Proceedings of International Conference
on Computer Architecture

An Efficient Data and Computation Decomposition Technique for Nested Loops on NUMA Multiprocessor Systems*

Guan-Joe Lai, Haw-Jaw Lee and Cheng Chen
Institute of Computer Science and Information Engineering
National Chiao Tung University, Hsinchu, Taiwan, R.O.C.

Abstract

This paper presents an automatic computation/data decomposition technique for nested loops on NUMA(Non-Uniform Memory Access) systems. In NUMA systems, the remote memory access time is longer than the local one, and computation/data decomposition affects the amount of remote accesses incurred by parallel processing. Therefore, the system performance is dependent on how to decompose computation/data onto parallel processors. Here, we propose a modified locality algorithm to improve the one in [6] for the case when the decomposition is not communication-free. In addition, a new performance estimating method is also presented. The whole method has been implemented on SUIF [8]. Experimental results demonstrate the superiority of our proposed algorithm over that in previous literature.

1. Introduction

In NUMA systems, remote memory access time is longer than local one. Thus, minimizing communication by increasing the data locality is important [2-7]. Mapping computation onto processors is called the computation decomposition [2, 3, 6]; allocating data onto memory modules is called the data decomposition [1, 5, 6]. Thus, finding the optimal decomposition of programs for increasing parallelism and decreasing communication, is a key issue in parallel compilers [6, 8].

There are many related works about this problem [1-3,6,7]. The computation/data decomposition problems are NP-complete [3-5]; therefore, only heuristics were proposed. In this paper, a relax algorithm is proposed for automatically finding computation/data decomposition of nested loops on NUMA systems. Our work was based on [6]. First, the program is divided into clusters. Each cluster includes several loop nests. Loop nests within the same cluster have the same decomposition. Data redistribution only occurs between clusters. The proposed algorithm decides computation/data decomposition in clusters for loop nests and data respectively. It uses data access references to form the data locality constraints as a system of homogeneous linear equations. The system of equations can be solved by using Gaussian elimination method. When the solution is non-trivial, the decomposition is communication-free. However, when there is only the trivial solution, we relax the constraints in decreasing order of the dependence weight. After relaxing constraints, the pipeline decomposition could be found and the computation could be executed in pipeline fashion. For the case that the decomposition is not communication-free, a modified locality algorithm is proposed to decompose program computation/data. We first compute the dependence weight, and then relaxes the data locality constraints according the dependence weight in decreasing order for finding better pipeline execution. We have implemented the proposed algorithm in SUIF [8] and evaluated by our simulator [9]. Experimental results demonstrate the superiority of our proposed algorithm over that in previous literature.

The rest of this paper is organized as follows. Section 2 describes the proposed algorithm. Section 3 shows simulation environment and experimental results. Concluding remarks and future work are given finally.

2. The proposed algorithm

Our method finds the computation decomposition for loop nests. In loop nests, the loop bounds and array subscripts are affine functions of the loop indices and symbolic constants. The number of loop iterations is assumed to be much larger than the number of processors. The data decomposition is for array structures. Only the hyperplane partition is considered here, i.e., only the first-order of the decomposition. Here, we do not address the issues such as block size and load balancing.

We have implemented the proposed algorithm in SUIF, as shown in Fig. 1. The benchmarks are preprocessed by SUIF front-end processor and the common optimization analysis, then decomposed by our proposed algorithm and transferred into parallel codes.

2.1 The construction of communication graphs

We use the communication graph CG = (V, E) to

* This paper is supported by National Science Council under contract number: NSC85-2221-E-009-037

model programs and represent the relation between the clusters. The nodes in the CG correspond to loop nests. A cluster includes several loop nests. Two nodes in the CG are connected by an edge if they reference the same data array. For each node, there is a value associated with it to represent the execution time. We use the computation/data decomposition and the number of times this node executes to estimate the execution time. For each edge, there is an associated value to represent the communication time needed to redistribute the data array. Estimating the execution/communication time (the values of node and edge) is done by a cost estimation function described in subsection 2.3. The critical path length of a CG is the sum of the costs of the nodes and edges along the critical path. The proposed algorithm is aimed to label the arrays and the loop nests with decomposition such that the critical path length of CG is minimal, i.e., the execution time of the program is shortest.

