Automatic Parallelization of CRAY T3D MPP FORTRAN Programs

Min Feng and *Kara L. Nance*, University of Alaska, Fairbanks, Alaska

ABSTRACT: *Although parallel computers have been in existence for a number of years, the majority of code currently being utilized is sequential. One reason for this is the difficulty in converting sequential code into parallel code. To combat this problem, a compiler technique for automatically converting FORTRAN programs to CRAY T3D MPP FORTRAN form was developed. The automatic parallelization focuses on assuring both the correctness of the converted programs and the effective distribution of shared loops. Programs were tested and compared with manually parallelized programs. The results indicate that automatic parallelization is possible and that the effect can be as good as manual parallelization.*

1 Introduction

Scientists wanting to access the computational powers of parallel machines usually have to change their programming model and way of thinking to effectively and efficiently utilize the capabilities of parallelization. The effort needed to accomplish this change discourages many from taking advantage of parallelization. Unfortunately, there are no widely available effective tools to aid individuals in overcoming the learning curve, nor tools which can automatically transform traditional sequential programs to parallel form.

The CRAY T3D installed at Arctic Region Supercomputing Center (ARSC) was expected to alleviate the heavy load on the CRAY Y/MP also located there. Although ARSC has policy encouraging the use of the T3D, most individuals lack the expertise required to effectively convert serial code to parallel form. This research effort focused on alleviating the necessity of becoming an expert in parallelizing programs. The target source code is the set of FORTRAN programs which have parallel potential and which are run repeatedly, as the time requirement for parallelizing a program which is not utilized repeatedly may preclude the efficiency obtained by the tool. The result of the research effort is a prototypical tool which automatically transforms simple sequential FORTRAN programs into CRAY T3D MPP FORTRAN form, facilitating parallel processing.

2 Process

The overall phases in this research effort include the following:

- T3D FORTRAN subset parser
- data dependency analysis
- parallelizable loops selection
- communication analysis
- source code conversion

The five phases are not really discrete. All phases except the conversion phase are accomplished while parsing the source code. The data dependency analysis, parallel loops selection, and the communication analysis phase are applied to each segment of the source code individually during parsing. The methods used for these three phases are the major focus of this research effort. Each of the three primary phases will be described in detail. The other phases will be discussed in the remaining part of this section.

Since the focus of this research effort did not include creating a complete FORTRAN compiler, only the most commonly used FORTRAN statements were included. These statements represent a subset of the FORTRAN language. Minor modifications would be required to allow processing of all FORTRAN statements. The chosen subset of FORTRAN allows the declaration part of the program to contain the following statements:

- type declaration statement
- parameter statement
- array declaration statement

The executable part of the program allows the following types of statements:

- scalar and array assignments
- two forms of the DO statement

- conditional statements (IF statement, IF-THEN-ELSE statement)
- I/O statements
- other non-executable statements like FORMAT, CONTINUE

Comments are permitted anywhere in the source code. In addition, the shared data declaration statement CDIR$ SHARED array(dis) is also permitted in the declaration part. This statement does not belong to standard FORTRAN . It is a specific compiler directive of CRAY T3D MPP FORTRAN. It is included in the grammar to allow users to determine the distribution scheme of data. Data distribution was preferred, but not demanded. If no data distribution was specified, a default data distribution was set for each array.

Although data distribution determines the performance of the parallelized code, the automatic determination of the best data distribution scheme remains a theoretical research topic at this time. It is beyond the scope of practical parallelization. In this project, user's distribution scheme was utilized if provided. If no scheme was given, the same distribution scheme for all the arrays declared in the program was added to the end of declaration section. The scheme used for every dimension is block distribution (:BLOCK). This scheme equally divides one dimension of an array to all the PEs assigned to this dimension, and thus provides a possible load balance. This is no guarantee that the result is best possible distribution.

The source code in the executable section of the program was analyzed segment by segment. The beginning and end of loops are used to divide source code into segments during parsing. It is important to note that not every loop's beginning and end can be used for segment cutting. Only the outermost potentially parallelizable loop's beginning and end can be used this way. Code between two nested loops, including those before or after any loops, are cut out as segments also. For segments which contains loops, data dependence analysis and communication analysis are performed to select the best parallelization method. Segments which do not include loops are analyzed to determine which statements should be put into a sequential region. During the analysis, in addition, all necessary T3D MPP compiler directives and their appropriate locations are written out. The conversion phase combines these directives with the original source code.

