuc.edu

1 Introduction

This paper reports the results of our study on the Perfect Benchmarks code DYFESM in order to determine how effective automatic parallelism detection techniques are on this code.

So far, researchers in the area of automatic parallelization have not reported any speedups for DYFESM. In fact, it is difficult to automatically find the parallelism in this code. The main reason is that indirectly accessed arrays are used extensively in DYFESM and, currently no compiler techniques can effectively detect the parallelism in many loops with indirectly accessed arrays. Our objective in studying this code is to evaluate the effectiveness of the parallelism detection techniques for sparse/irregular programs recently proposed by the authors[3].

To our knowledge, no work of this kind on DYFESM has been reported. There are, however, some important studies of this code. For more information about DYFESM, the reader is referred to [5], which presents a high-level description of the problem solved in this code. Reference [6] discusses the effectiveness of algorithm replacement and the mapping of two computation primitives that are key to the Cedar architecture. A detailed description of the structure of this code also is available in [6]. Reference [2] considers the code from the point of view of a parallelizing compiler. It does not, however, discuss how to detect the parallelism automatically by a compiler.

2 Program Structure

DYFESM “is a two-dimensional finite element code for the analysis of symmetric anisotropic structures”[5]. The code contains 7,600 lines of Fortran 77 codes in 113 subroutines. About 99.93% of total execution time is spent in a leap-frog method (subroutine LEAP) which is used to solve for the displacements and stresses, as well as velocities and accelerations at each time step. Each iteration of loop LEAP/dol0, which corresponds to one time-step, is processed as follows[5]:

1. prepare the $x_{t+\Delta t}$ for this time-step by copying it to x_t , (subroutine COPYV)
2. prepare the $\dot{x}_{t+\frac{\Delta t}{2}}$ for this time-step by copying it to $\dot{x}_{t-\frac{\Delta t}{2}}$, (subroutine COPYV)
3. solve for stresses h_t via $Fh_t = Sx_t + \frac{1}{2}M_{nl}x_t^2$, (subroutine SOLH)
4. solve for accelerations \ddot{x}_t via $M\ddot{x}_t = p_t - Sh_t - M_{nl}h_tx_t$, (subroutine SOLXDD)
5. compute velocities $\dot{x}_{t+\frac{\Delta t}{2}}$ via $\dot{x}_{t+\frac{\Delta t}{2}} = \dot{x}_{t-\frac{\Delta t}{2}} + \Delta t\ddot{x}_t$, or via $\dot{x}_{t+\frac{\Delta t}{2}} = \dot{x}_{t=0} + \frac{\Delta t}{2}\ddot{x}_{t=0}$ for the first time-step, (subroutine HOP)
6. compute displacements $x_{t+\Delta t}$ via $x_{t+\Delta t} = x_t + \Delta t\dot{x}_{t+\frac{\Delta t}{2}}$, (subroutine HOP)
7. compute $\dot{x}_t \leftarrow \frac{\dot{x}_{t+\frac{\Delta t}{2}} + \dot{x}_{t-\frac{\Delta t}{2}}}{2}$ (subroutine DRPOST)

Due to the nature of time-step methods, loop LEAP/dol0 is sequential. Each step in this loop, however, can be parallelized. Steps 1, 2, 5, 6 and 7 are simple vector operations, which are fully parallel. Steps 3 and 4 solve two linear equation systems whose parallelization methods are well studied in the literature.

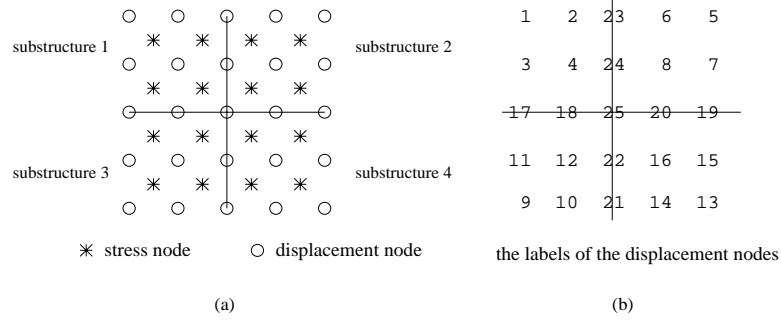


Figure 1: The Grid in the Test Problem

Though it seems trivial to find the parallelism in these steps by hand, to do so (except for the loops in COPYV) automatically is difficult because of the complicated data structure used in this code. In fact, DYFESM shows a characteristic typical to most sparse/irregular programs: although the sparse implementation has the same parallel loops as its dense counterpart, the parallelism is difficult to detect in the sparse version.