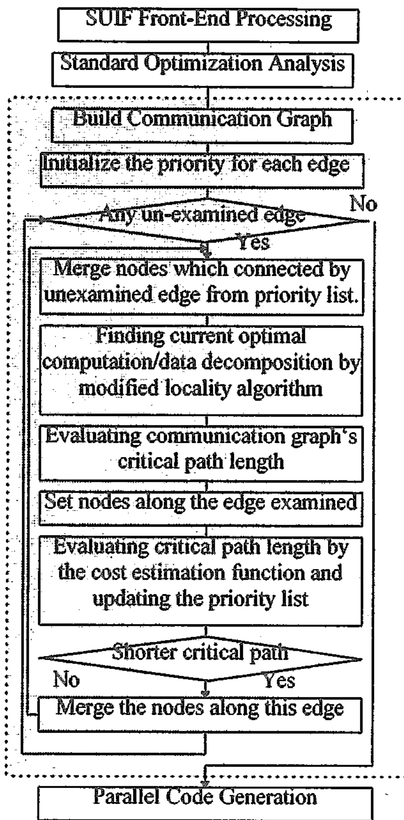


Figure 1 Flowchart of the proposed algorithm.

2.2 Modified locality algorithm

First we introduce the formal definitions of the computation and data decomposition [6].

Definition 1 For each iteration $\vec{i} = (i_1, i_2, \ldots, i_k)$ of a loop nest of depth k, the computation decomposition of the loop nest is defined by a vector $\vec{c} = (c_1, c_2, \ldots, c_k)$ and a scalar offset f_c, such that the iteration $\vec{i} = (i_1, i_2, \ldots, i_k)$ of the loop nest is mapped to virtual processor p using:

$$p = \vec{c} \cdot \vec{i}^T + f_c = c_1 * i_1 + c_2 * i_2 + \ldots + c_k * i_k + f_c \quad (1)$$

Definition 2 For each element $\vec{a} = (a_1, a_2, \ldots, a_n)$ of an n-dimensional array, the data decomposition of the array is defined by a vector $\vec{d} = (d_1, d_2, \ldots, d_n)$ and a scalar offset f_d, such that the data element $\vec{a} = (a_1, a_2, \ldots, a_n)$ of the array is mapped to virtual processor p using:

$$p = \vec{d} \cdot a^T + f_d = d_1 * a_1 + d_2 * a_2 + \ldots + d_n * a_n + f_d \quad (2)$$

No communication will occur when the data is local to the processor that references the data. This relation between the computation and data decomposition can be represented in Theorem 1.

Theorem 1 [6] Let $(\vec{c}_s, f_{cs})$ be the computation decomposition of loop nest i, and $(\vec{d}_a, f_{da})$ be the data decomposition of array A. Let F_a be an array access function for array A in loop nest i, then the reference is local to the processor if and only if

$$\vec{c}_s \cdot (i_1, i_2, \ldots, i_k)^T + f_{cs} = \vec{d}_a \cdot F_a(i_1, i_2, \ldots, i_k)^T + f_{da} \quad (3)$$

The relations between computation and data decomposition are caused by the data references. When the iteration and the data elements accessed by the iteration are mapped onto the same processor, all the memory accessed can be fulfilled in the local memory. Otherwise, there will be remote data access.

A trivial solution for Eq. (3) is $\vec{c}_s = \vec{d}_a = \vec{0}$ and $f_{cs} = f_{da} = 0$ if it maps all elements onto one processor. There is no communication if it executes the program sequentially. However, our target machine is MP system, so our objective is to find the non-trivial decomposition such that the total execution time is minimized for the whole program.

When F_a is affine function, i.e.,

$$F_a(i_1, i_2, \ldots, i_k) = f \cdot (i_1, i_2, \ldots, i_k)^T + g \quad (4)$$

where f is a constant matrix and g is a constant vector, Eq. (3) can be rewritten as :

$$(c_1, c_2, \ldots, c_k) \cdot (i_1, i_2, \ldots, i_k)^T + f_{cs} = (d_1, d_2, \ldots, d_n) \cdot (f \cdot (i_1, i_2, \ldots, i_k)^T + g) + f_{da} \quad (5)$$