3 Parallelizing Loops

The semantics of traditional FORTRAN as well as other traditionally sequential languages, specifies a linear execution order for the statements in a program. Sometimes such order can be broken without affecting the consistency and integrity of the program. When parallelizing programs, such statements should be found and converted to parallel form. The data dependency analysis phase accomplished this task.

3.1 Data Dependency Analysis

Determining data dependency is a time consuming but important task. All unnecessary data dependencies should be disapproved, because they constrain the parallelization that can be performed. It is of extreme importance not to overlook possible dependencies. Thus fast, strict, but conservative dependency tests are required. Among the existing dependency test methods, Banerjee's inequality algorithm can satisfy the above stated requirements. It has been studied extensively in literature and modified for several purposes. Banerjee's inequality algorithm was developed mainly for the purpose of optimization; however it does fit this project's need for a data dependency test. In parallel programming, data are divided into shared data and private data. This was not considered in Banerjee's algorithm. Thus some modifications were required to meet the particular needs of this research effort.

Banerjee's algorithm can be applied directly to shared arrays. For private arrays, if any private array appears at the left-hand side of an assignment statement, it will prevent all the loops outside the assignment statement from being parallelized, even though there may not be a data dependency for these loops. This occurs because every PE has a copy of the private data in its local memory. When dealing with private data, every PE uses its own copy of the private data. In the following code segment, let A be a private array. If the loop, I, is parallelized using 8 PEs, assume PE 0 is assigned iteration 1, PE 1 is assigned iteration 2, etc. After the execution of the loop, on PE 0 only A(1) is modified. All others remain unchanged. If later on, PE 0 needs to use A(2), instead of using the modified value of A(2), the old value would be used. This is not consistent with the original sequential code. Similar results can be seen for all the other PEs. In order to force every PE to modify all of its private data, we have to force the loops to be sequential.

```
DO I =1, 8
    A(I) = 1
ENDDO
```

Another case can prevent a loop from being parallelized even when there is no data dependency. For such loops, the loop index is not used in any subscripter expression of an array which appears at the left-hand side of an assignment within the loop. The following code segment demonstrates this case:

```
DO I = 1, N
  DO J = 1, M
s:    A (I ) = A(I) +1
  ENDDO
ENDDO
```

In the code , there is no data dependence for loop J according to Banerjee's algorithm. For every I, s is supposed to be executed M times and at last A(I) is increased by M. If the J loop is distributed among several PEs, every PE will try to fetch the value of A(I), add 1 to it , and then store the result back. Because

of the nondeterminism of this read/write conflict, some PEs may fetch the same old value of A(I) before another PE stores it. This could result in inaccurate results. For the above two cases, the introduction of symbol 's' to the dependency vector indicates that the corresponding loop cannot be parallelized. Also, 'x' in the dependency vector indicates that there is no dependency for that loop.

3.2 Selecting Parallel Loops

After the data dependence analysis is completed, the results include a dependency direction vector. Every entry of the vector can be one of '=','<', '>', 's', and 'x'. Based on this information, potentially parallelizable loops can be determined. The criteria are given in order of priority.

1. 's' loops must be left sequential.
2. Parallelize outer loops if both outer and inner loops can be parallelized and they are not tightly nested.
3. Select tightly nested loops to parallelize. If the loops are not tightly nested, or some loops in the nested loops must be left sequential, then choose as many tightly loops as possible from the outer ones.
4. If any loop can be proven independent, then all the nested loops can be parallelized. Independent means no references with regard to this loop access the same data.
5. All outer loops whose dependency direction are '*' and cannot be parallelized.
6. If the direction for the first parallelizable loop is '=', then this loop and all the '=' loops immediately follows it can be parallelized.
7. If the first not 's' , not '*' loop is '<' or '>', then leave this loop sequential, all inner loops can be parallelized.