3 Data Structure

The test problem in the Perfect Benchmarks works on the grid shown in Figure 1.(a)[6]. Each of the four substructures consists of one element¹ with nine nodes for displacements and four nodes for stresses. Each displacement has five degrees of freedom and each stress has eight degrees of freedom[5]. To save space, we discuss only the data structure for the displacements here. The data structure for stresses is similar.

Figure 1.(b) shows how the twenty five displacements are labeled. Since the displacement has five degrees of freedom, each of the twenty five nodes uses a vector of five to store its value. Six arrays are used to represent this five by five grid of displacements. These arrays and their relations are illustrated in Figure 2. The value of all displacement vectors are stored in a one-dimensional array $X()$, from node 1 to node 25. Thus, $X()$ has 5×25 array elements, one for each degree of freedom of each displacement. This array is conceptually divided into five blocks. The first four blocks contain the nodes that only belong to one substructure. The fifth block contains the boundary nodes (i.e., node 17 through node 25). Because each node uses a vector of five, each of the first four blocks has 5×4 array elements, and the last block has 5×9 array elements. The starting positions and the lengths of the blocks are given by arrays $PPTR()$ and $IBLEN()$, respectively. The values in array $IWHERD(*, 2)$ give the starting position for each displacement node within a block. And, the values in array $IWHERD(*, 1)$ indicate to which block each displacement node belongs. Array $ICOND(1..9, i)$ records which displacement nodes are contained in substructure i . Among the six arrays, $X()$ stores the results of the computation and its value changes along the execution of the program. The other five arrays are *glacial*, that is, once defined, their value never changes. $IBLEN()$, $PPTR()$, and $IWHERD()$ are defined in the program, while the value of $ICOND()$ is read in from the input file. It is not difficult to see that the main data structure employed in DYFESM is a variant of the Compress Column/Row Storage format, which is commonly used in sparse/irregular programs.

As we can see, indirectly accessed arrays are used extensively in DYFESM. This is also true for most sparse/irregular programs written in FORTRAN 77, which has only one compound data structure (array) and requires the programmers to handle the memory management.

4 Code Analysis and Parallelization

We studied all the major loops ($> 8\%$ of total sequential execution time each) in DYFESM and found that all of them, except for SOLVHE/do20², can be parallelized at compile time³. All the major kernel loops can

¹The code is capable of handling multiple elements per substructure.

²Run-time information is needed to parallelize SOLVHE/do20, which accounts for 12.33% of the total sequential execution time. Run-time methods, such as the one presented in [4], can be used to parallelize this loop.

³A detailed description of each of the major loops will appear in the full version of this paper.

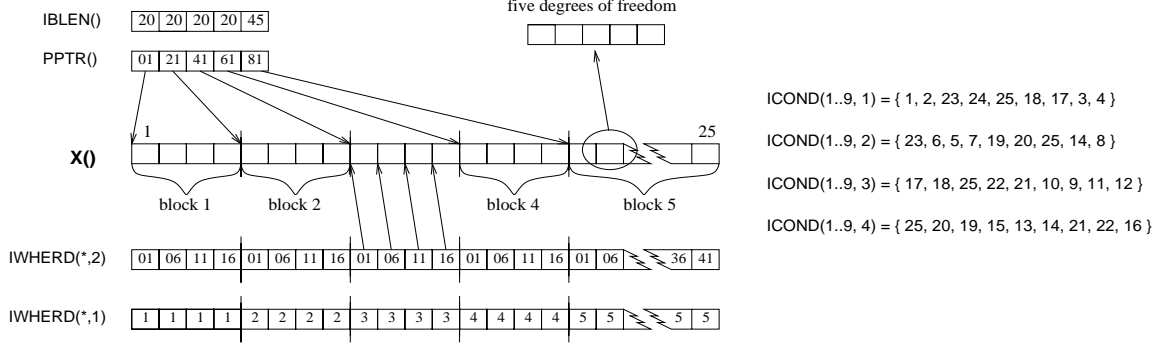


Figure 2: Displacement Nodes

```

DO 10 IBLOCK = 1, NBLOCK
  ILOC = PPTR(IBLOCK)
  NRB = IBLEN(IBLOCK)
  DO 20 I = 1, NRB
    ... = X(ILOC+I-1)
    X(ILOC+I-1) = ...
  20 CONTINUE
10 CONTINUE