Extending the Eq. (5), we get the followings:

$$\begin{aligned} & c_1 * i_1 + c_2 * i_2 \ldots + c_k * i_k + f_c = \\ & (d_1 * f_{11} + d_2 * f_{21} \ldots + d_n * f_{n1}) * i_1 \\ & + (d_1 * f_{12} + d_2 * f_{22} \ldots + d_n * f_{n2}) * i_2 \\ & + \ldots \\ & + (d_1 * f_{1k} + d_2 * f_{2k} \ldots + d_n * f_{nk}) * i_k \end{aligned}$$

$$+(d_1*g_1+d_2*g_2....+d_n*g_n)+f_d \quad (6)$$

where f_{ij} is the (i,j)th element of matrix f. The index i_1 is an independent variable in Eq. (6) which can take a range of values between its lower and upper loop bounds. The coefficients of i_1 on both sides of Eq. (6) should be equal. This also applied to other index variables. Hence, we get the set of homogeneous linear equations:

$$c_1-(d_1*f_{11}+d_2*f_{21}....+d_n*f_{n1})=0$$
$$c_2-(d_1*f_{12}+d_2*f_{22}....+d_n*f_{n2})=0$$
$$\cdots\cdots\cdots\cdots$$
$$c_k-(d_1*f_{1k}+d_2*f_{2k}....+d_n*f_{nk})=0$$
$$f_c-(d_1*g_1+d_2*g_2....+d_n*g_n)-f_d=0 \quad (7)$$

If there are more than one data reference, the system of homogeneous linear equation is constituted by union of all the equations formed by all the data references. We call the system of equations as *data locality constraints* [6]. The system of homogeneous linear equations can be solved by using the Gaussian elimination method. There are two kinds of solution when solving the system of equations. That is, non-trivial solution and trivial solution.

Case I: non-trivial solution

The iteration and the data elements accessed are mapped onto the same processor. Each processor accesses data from its local memory. It doesn't need any data from remote memories. There is no data communication between the processors. We can call this kind of decomposition as communication-free decomposition.

Example 1

```
for ( i= 0; i< N; i++)
    for( j= 0; j< N; j++)
        A[i][j]= A[i][j-1]+ 10;
```

The reference A[i][j] causes the following constraint equations:

$$c_i-d_1=0,$$
$$c_j-d_2=0,$$
$$f_c-f_d=0.$$

Similarly, the reference A[i][j-1] causes the following constraint equations:

$$c_i-d_1=0,$$
$$c_j-d_2=0,$$
$$f_c+d_2-f_d=0.$$

Unionizing this two system of constraint equations, we get the following constraint equations:

$$c_i-d_1=0,$$
$$c_j-d_2=0,$$
$$f_c-f_d=0,$$
$$f_c+d_2-f_d=0.$$

We get one solution by using the Gaussian elimination method; the computation decomposition is $(c_i,c_j)=(1,0)$, $f_c=0$ and the data decomposition is $(d_1,d_2)=(1,0)$, $f_d=0$. When we distribute the loop nest and the data using this decomposition, there will be no communication between the processors, and the processor can simultaneously execute the program, as shown in the following example.

```
forall ( i= 0; i< N; i++)
    for( j= 0; j< N; j++)
        A[i][j]= A[i][j-1]+ 10;
```

Case II: trivial solution

This is the case that we must avoid. Let us see an example that can only find the trivial solution.

Example 2

```
for ( i= 0; i< N; i++)
    for( j= 0; j< N; j++)
        A[i+1][j]= A[i][j-1]+ A[i][j]+ 10;
```

There are three data references: A[i][j], A[i][j-1] and A[i+1][j]. Each reference forms a system of constraint equations. Therefore we get the following constraint equations:

$$c_i-d_1=0,$$
$$c_j-d_2=0,$$
$$f_c-f_d=0,$$
$$f_c-d_1-f_d=0,$$
$$f_c+d_2-f_d=0.$$

We can only get the trivial solution by using the Gaussian elimination method: $(c_i,c_j)=(0,0)$, $f_c=0$ and $(d_1,d_2)=(0,0)$, $f_d=0$. This is because that two data references A[i][j] and A[i+1][j] have data dependence, as well as another two data references A[i][j-1] and A[i+1][j]. Thus, we should map all data and program onto the same processor to guarantee no communication. But this trivial solution is contradict to our objective. We should try to relax some constraints so that we can find the non-trivial solution. The non-trivial solution found in this case is not the same with case I. In case I, the non-trivial solution means communication-free decomposition. In case II, the non-trivial solution means pipeline decomposition. Although the data and the program are distributed onto each processor, each processor not only accesses the data in its local memory, but also it needs the data from remote memories. There will be some data communications and synchronization between different processors during execution. We can execute the program in pipeline fashion. It is better than that mapping all data and program in one processor and execute the program sequentially.