If the dependency vector is (s, <, >, =, *), then all the loops after '<' can be parallelized. This criterion deserves further discussion. Without loss of generality, suppose in the following tightly nested loops, every loop index's lower bound is smaller than its upper bound.

```
DO I = I_L, I_U
   DO J = J_L, J_U
      ...
   ENDDO
ENDDO
```

No matter what the data dependency direction vector may be, the sequential semantics of the loops is: " $I_1 < I_2$, " $J_1 < J_2$, iteration (I_1, J_1) is executed before iteration (I_2, J_2). If the I loop is left sequential and the J loop is parallelized, then " $I_1 < I_2$. No matter what J1 and J2 are, iteration (I1, J1) is always executed before iteration (I2, J2). So the sequential semantics are retained as the result of the parallelized code must be the same as the sequential code.

4 Optimizing Loop Distribution

T3D MPP FORTRAN distributes the iterations of shared loops based on a shared array using the following syntax:

```
CDIR$ DOSHARED (loopindex [, loopindex, ...,
loopindex]) on array (sub [,sub, ..., sub])
```

The T3D requires that the selected array must use at most one loop index in each of the array's subscript expression, and that no division operation is allowed in the expressions. An iteration is assigned to the processor where with the loop indices value the array element is located. For example, suppose shared array A is distributed among 8 PEs in the following manner: A(1) is at PE 0, A(2) is at PE 1, etc. If the following code segment is executed by 8 PEs too,

```
CDIR$DOSHARED (I) on A(I)
DO I =1, 8
 ...
ENDDO
```

then iteration I =1 is assigned to PE 0, iteration I =2 is assigned to PE 1, etc. Knowing how a selected array is distributed, it can be determined to which processor every loop iteration is assigned. If how every shared array used in the loop is distributed is also known, comparison of the owner of a certain iteration and the owner of the array element under these loop indices determines whether the reference to the array element in that iteration is local or not. So the total number of remote accesses in the shared loops can be computed.

In a loop, usually there is more than one shared array used. Distributing the shared loop based on a different array may result in different number of remote accesses. Because remote accesses are expensive during execution, the number of remote accesses in shared loops should be minimized. This emphasizes the importance of optimizing loop distribution.

The iterations of a shared loop form the iteration space, and the processor elements form the processor space. The dominant array actually specifies a function that maps the iteration space to the processor space. The mapping may not be 1 to 1. Different iterations can be mapped to the same processor element. The T3D has some constraints on the mapping including the following: every subexpression can contain at most one loop index, and no division operation is allowed in the subexpression. These constraints can preclude some arrays used in the shared loop from being a dominant array.

The use of dominant array is for keeping data local. The mapping function assigns the loop iteration to the processor which owns the element of the dominant array of this iteration. If array A(16,8) is distributed as in figure 1, the following code segment will map the iteration space of I = 6 - 16, J =1 - 8 in the following shared loops to the processor space shown in figure 2.

```
CDIR$DOSHARED (I,J) on A(I-5, J)
    DO I =6, 16
```

```
      DO J = 1,8
    work
      ENDDO
ENDDO
```

	y =1	y =2	y =3	y =4	y=5	y=6	y=7	y=8
x=1	0	4	8	12	16	20	24	28
x=2	0	4	8	12	16	20	24	28
x=3	0	4	8	12	16	20	24	28
x=4	0	4	8	12	16	20	24	28
x=5	1	5	9	13	17	21	25	29
x=6	1	5	9	13	17	21	25	29
x=7	1	5	9	13	17	21	25	29
x=8	1	5	9	13	17	21	25	29
x=9	2	6	10	14	18	22	26	30
x=10	2	6	10	14	18	22	26	30
x=11	2	6	10	14	18	22	26	30
x=12	2	6	10	14	18	22	26	30
x=13	3	7	11	15	19	23	27	31
x=14	3	7	11	15	19	23	27	31
x=15	3	7	11	15	19	23	27	31
x=16	3	7	11	15	19	23	27	31

Figure 1: Distribution of A(16,8) using scheme (8:BLOCK(4), :BLOCK) and 32 Pes.

	J=1	J=2	J=3	J=4	J=5	J=6	J=7	J=8
I=6	0	4	8	12	16	20	24	28
I=7	0	4	8	12	16	20	24	28
I=8	0	4	8	12	16	20	24	28
I=4	0	4	8	12	16	20	24	28
I=9	1	5	9	13	17	21	25	29
I=10	1	5	9	13	17	21	25	29
I=11	1	5	9	13	17	21	25	29
I=12	1	5	9	13	17	21	25	29
I=13	2	6	10	14	18	22	26	30
I=14	2	6	10	14	18	22	26	30
I=15	2	6	10	14	18	22	26	30
I=16	2	6	10	14	18	22	26	30

Figure 2: Mapping of iterations to process elements.

