A PARALLELIZING COMPILER FOR A NETWORK OF PROCESSORS

J. Mazumdar[1], D. Das[2] , B. Saha, P. P. Das and S. C. DeSarkar
Dept. of Computer Science and Engineering,
Indian Institute of Technology, Kharagpur 721302.
email: deedee,ppdas@cse.iitkgp.ernet.in

ABSTRACT

We present a working parallelizing compiler for a loosely coupled network of processors. The current compiler has been patched to a F77 front-end and a back-end generating code for a network of machines running PVM (The Parallel Virtual Machine). We present the performance of some well-known kernels using our compiler in such an environment.

1 INTRODUCTION

Distributed memory multiprocessors are increasingly being used for providing high levels of performance for scientific applications. These machines offer significant advantages over their shared memory counterparts in terms of cost and scalability, but they are much more difficult to program than shared memory machines. This is because of the absence of a single global address space. As a result, the programmer has to distribute the code and data across processors and manage communication among tasks explicitly. Clearly there is a need for a **parallelizing compiler** to relieve the programmer of this burden. The target machine we assume for developing our parallelizing compiler is a loosely coupled message passing parallel

[1]Currently with Tata Consultancy Services
[2]Supported by K. S. Krishnan Fellowship

computer. The individual processors are connected by a time-shared communication bus. We have developed a prototype compiler for such a system and run kernels on them. The input to our compiler is the source code written in Fortran 77 and the output generated are programs which run on the individual nodes/machines in a loosely synchronous manner. Data dependencies are handled with explicit **send** or **receive** calls. We have used the **PVM**[5] library calls for message passing and remote task creation. This approach of programming is known as the **SPMD** [5] approach.

The rest of the paper is organized as follows: Section 2 highlights the requirements of dependence analysis and their different approaches. Section 3 discusses our strategy of data distribution in depth. Section 4 describes the partitioning analysis. Section 5 presents the code generation strategy of our compiler. Section 6 presents the results of our study on Fortran Benchmark programs. Finally we conclude in section 7.

2 DEPENDENCE ANALYSIS

Dependence Analysis is a compile time analysis of control and memory accesses to determine the statement execution order that preserves the semantics of the original program. It is needed in our approach for determining **parallelization constraints**(as discussed in section 3) and to find out appropriate places for inserting messages. There are many approaches for solving the data dependency problem. Some of the well known tests for data dependence include **Banerjee's Test, Power test,Omega test** etc. [2],[3],[4],[6],[7]. We have implemented Power test in our work.

3 DISTRIBUTION ANALYSIS

The distribution of data across processors is of critical importance to the efficiency of parallel programs in a distributed system. Since interprocessor communication is more expensive than computation on processors, it is essential that a processor be able to do as much computation as possible using just local data. Another important consideration of a good data distribution scheme is the proper balancing of load across processors. In our work, we have followed a **constraint based approach**, proposed by other researchers. The constraint based approach tries to minimize the data communication traffic and maximally utilize the available parallelism. Thus, the optimality of the data distribution is the main goal in the presence of

constraints.
The constraint based approach [1]basically deals with two types of constraints: parallelization constraint and communication constraint. The former kind gives constraints on the distribution of the arrays , appearing on the left hand side(lhs) of an assignment statement. The distribution should be such that the array elements being assigned values in a parallelizable loop are distributed evenly on as many processors as possible. The latter tries to ensure that the data elements being read in a statement reside on the same processor.We follow the owner computes rule by which the processor responsible for a computation, owns the data being assigned a value in that computation.
For distributing the elements of arrays onto the processors, we have considered three kinds of standard distribution techniques : row-wise,column-wise,and block-wise as shown in Fig 1.