So the problem is that how to choose constraints to relax for solving the Eq.(7), if only the trivial solution can be found. Here we introduce the *dependence weight* to decide the constraint relaxing order. Before the formal definition of dependence weight, we see an example first.

Example 3
```
for ( i= 0; i< N; i++)
    for( j= 0; j< N; j++)
        A[j][i]= A[j][i-1]+ A[j][i-2]- A[j-1][i];
```

Considering the data reference's constraints, we get the following system of equations:

$$c_j - d_1 = 0,$$
$$c_i - d_2 = 0,$$
$$f_c - f_d = 0,$$
$$f_c + d_1 - f_d = 0,$$
$$f_c + d_2 - f_d = 0,$$
$$f_c + 2d_2 - f_d = 0.$$

We can find the trivial solution only. The iteration space of example 3 is shown in Fig. 2(A). The arrow means data dependence. There are two references related to the second dimension of array A, A[j][i-1] and A[j][i-2]. Because A[j][i] is written for each iteration, there are two different data dependence vectors in the i-axis, as shown in Fig. 2(A). For the first dimension of array A, there is only one reference, A[j-1][i], and there is one data dependence vector in the j-axis. In the iteration space, when two nodes have data dependence relation, it will have data communication after distributing two nodes into two processors.

We combine two dependence vectors in the i-axis, as shown in Fig. 2(B). The bold arrow means that it has more communication volume than thin one. Therefore, when we partition the iteration space in column-major, there will be less communication volume than partition in row-major way. Fig. 2(C) is the result of column-major partition. The node in the gray part is distributed into the same processor, so all the data references in the i-axis become local memory access. Now, the formal definition of the dependence weight is introduced.

Definition 3 The loop dependence weight is the degree that a loop affects data locality constraints. It is related to the number of loop-carried dependence with the loop index and the number of execution times of the iteration. The dependence weight can be defined as follows.

Dependence weight for loop L =

$$\sum_{\text{the number of L's iteration}} \text{the number of L's loop-carried dependence}$$

We can say that the dependence weight means the volume of communication related to the loop. Thus, if we keep the data locality constraints caused by the loop with high dependence weight, and relax the constraints caused by the loop with small dependence weight first, we can find the decomposition with less remote data access. In NUMA system, remote memory access time is longer than local one. We must decrease the number of remote data access to avoid the overhead of data communication. When we determine the partition, we can count the number of data accesses, and group most of the data accesses to the same part, let them all be distributed to the same processor's local memory.

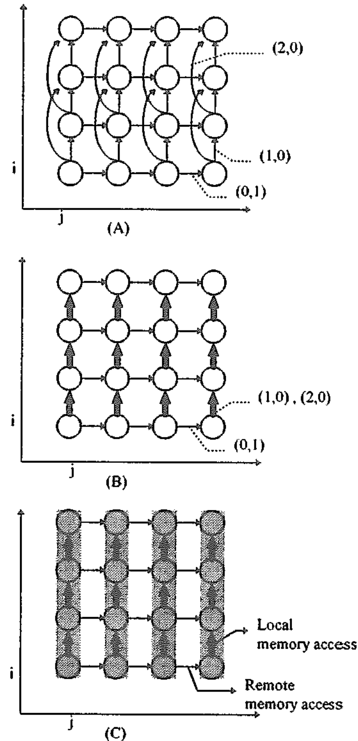


Fig. 2 Iteration space of example 3.

As shown in example 3, the loop i's dependence weight is 2*N, and the loop j's dependence weight is N. Then we choose to relax the loop with small dependence weight first, i.e., the loop j. The relaxation ignores these loop-carried dependencies by the following techniques. For example, the reference A[j-1][i], the loop j's computation decomposition equation is $c_j - d_1 = 0$. In this equation, there can be only one computation decomposition variable, c_j. The others are all data decomposition variables. In this case, only d_1 has a non-zero coefficient, meaning that the first dimension of A is bounded to the decomposition of loop j. Then, we look at the last equation raised by this reference, which involves the offsets, namely $f_c + d_1 - f_d = 0$. In the equation, if the coefficient of d_1 is not zero, then the relaxation changes it to zero. The effect of this is to, while keeping the relation between the c_j and d_1, disengage the relation of d_1 with the offsets, which is considered less important compared to other loops.