Using different arrays as the dominant array, results in different mappings from iteration space to processor space. The number of remote accesses can also vary. In order to reduce the communication among processors, it is desirable to use the array which results in a minimum number of remote accesses. A "brute-force" algorithm was used for selecting such a dominant array. Although arrays not referenced in a shared loop can also be used as that shared loop's dominant array, it does not aid in reducing the number of remote accesses. So the selection is limited to the legal arrays referenced in the shared loop. Every legal array is tried. The number of remote access for the shared loop is computed for each candidate. Finally, the one with the fewest remote accesses is used as the dominant array.

Preliminary integrated tests were applied to FORTRAN programs for which manually parallelized versions exist. Programs were first converted automatically using the data distribution given in the manually parallelized version of the corresponding program. Both automatically parallelized code and manually parallelized code were compiled by CF77 and run on the T3D. Outputs from the two were compared using UNIX command *diff*. This comparison was to ensure that the output is consistent. For comparing the performance of automatically parallelized code, the timing function (irtc()) was added to both manually parallelized code and automatically parallelized code to obtain the execution time. The T3D parallel tool MPP APPRENTICE was also used to verify how time was distributed among different tasks in the two parallel versions. Programs were also converted to parallel form without given data distribution. The same comparison procedure was applied to test the correctness and the performance of the automatically parallelized program. 1, 2, 4, 8, 16, 32, and 64 processors were used in the tests.

5 Results and Conclusions

One of the test programs utilized was a neural network program called *Backpropagation*. This program had been parallelized manually. The parallel version used for comparison had been optimized carefully by its author as well. There is almost nothing to do to further improve the program's performance. The *Backpropagation* program has one hidden layer. The parameters for the program are: 32 input nodes, 64 hidden nodes, 16 output nodes, and 1000 cycles.

From the preliminary tests, it can be concluded that the method used in parallelizing programs is effective. When a data distribution is given, the automatically parallelized program can have the best performance for the certain data distribution. When a data distribution is not given, programs can be parallelized, and speedup can also be obtained for this case.

At this point, the resulting compiler tool is just a prototype. In order to develop a practical product, many modifications and enhancements remain to be made. First, the finished product can only handle a subset of FORTRAN. Also, the syntax and semantics analysis portions need to be modified to handle all aspects of FORTRAN. In this project, when doing communication analysis and selecting the best loop distribution, a "brute-force" algorithm is used. A more efficient algorithm should be developed and utilized to improve the efficiency of the mycf77 program. Since data distribution determines the best performance a parallelized program can obtain, future emphasis should be placed on determining the best data distribution for programs.

6 References

1. Agarwal, Anant, and Kranz, DAvid, Automatic Partitioning of Parallel Loops and Data Arrays for Distributed Shared Memory Multiprocessores. MIT technical report, 1993.

2. Brooks, Jeff, Single PE Optimization Techniques for the CRAY-T3D System. Benchmarking Group, Gray Research, September 1994.
3. Cray Research Inc., MPP FORTRAN Programming Model. March , 1994.
4. Cray Research Inc., Cray MPP FORTRAN Reference Manual. SR-2504 6.1.
5. Cray Research Inc., Introducing the MPP Apprentice Tool. IN-2511 1.1.
6. Fleisch, Brett Dwayne, Distributed Shared Memory in a Loosely Couples Environment. University of California, Loa Angeles, Ph.D. thesis, 1989
7. Goff, Gina, Kennedy, Ken and Tseng, Chau-Wen, Practical Dependence Testing. Rice University report CRPC-TR90103-S, November 1990.
8. Hasting, Andrew Blair, Transactional Distributed Shared Memory. Carnegie Mellon, Ph.D. Thesis, 1992.
9. Katzan, Harry Jr, FORTRAN 77. Van Nostrand Reinhold Company, 1978
10. Meltzer, Andrew, Programming for Performance in CRAFT on the T3D. Cray Research Inc., July 1994.
11. Wolfe, Michael, Optimizing Supercompilers for Supercomputers.
12. Zima, Hans, Supercompilers for Parallel and Vector Computers. Addison-Wesley Publishing Company, 1990.