```

(a)

```

IPTR = 1
DO 50 I = 1, NBLOCK
  PPTR(I) = IPTR
  IPTR = IPTR + IBLEN(I)
50 CONTINUE

```

(b)

Figure 3: Offset/length Pattern

be parallelized by using traditional techniques. To parallelize the high-level loops, subscript array analysis for *offset/length* pattern and transformation of histogram reductions are required. This result is consistent with what we found in our previous study[3] of other sparse/irregular programs.

4.1 Parallelizing Loops with *Offset/length* Pattern

A large number of loops in DYFESM have a similar program pattern, which can be called *offset/length*[3]. As discussed in Section 3, the displacement solution is stored in a one-dimensional array $X()$, which is conceptually divided into five blocks. The starting position and length of each block in $X()$ are given by $PPTR()$ and $IBLEN()$, respectively, as illustrated in Figure 2. The program uses the pattern in Figure 3.(a), where $NBLOCK$ is the number of blocks, to process the displacement data in each block.

Because different iterations of loop DO_10 in Figure 3.(a) work on different blocks, loop DO_10 can be executed in parallel. In some more complicated cases, the body of loop DO_10 involves subroutine calls and loop DO_20 appears in the called subroutines. However, the program pattern is the same. We call this pattern the *offset/length* pattern.

The key to ensuring that $do10$ is parallel is that the following equation is always satisfied $PPTR(i+1) = PPTR(i) + IBLEN(i)$, ($i = 1, ..nblock - 1$). In the program, this equation can be checked at compile-time, because array $PPTR()$ is defined in loop $SETHM/do50$ in the way shown in Figure 3.(b).

Subscript analysis is the method to get the property of an array from where the array is defined and to use this property in the analysis of loops where the array is used. This method has been proposed to solve the problem of detecting parallelism in loop with indirectly accessed arrays[1][3] and will be described in the full version of this paper.

We found this method effective in finding parallel loops in DYFESM. The operations in DYFESM that have this *offset/length* pattern are vector additions, SAXPY operations, mixed SAXPY and vector inner product operations, and preconditioning in a preconditioned conjugate gradient algorithm. These operations account for about 16% of total sequential execution time.

```

 $R_5 \leftarrow \vec{0}$ 
do i=1, 4
   $R_i \leftarrow \vec{0}$  (1)
  extract  $S_i''$  from  $S$  (2)
   $T \leftarrow M_i'' * S_i''$  (3)
   $R_i \leftarrow R_i + \text{corresponding\_part}(T, i)$  (4)
   $R_5 \leftarrow R_5 + \text{corresponding\_part}(T, 5)$  (5)
end do

```

Figure 4: MXMULT

4.2 Parallelization of Histogram Reduction in Matrix-vector Multiplication

About 63% of total sequential execution time is spent in loop MXMULT/do10, which performs a matrix-vector multiplication between a matrix M by a vector S . M and S have the following block structure:

$$M = \begin{pmatrix} M_{11} & & M_{15} \\ & M_{22} & M_{25} \\ & & \ddots & \vdots \\ M_{51} & M_{52} & \dots & M_{55} \end{pmatrix}, \text{ and } S = \begin{pmatrix} S_1 \\ S_2 \\ \vdots \\ S_5 \end{pmatrix}$$

Each M_{ij} is a sub-matrix and each S_i is a sub-vector. In DYFESM, however, this multiplication is accomplished by performing four *small* matrix-vector multiplications between M_i'' and S_i'' ($i = 1, \dots, 4$), where

$$M_i'' = P_i * M_i' * P_i^T, S_i'' = P_i * S_i', \text{ where } M_i' = \begin{pmatrix} M_{ii} & M_{i5} \\ M_{5i} & M_{55} \end{pmatrix}, S_i' = \begin{pmatrix} S_i \\ S_5 \end{pmatrix}, \text{ and } P_i \text{ is a permutation matrix}$$

For example, for substructure one, $S_1'' = (s_1, s_2, s_{23}, s_{24}, s_{25}, s_{18}, s_{17}, s_3, s_4)$, where s_1, s_2, s_3 and s_4 belong to block one, and $s_{23}, s_{24}, s_{25}, s_{18}$ and s_{17} belong to block five. The program is coded in this way because the data for M is stored in the format of M_i'' ($i = 1, \dots, 4$).