After all the relaxations have been done for the equations, the new system of equations is the following:

$$c_j - d_1 = 0,$$
$$c_i - d_2 = 0,$$
$$f_c - f_d = 0,$$
$$f_c + d_2 - f_d = 0,$$
$$f_c + 2d_2 - f_d = 0.$$

Solve it by using Gaussian elimination method, we can find the decomposition shown in Fig. 2(C).

When the loop we relax is in the inner loop of the loop nests, we should perform loop interchange to move it to the outer loop to reduce the communication overhead, if loop interchange is legal. The steps of the modified locality algorithm are described in the following.

Modified locality algorithm

Input: A cluster of loop nests
Output: Computation and data decomposition of each loop nest and data in this cluster
Begin
Make a union of all the constraints of Eq. (7) for all data references in the cluster.
Solve it by using the Gaussian elimination method.
If a non-trivial solution exists
{ finding a communication-free decomposition }
else
{ Compute the dependence weight of each loop in the cluster.
 While non-trivial solution still not found
 { relax the constraint according to the dependence weight in decreasing order, and modify the system of equations.
  Resolve the system of equations by using the Gaussian elimination method.
  If a non-trivial solution exists
  { finding a pipeline decomposition }
 }
}
End.

2.3 The cost estimation function

The cost estimation function is used to determine the priority in the proposed algorithm. In the following, the symbols used in the cost estimation function are introduced at first.

T: The total data access time.

T_l: The local data access time of one data element.

T_r: The remote data access time of one data element.

In the following, we first consider the 2-dimensional array case and then the N-dimensional array.

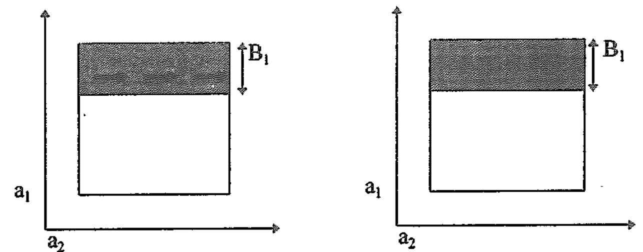
(1) The 2-dimensional array case

Suppose there is a NxN 2-dimensional array A. B_1 and B_2 are the block size of the first and second dimension of array A respectively. We can classify the decomposition into the following four cases.

Case 1: The data accessed by the computation decomposition is the same as the data decomposition, and the data dependence is within the partition, as shown in Fig. 3.

This is the case of communication-free decomposition. Each processor can access the data needed in its local memory. So the total data access time can be represented in the following:

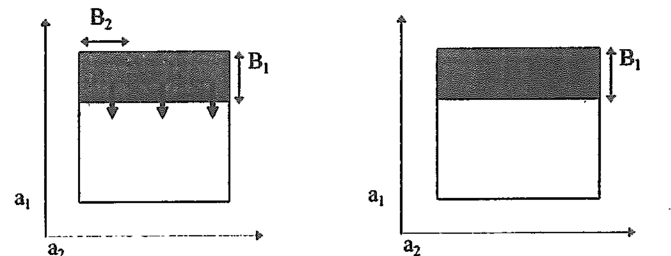
$$T = N * T_l * B_l \tag{A-1}$$



(A).The data decomposition accessed by the computation decomposition (B). The data decomposition

Figure 3.

Case 2: The data accessed by the computation decomposition is the same as the data decomposition, and the data dependence crosses partitions, as shown in Fig. 4.



(A).The data decomposition accessed by the computation decomposition (B). The data decomposition

Figure 4.

The processor can access most of the data needed in its local memory. But the data dependencies are cross boundary between two of neighboring partitions, and each processor needs to access the data in the remote memory. The system executes the program and transfers the data in pipeline, so we can represent the total data access time in the following form:

$$T = (B_1 * N * T_l + N * T_r) + (\frac{N}{B_1} - 1) * (B_1 * B_2 * T_l + B_2 * Tr) \tag{A-2}$$

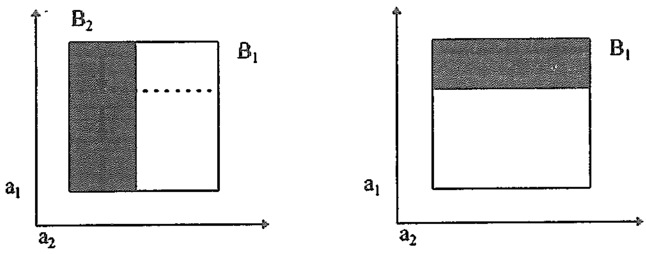
Here, we divide the data communication into N/B_1 pipelines, the execution time of each pipeline is $(B_l * N * T_l + N * T_r)$. The block size of each pipeline

stage is $B_1 * B_2$, each takes communication time $(B_1 * B_2 * T_l + B_2 * Tr)$.