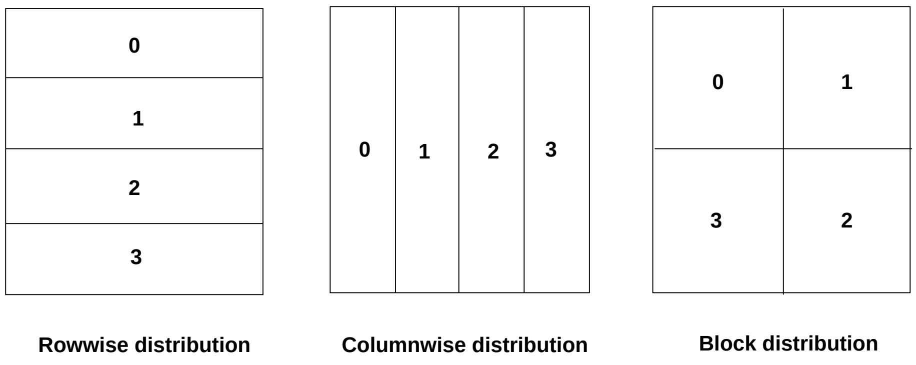

Figure 1: Some data distribution techniques

Consider the code given in Fig 2:

```
for ( i = 1; i < n; i++ )
    for ( j = 0; j < n; j++ )
        A[i][j] = A[i-1][j];
```

Figure 2: Code example

Here, if A is distributed row-wise, each processor(except that one from which distribution starts) has to access a boundary row from a processor that holds it. This results in a communication overhead. If ,however, A is column-wise distributed, this communication can be avoided as the direction vector is of the form (<,=). Similarly, if the direction vector for an array is found to be (=,<), row-wise distribution is suitable. However, in the presence of both types of direction vectors ,the conflict is resolved by simply choosing block-wise distribution . Parallelization constraint, thus, imposes a distribution on the lhs array.

For a rhs array, if the dimensions are not aligned according to the distribution of the lhs array, significant communication may result as will be shown by examples later. Hence, one of the major constraints that guide the distribution of a rhs array is its alignment with the lhs neighbour. This is termed as the communication constraint and imposes restrictions on the possible ways of data (appearing on the rhs) distribution.

Our next step of finding data distribution is to build a Component-Affinity-Graph(**CAG**)[1]. In the CAG, nodes represent the dimensions of various arrays. Two nodes are connected by an edge if the dimensions of corresponding arrays need to be aligned in order to localize data access. In such a case, a weight is associated with the edge that is equal to the communication cost if the dimensions are not aligned finally due to other conflicting constraints. Once the alignments of the various array dimensions are known, the parallelization constraints are used to choose the actual data distribution of the unconstrained data.

To build the CAG, we first determine the alignment of dimensions of various arrays used in the program. For example, in the code given in Fig 3, 1st. dimension of A is aligned with 1st. dimension of B while 1st dimension of C is aligned with 2nd dimension of D i.e. if C is row-wise distributed then D should be column-wise distributed.

```
    for ( i = 1;i < n; i++ )
        for ( j = 0; j < n; j++ ) {
            A[i][j] = B[i][j];
            C[i][j] = D[j][i];
    }
```

Figure 3: Code example

The communication cost can be calculated as follows: Suppose in the previous example A is row-wise distributed and B is column-wise distributed. Then communication will occur in the shaded portions as shown in the lhs part of Fig 4. If B is block-wise distributed then communication will occur in the shaded portions as shown in the rhs part of fig 4. We maintain a communication cost table. This table has an entry for each data distribution strategy and communication costs for each remaining strategy if the data distributions conflict.

There may be conflicts in the CAG(a pair of graph nodes is said to be conflicted if they correspond to different dimensions of the same array and there exists a path between them). In that case , we partition the CAG into d (where d is the number of array dimensions) disjoint subsets so that total weight of edges across nodes in different subsets is minimized. In addition, no two nodes of different dimensions corresponding to the same array are in the