The high-level structure of MXMULT is illustrated in Figure 4⁴, where the product of multiplication $M * S$ is stored in $R = (R_1, R_2, \dots, R_5)^T$. The loop body involves nested subroutine calls.

It can be derived from this high-level structure that loop MXMULT/do10 can be parallelized. Current parallelizing compilers are able to detect the parallelism by recognizing statements (4) and (5) as a histogram reduction. However, in fact, only statement (5) performs reductions across the iterations. Although statement (4) has the form of a reduction in the implementation, it actually has no loop carried dependences. We are currently investigating how to detect this by using our subscript array analysis methods. This is important when the *privatized/expansion* method is used to transform the loops into a parallel reduction. When both statements (4) and (5) are treated as reductions, the overhead cost of the *privatized/expansion* method is $O(n^2)$, where n^2 is the number of displacement nodes. The cost is reduced to $O(n)$ when only statement (5) is treated as a reduction.

This loop also can be parallelized by putting statement (5) into a critical section. As we pointed out in our previous study[3], selecting an appropriate method to parallelize loops with histogram reduction still remains a challenge for parallelizing compilers, which is a topic we are also working on now.

4.3 Speedups

Traditional compiler techniques can only detect the kernel-level loop parallelism in DYFESM. Using the new techniques, another high-level loop parallelism can be found. We get the breakdown of sequential execution time by running the code on an SGI Power Challenge (195MHz R1000 processor) using one processor. We assume the target architecture can exploit nested levels of parallelism and ignore the cache interference. We calculated the perfect speedups that can be achieved with and without parallelizing the high-level loops and found that exploiting both levels of parallelism can generate a speedup seven times as large as that produced by only exploiting only the kernel-level parallelism.

⁴In order to be precise here, we ignore the codes that handle the case of multiple elements per substructure.

5 Summary

DYFESM is a program with many implicit parallel loops. Automatically detecting the parallelism in these loops is difficult because of the complicated data structure and the extensive number of indirectly accessed arrays used in this program. It is not impossible, however. This report reveals that two of the techniques we discussed in our previous study [3] (i.e., subscript array analysis for *offset/* pattern and histogram reduction transformation) can be used to detect the parallelism in high-level loops that account for 78% of the total sequential execution time. This result strengthens our belief that compiler techniques can be applied to automatically detect the parallelism in sparse/irregular programs.

References

- [1] W. Blume and R. Eigenmann. An overview of symbolic analysis techniques needed for the effective parallelization of the perfect benchmarks. In K. C. Tai, editor, *Proceedings of the 23rd International Conference on Parallel Processing. Volume 2: Software*, pages 233–238, Boca Raton, FL, USA, August 1994. CRC Press.
- [2] Rudolf Eigenmann. Automatic parallelization and manual improvements of the perfect club program dyfesm on alliant fx/80, fx8, and cedar. Technical report, University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, November 1991.
- [3] Y. Lin and D. Padua. On the automatic parallelization of sparse and irregular fortran programs. In *Proc. of 4th Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR98)*, volume 1511 of *Lecture Notes in Computer Science*, pages 41–56. Springer-Verlag, Pittsburgh, PA, 1998.
- [4] D. Patel and L. Rauchwerger. Principles of compiler integration of speculative run-time parallelization. In *Proceedings of the 11th International Workshop on Languages and Compilers for Parallel Computing*, pages 330–348, August 1998.
- [5] Lynn Pointer. Perfect: performance evaluation for cost-effective transformations, report 2. Technical Report CSRD 964, University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, Urbana, IL 61801, USA, March 1990.
- [6] U. Meier Yang and K. A. Gallivan. An analysis of a cedar implementation of dyfesm. Technical Report CSRD 1284, University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, Urbana, IL 61801, USA.