Case 3: The data accessed by the computation decomposition is not the same as the data decomposition; the data dependence is within a partition, as shown in Fig. 5.

Each processor can access only $B_1 * B_2$ data elements in its local memory, most of the $(B_2 * N - B_1 * B_2)$ elements need to access in the remote memory. So the total data access time can be represented in the following:

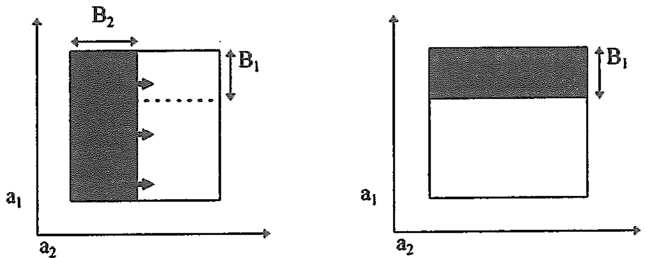
$$T = B_1 * B_2 * T_l + (B_2 * N - B_1 * B_2) * Tr \qquad \text{(A-3)}$$



(A).The data decomposition accessed by the computation decomposition (B). The data decomposition

Figure 5.

Case 4: The data accessed by the computation decomposition is not the same as the data decomposition; the data dependence is cross the partition, as shown in Fig. 6.



(A).The data decomposition accessed by the computation decomposition (B). The data decomposition

Figure 6.

Each processor can access only $B_1 * B_2$ data elements in its local memory, most of the $(B_2 * N - B_1 * B_2)$ elements need to access in the remote memory. The data dependencies are cross the partition, so the total data access time can be represented in the following form.

$$T = B_1^2 * T_l + (B_1 * N - B_1^2) * Tr + (\frac{N}{B_1} - 1) * (B_1 * B_2 * T_r). \qquad \text{(A-4)}$$

(2) The N-dimensional Array

The calculation of the data access time of the N-dimensional array is similar to the 2-dimensional one. We can derive the following formula by extending from (A-1) to (A-4) directly and easily.

Case 1: The data accessed by the computation decomposition is the same as the data decomposition, and the data dependence is within the partition.

$$T = (N^{n-1} * T_l + N^{n-1} * T_l) * B_1 \qquad \text{(B-1)}$$

Case 2: The data accessed by the computation decomposition is the same as the data decomposition, and the data dependence is cross the partition.

$$T = (B_1 * N^{n-1} * T_l + N^{n-1} * T_r) + (\frac{N}{B_1} - 1) * (B_1 * (\prod_{i=2}^{n} B_i) * T_l + (\prod_{i=2}^{n} B_i) * Tr) \qquad \text{(B-2)}$$

Case 3: The data accessed by the computation decomposition is not the same as the data decomposition, and the data dependence is within the partition.

$$T = B_1 * B_2 * N^{n-2} * T_l + (B_1 * N^{n-1} - B_1 * B_2 * N^{n-2}) * Tr \qquad \text{(B-3)}$$

Case 4: The data accessed by the computation decomposition is not the same as the data decomposition, and the data dependence is cross the partition.

$$T = B_1 * B_2 * T_l + (B_1 * N - B_1 * B_2) * Tr + (\frac{N}{B_1} - 1) * (B_1 * (\prod_{i=2}^{n} B_i) * T_r)) \qquad \text{(B-4)}$$

2.4 The whole algorithm

The proposed algorithm is aimed to eliminate the largest amount of communication, and then merge the nodes having the greatest edge values into the same cluster. If all nodes in CG are merged into one cluster finally, there is a static decomposition for the program. Otherwise, the CG may have more than one cluster, each data and loop nests within the cluster are labeled with their decomposition, and data communication will occur between the clusters. That is, there is a dynamic decomposition for the program. The steps of this algorithm are as follows:

The Decomposition Algorithm

Input: A program and its communication graph
Output: A collection of clusters, each cluster is labeled with computation and data decomposition.
Begin
Initializing computation and data decomposition for each node in CG by the modified locality algorithm.
Construct the priority list by evaluating the cost estimation function for each node in CG.
While there is any un-examined edge
{ Merge the nodes along the unexamined edge with maximal priority in the CG.
Finding the current optimal computation/data decomposition by modified locality algorithm.
Evaluating the communication graph's critical path

length by cost estimation function and updating the priority list.
If the critical path length is shorter than that of previous step
{ Merge the nodes along this edge. }
}
End

This algorithm is a greedy approach, it never back tracks the decision of each step. Therefore, the time complexity is polynomial. Assume that m is the number of data references in a cluster, l is the number of iterations and a is the number of arrays. Since there are at most m constraints to be relaxed, at most O(l+a) variables and O(m*l) equations. Therefore, the time complexity of the modified locality algorithm is $O(m^*(m^*l)^2(1+a)) = O(m^3l^2(1+a))$. This complexity is reasonable, because in [6] corresponding complexity is $O(m^3k^3)$, where m is the number of total data references in the cluster, and k is the number of loop indices and arrays in the cluster. Consequently, the time complexity of proposed algorithm is $O(em^3l^2(1+a))$, where e is the number of edges in the CG.

3. Performance evaluations

We have implemented the proposed algorithms in SUIF [8]. The performance is evaluated by our simulation environment [9], which has been implemented in the front-end part of the MINT simulator. The progress of our performance evaluation is shown in Fig. 7.

The benchmarks used here are the following:

1. Five-Point Stencil: Calculate the average value of five points, it usually used in the image processing.
2. ADI Integration: Solving the partial differential equations.
3. Vpenta: One of the kernel in nasa7, a program in the SPEC92 floating-point benchmark suite. It simultaneously inverts three pentadiagonal matrices.
4. MxM: One of the kernel in nasa7, the multiplication of two matrices.

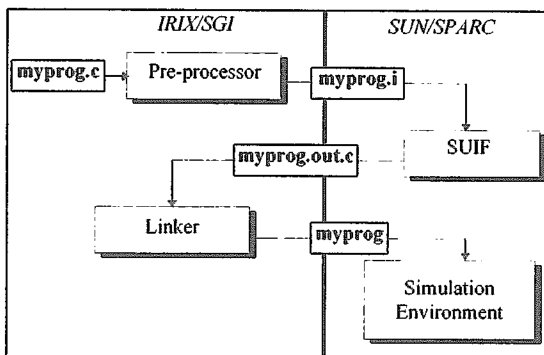


Figure 7 Simulation environment.

Each benchmark is compiled to the parallel code of different number of processors: 1, 2, 4, 8 and 16. We gather the execution time and network traffic from executing the above benchmarks based on our method compared with that of [6]. Basically, we use the following evaluation criteria.

Improvement of Execution time =

$$\frac{\text{Ning's Execution time - Our method's Execution time}}{\text{Ning's Execution time}}$$

Improvement of Network Traffic =

$$\frac{\text{Ning's Network Traffic - Our method's Network Traffic}}{\text{Ning's Network Traffic}}$$

Because the data arrays are distributed onto different processors, the network traffic is increasing when the number of processors is increasing. Fig. 8 is the improvement of Five-Point Stencil, the network traffic decreases about 29%, and the execution time improve about 10% in average. Fig. 9 is the improvement of ADI, the network traffic decreases about 17%, and the execution time improve about 7.9% in average. Fig. 10 is the improvement of Vpenta, the network traffic decreases about 10.9%, and the execution time improve about 7.9% in average. Fig. 11 is the improvement of MxM, the network traffic decreases about 6.5%, and the execution time improve about 7.1% in average. Our method can efficiently decreasing the network traffic than that of [6]. According to the experimental results, we can say that, selectively relaxing the data locality constraints to decrease the network traffic can improve the system performance.

4 Concluding remarks

In this paper, we have presented the method of automatically decompose computation/data for loop nests in programs on NUMA systems. The proposed algorithm could balance the criteria between parallelism exploitation and locality extraction. Using the function of cost estimation, we could relax the data locality constraints with less communication volume, reduce the network traffic, and then improve the system performance. Experimental results show that the proposed algorithm could improve the execution time about 8.5% in average compared to that of [6].

References

[1] B.Bixby, K.Kennedy, and U.Kremer, "Automatic data layout using 0-1 interger programming", Proceeding of the International Conference on Parallel Architectures and Compilation Techniques(PACT), pp. 111-122, Montreal, Cannada, August 1994.