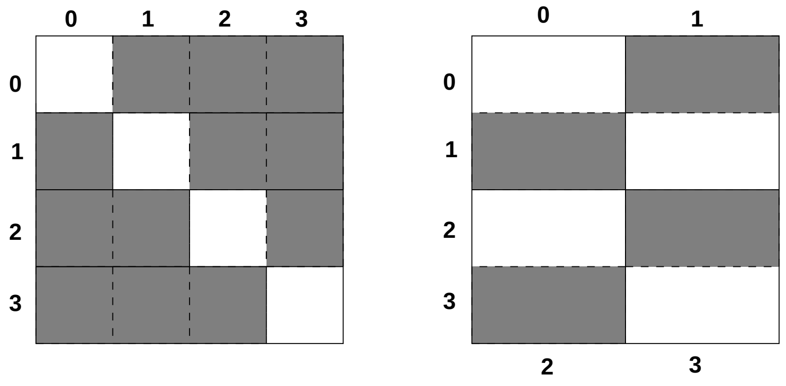

Figure 4: Data distribution with communication overhead

same subset. This corresponds to a d way cut. We have assumed the maximum dimensionality of an array to be 2,so we can partition the CAG into atmost two disjoint subsets. Nodes which are grouped into the same partition correspond to different arrays whose dimensions have to be aligned in order to reduce the communication overhead as a result of the data distribution.We have used the maximum flow (minimum cut) algorithm to partition the CAG into 2 disjoint partitions with the added property mentioned before.
Here we give a code segment in Fig 5 and its associated CAG with the partitions in Fig 6. The values are based on a configuration of 4 processors and data distribution of the form shown on the lhs part of Fig 4.

```
for ( i = 0; i < 100; i++ )
    for ( j = 0; j < 70; j++ ) {
        z[i][j] = a[i][j] + z[i][j];
        e[i]    = z[i][j];
}
for ( i = 0; i < 30; i++ ) {
    z[1][i] = d[i];
    a[1][i] = d[i];
}
for ( j = 0; j < 80; j++ ) {
    z[j][3] = d[j];
    d[j] = e[j];
}
```

Figure 5: Code example

For selecting the final distribution of each array, each assignment statement which is in a

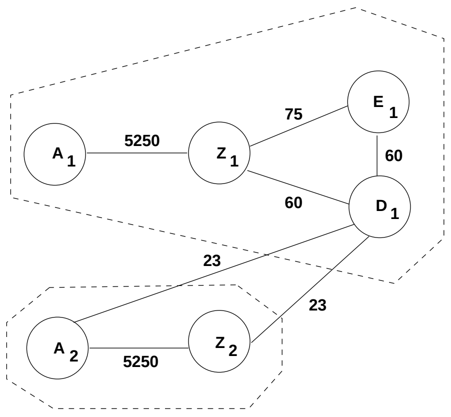

Figure 6: An example of a CAG and its associated partitions.

loop is taken. If the lhs is an array reference then each dimension of it is considered. If its distribution is not yet selected, a distribution is set according to its parallelization constraint. If any array appears in the rhs of that assignment statement, then consider each dimension of it. If it is in the same subset with respect to any lhs dimension, the distribution is set in terms of the distribution of the lhs array dimension. Finally, if a dimension of an array is not aligned with a dimension of any other array, then that dimension is replicated among processors. Consider the program for matrix multiplication:

```
for ( i = 0; i < n; i++ ) {
    for ( j = 0; j < n; j++ ) {
        C[i][j] = 0;
        for ( k = 0; k < n; k++ )
            C[i][j] = C[i][j] + A[i][k] * B[k][j];
    }
}
```

Figure 7: Code example

In this case there is no conflicted node. There is no parallelization constraint on any array. So the compiler chooses the row-wise distribution for array C. As the arrays A and C are row-wise aligned, C takes the row-wise distribution too. Also, array B has its column aligned with the column of array C.As C's columns are replicated, columns of array B are also replicated across processors. The rows of array B are not aligned with any array and hence replicated. This means that the whole array B is replicated among the processors.

4 PARTITIONING ANALYSIS

After the data distribution phase is performed, the program-partitioning analysis divides the overall data and computation among the processors. This is accomplished by first partitioning all the arrays onto processors and then using the owner computes rule to derive the functional decomposition of the program.Here, we will discuss briefly the computation of the local index set of each array used in the program and the computation of the local iteration set .

Since our compiler creates **SPMD** node programs, all processors must posses the same array declarations. This forces all processors to adopt local indices . For example, consider the following program segment :

```
for ( i = 1; i <= 100; i++ )
    A[i] = 0;
```

Figure 8: Code example

if the array A is row-wise block distributed across four processors then each processor will get 25 elements i.e. the local index for A on each processor is [1:25] , even though the equivalent global indices for A are [1:25], [26:50], [51:75] and [76:100] on processors 1 through 4 respectively. The modified program is given below :

```
for ( i = 1; i <= 25; i++ )
    A[i] = 0;
```

Figure 9: Code example

In the previous example, the original loop iterates from 1 to 100. While in the node program it iterates from 1 to 25 because each processor owns 25 elements of the array. This constitutes the local iteration set.

All loops in the program are checked . For each loop , say **L** , we check whether the loop index is used to index any array , among all the statements under the control of this loop and in that case consider the corresponding dimension of that array. If the dimension is replicated among processors , then the loop is kept unchanged. Otherwise , the upper limit and and the lower limit of the loop are modified. In the code of Fig 9, if A is a 100×100 array and it is row- wise block distributed among 4 processors, the j loop remains same but loop i changes. as shown below:

```
/* The original loop */
for ( i = lb; i < ub; i++ )
    for ( j = lb1; j < ub1; j++ )
        A[i][j] = 0;

/* The modified loop */
for ( i = max(lb,PID*t); i < min(ub,(PID+1)*t);
      i++ ) {
    /* map this index into local index */
    for ( j = lb1; j < ub1; j++ )
        A[i][j] = 0
}
```

Figure 10: Code example

where PID is the processor logical id and each processor will get t number of rows. In this example t = 100/4 = 25.

5 CODE GENERATION

Our compiler utilizes information concerning data dependence, data distribution ,local index and iteration sets to create the actual node program. In addition, issues like computing nonlocal index set, finding proper position for message insertion, introducing send/receive calls and creating the **SPMD** programs need to be looked at.

5.1 COMPUTING NONLOCAL INDEX SET

We have assumed that any array subscript expression appearing in the right hand side of an assignment statement must be of the form i+c or i-c where i is an index variable and c is any constant. For each right hand side array reference of an assignment statement, all the dimensions are considered. If a dimension is replicated , no action is taken i.e. no communication will occur along that dimension. Otherwise , the loop whose index is used in the subscript expression of that dimension is found. If the iteration set of this loop is also distributed among the processors, then communication will occur if the subscript expression of that dimension is of the form i+c where c is a constant or constant expression and i is an index variable. In this case , a particular processor will receive data from its left hand side processor or from its right hand processor depending on the sign of c. Otherwise, if the iteration set of the loop is not distributed among the processors, then a processor will receive data from more than one processor. Similarly a processor will send data to more than one processor. Consider the code in :

```
for ( k = 0; k < 10; k++ )
    for ( j = 1; j < 99; j++ )
        for ( i = 2; i < 97; i++ )
            A[i][j] = A[i-2][j] + A[i+3][j]
                    + A[i][j-1] + A[i][j+1];
```

Figure 11: Code example

After data distribution analysis is performed,array A is found to be row-wise block distributed. After local iteration set calculation the iteration set of the i loop is distributed among the processors. We consider first the right hand side reference A[i-2][j]. Here c=-2 which implies that boundary communication will occur for this reference.Similarly for the second reference boundary communication will occur ,as in this case c=3.But for both the third and fourth references, c=0 and no communication will occur in those cases.

5.2 FINDING POSITION FOR MESSAGE INSERTION

A well-known algorithm known as message vectorization is used for this purpose. It uses the level of loop-carried data dependences to calculate whether communication may be legally performed at outer loops. This replaces many small messages with one large message, reducing both message start-up cost and latency.
To vectorize messages for a rhs nonlocal index set , we examine all cross processor flow dependences with the local iteration set R as a source. The commlevel for loop carried dependence is the level of the dependence. For loop independent dependences it is defined to be the level of the deepest loop common to both the source and the sink of a dependence. The

deepest commlevel of all such dependences determines the loop level at which the message may be inserted. If the deepest commlevel is for a dependence carried by loop L , a message tag marked carried is inserted at the header of loop L. This tag indicates that nonlocal data accessed by R between iterations of loop L.
Otherwise, the deepest commlevel is for a loop independent dependence with loop L as the deepest loop enclosing both the source and the sink. A tag for R is inserted marked independent at the header of the next deeper loop enclosing R at level L+1 , or at R if no such loop exists. This tag indicates that nonlocal data accessed by R must be communicated at that point on each iteration of loop L.
In the previous example , A is row-wise block distributed and nonlocal references occur for the first and second reference. The reference A[i+3][j] has a cross-processor true dependence carried on the k loop. A tag is inserted in the k loop header. The deepest loop-carried dependence for A[i-2][j] is carried on the i loop, so we insert a tag at the i loop header.
We provide the message inserted code (not in its entirety) of the C code chunk provided before.

```
for ( k = 0; k < 10; k++ ) {
    if ( pid > 0 ) {
        send_data(part of A,pid-1); /* for the A[i+3,j] */
    if ( pid < MAX_PID )
        recv_data(part of A,pid+1); /* for the A[i+3,j] */
    for ( j = 1; j < 99; j++ ) {
        if ( pid > 0 )
           recv_data(part of A, pid-1); /* for the A[i-2,j] */
        /* block size is the size of the block for block
           distribution */
        for ( i = max(2,pid*block_size);
              i < min ( 97,(block_size+1)*pid; i++ ) {
            /* do the computation */
        }
        if ( pid < MAX_PID )
            send_data(part of A,pid+1); /* for the A[i-2,j] */
    }
}
```