[2] C.H.Huang, and P.Sadayappan, "Communication-Free Hyperplane Partitioning of Nested Loops", Languages and Compilers for Parallel Computing, pp. 186-200,

Springer-Verlag, 1992.

[3] J.M.Anderson and M.S.Lam, "Global optimizations for parallelism and locality on scalable parallel machine", in Proc. of the SIGPLAN'93 Conference on Program Language Design and Implementation, pp. 112-125, Albuquerque, NM, June 1993.

[4] U.Kremer, "NP-Completeness of dynamic mapping". In Proceddings of the Forth Workshop on Compilers for Parallel Computers, Delft, The Netherland, 1991.

[5] J. Li and M.Chen, "Index domain alignment: minimizing cost of cross-reference between distributed arrays". In Proceedings of the 3rd Symposium on the Frontiers of Massively Parallel Computation, College Park, Maryland, October 1990.

[6] Q.Ning, V.V.Dongen and G.R.Gao, "Automatic data and computation decomposition for distributed memory machines". In Proceedings of the 28th Annual Hawaii Int. Conference on System Sciences, pp. 103-112, 1995.

[7] J.Ramanujam, and P.Sadayappan, "Compile-Time Techniques for Data Distribution in Distributed Memory Machines", IEEE Trans. Parallel and Distributed Systems, vol.2, pp. 472-282, Oct. 1991.

[8] Stanford SUIF Compiler Group, SUIF: A parallelizing & optimizing research compiler, Tech. Rep. CSL-TR-94-620, Stanford University, May 1994.

[9] Su, Jen-Pin, "A Study on Memory Subsystem Design for Multiprocessor System and Implementation of Its Simulation and Evaluation Environment," Thesis, CSIE, NCTU, 1996.

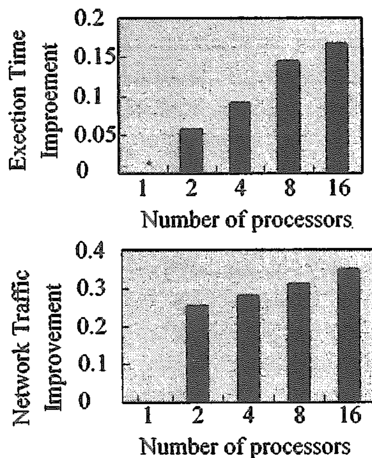


Figure 8 Improvements of Five-Point Stencil

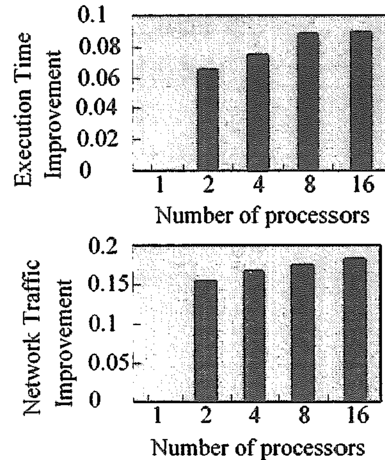


Figure 9 Improvements of ADI

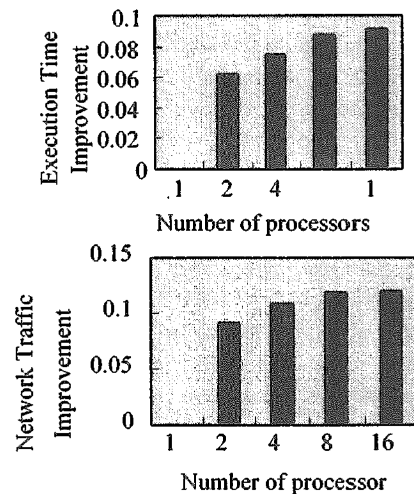


Figure 10 Improvement of Vpenta

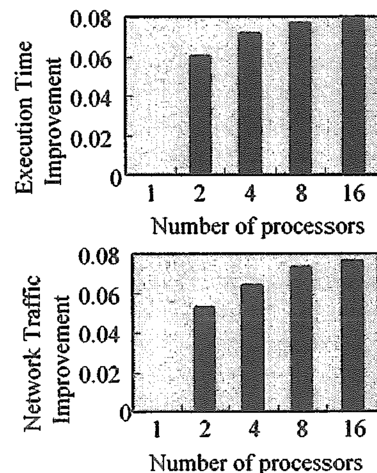


Figure 11 Improvement of MxM