Figure 12: Code example

5.3 MESSAGE GENERATION

The compiler uses the information computed in the nonlocal reference calculation to insert appropriate positions in the source code. The cross-processor dependences are coded as send or receive calls (in this case PVM calls). The send/receive may have to specify local buffers from which the data is copied if the array referencing pattern is complex. Otherwise, most send/receive primitives allow partial data copying from/to the arrays accessed. PVM supplies non-blocking sends and blocking receives. Hence, for a sequence of send/receive calls the sends are emitted first, followed by the receives. In the previous example the send_data() portion will be coded as pvm_send() and the receive_data as pvm_receive().

6 RESULTS

This compiler was tested on different programs and the speedup obtained for the some well-known kernels. The parallel versions of the programs were run using **PVM** installed on three machines- a DEC ALPHA workstation, a HP workstation and a Silicon Graphics workstation. The sequential versions were run on the DEC ALPHA workstation - the fastest of the three. The time values have been obtained using the times call.

Table 1: Jacobi's Kernel

No. of Iterations	500	1000	5000	10000
Sequential Version Time	19	38	188	379
Parallel Version Time	17	28	118	231
Speedup Obtained	1.11	1.35	1.59	1.64

This kernel was tested using a 60×60 matrix with the program being iterated a variable number of times. The results are average values over 10 runs.

Table 2: Matrix Multiplication

Size of the matrix	60×60	99×99	150×150	210×210	300×300
Sequential Version Time	10	40	144	420	1273
Parallel Version Time	13	33	97	245	699
Speedup Obtained	0.76	1.21	1.48	1.71	1.82

This program was tested using matrices of different sizes .

Table 3: Livermore Kernel

No. of Iterations	500	1000	5000	10000
Sequential Version Time	30	56	270	530
Parallel Version Time	21	30	115	210
Speedup Obtained	1.42	1.86	2.34	2.52

This program was tested using a 60×60 matrix with the program being iterated a variable number of times.

Since the sequential versions in all cases were run on the fastest machine, the speedup values obtained are all on the conservative side. The programs were tested on machines without any user load.

7 CONCLUSION

This paper presents a working parallelizing compiler for a loosely coupled network of processors. The results, tested on a network of workstations environment on an Ethernet LAN, show that even with high communication latency, sufficiently compute-intensive programs can be run with favourable speedup values. We are currently in the process of making the compiler more robust in addition to embedding run-time compilation strategies.

References

[1] P. Banerjee and et al. The PARADIGM compiler for distributed memory message passing multicomputers. In *Ist International Workshop on Parallel Processing, Bangalore, India*, pages 123–128, Dec 1994.

[2] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Boston, Mass., 1988.

[3] C. D. Polychronopoulos. *Parallel Programming and Compilers.* Kluwer Academic Publishers, Boston, Mass., 1988.

[4] W. Pugh. A practical algorithm for exact array dependence analysis. *Commun. ACM 35,8*, pages 102–115, Aug 1992.

[5] V. S. Sunderam, G. A. Geist, J. Dongarra, and R. Manchek. The PVM Concurrent Computing System: Evolution,Experiences and Trends. *Parallel Computing*, Apr 1994.

[6] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers.* Research Monographs in Parallel and Distributed Computing. MIT Press, Cambridge, Mass., 1989b.

[7] M. J. Wolfe. Data Dependence and Program Restructuring. *The Journal of Supercomputing,4.*, pages 321–344, 1990.