Proceedings of International Conference
on Computer Architecture

Data Alignment for Parallelizing Compiler on Distributed-Memory Multiprocessors

Tai-Hsiang Tsai and Tsung-Chuan Huang
Department of Electrical Engineering
National Sun Yat-Sen University
Kaohsiung, TAIWAN, R.O.C.
tch@mail.nsysu.edu.tw

Abstract

In distributed-memory multiprocessors system, the distribution of data across processors is critically important to the efficiency in executing a data parallel program. This is due to how this distribution is done will affect the amount of data movement required and the interprocessor communication is much more expensive than computation.

In this paper, we analyze the reference patterns in a program where multiple arrays used have affinity relations. The abstract machine we adopt is a D-dimensional grid of $N_1 \times N_2 \times \ldots \times N_D$ processors, where D is the smallest dimensionality of arrays used in the program. Based on the reference patterns, we present a scheme to find a maximum spanning tree in the alignment group. For each maximum spanning tree, we then determine the alignment function for all dimensions of arrays. For each reference pattern in the maximum spanning tree, we can align each pair of the active elements for these two dimensions of arrays to the same place of template, such that the reference pattern with maximum cost will get free communication in all distribution fashions and in any number of processors.

Keywords : distributed-memory multiprocessors, reference pattern, affinity relation, alignment function, data distribution.

1. Introduction

Distributed-memory multiprocessors system is used for providing high levels of performance for scientific applications. The distributed memory machines offer significant advantages in terms of cost and scalability, but they are difficult to program. The code and data must be distributed to processors by the programmer himself. In SPMD(single program multiple data) program mode, the node programs have the same code, so the distribution of data across processors is of critical importance to the efficiency of parallel programs. This is because interprocessors communication is much more expensive than computation in processors. As the result, for a program executing in a distributed-memory multiprocessors system, it is essential that the data used by a processor had better be located at its local memory as much as possible.

For a data distribution strategy, there are two important considerations. The first is how to reduce the communication between processors, and the second is how to balance the load of processors. In the past years, data distribution has received much attention in the literature. The problem of finding optimal dimension distribution has been shown to be NP-complete[9]. Li and Chen[9], Gupta and Banerjee[3] formulated the component alignment problem from the whole source program and used it to determine data distribution. Kalns, Xu, and Ni [5] provided a cost model for determining a small set of appropriate data distribution patterns among many possible choices. Chen and Sheu[2] concentrated on automatically allocating the array elements of nested loops with uniformly generated references for communication-free execution on distributed memory multiprocessors. Ramanulam and Sadayappan[10] determined data distribution based on the hyperplane method. Lee[6,7,8] derived efficient algorithms for determining data distribution and generating communication sets.

In High Performance Fortran(HPF), data distribution is divided into alignment phase and distribution phase. Programmers have obligations to provide TEMPLATE, ALIGN, and DISTRIBUTE directives to specify data distribution. Then, base on these directives, the compiler generates all communication instructions. In this paper, we focus on alignment phase, all dimensions of arrays will be aligned to template using alignment functions, and distributed according to the distribution of template.

The rest of this paper is organized as follows. In section 2, we introduce the component alignment proposed by Li and Chen[9]. In section 3, we introduce the method of determining alignment functions for all dimensions in an alignment group.

In section 4, we show some results of communication evaluation. Finally, conclusions are given in section 5.

2. Background

In this paper, the abstract target machine we adopt is a D-dimensional grid of $N_1 \times N_2 \times \ldots \times N_D$ processors, where D is the smallest dimensionality of arrays used in the program. A processor on the grid is represented by the tuple $(p_1,p_2,\ldots,p_D)$, for $0 \le p_i \le N_i-1$ and $1 \le i \le D$.

Consider an array assignment statement appearing in an *m*-level multiple-loop. Let array references $A[i_1,\ldots, i_m]$ and $B[j_1,\ldots, j_n]$ are respectively on the left-hand side and right-hand side of the assignment statement. For each array reference $B[j_1,\ldots,j_n]$, the symbolic form $A[i_1,\ldots, i_m] \leftarrow B[j_1,\ldots,j_n]$ is called a *reference pattern*, where the indices $[i_1,\ldots, i_m]$ and $[j_1,\ldots, j_n]$ are quantified over their index domain[9]. According to this definition, multiple reference patterns may be derived from a single array assignment statement if there is more than one instance of array references occurring on the right-hand side of the statement. A reference pattern is either a *self-reference* pattern to the same array or a *cross-reference* pattern between different arrays. Given a cross-reference pattern $A[i_1,\ldots, i_p,\ldots, i_m] \leftarrow B[j_1,\ldots,j_q,\ldots,j_n]$, A_p and B_q are said to have an *affinity relation* if $j_q = i_p + c$, where c is a small constant. *Component affinity graph*(CAG)[9] is a weighted graph, whose nodes represent the component of index domains to be aligned and the edges specify the affinity relations between two nodes for each distinct reference pattern, excluding self-reference patterns, in the program. In other words, there is no edge between two nodes corresponding to the same array. The weight associated with an edge is equal to the communication cost if these two dimensions of arrays are distributed along different dimensions of processor grid. Li and Chen[9] constructed the CAG from the source program, and proposed a heuristic algorithm to solve component alignment problem. When constructing component affinity graph from a source program, the component alignment problem is defined as partitioning the node set into D disjointed subsets, where D is the dimension of the abstract target machine.

In this paper, we assume the dimension of processor grid is equal to the smallest dimensionality of all arrays. Before partitioning the node set, we must delete the redundant nodes such that each array has the same number of nodes equal to the dimension of processor grid. This is because we can't distribute an array whose dimension is smaller than that of processor grid, unless we replicate it on some dimensions of processor grid. This method will be introduced at the next section.

In partitioning the node set into D disjointed subsets , our aim is to minimize the total weight of edges across different subsets. There is one restriction at here: no two nodes corresponding to the same array are in the same subset. All dimensions of arrays in a subset will be distributed to the same dimension of processor grid, different subsets will be distributed to different dimensions of processor grid.

Table 1 lists the communication primitives used in the hypercube machine[4,8], for which the subscripts of dimension for arrays in the left-hand-side(lhs) and right-hand-side(rhs) of assignment have some specific patterns, and their costs on the hypercube. The parameter m denotes the message size; seq is a sequence of identifiers representing the processors in various dimensions over which the collective communication primitives are carried out. The function num returns the total number of processors.

Table 1. Communication primitives used in the hypercube and their costs.

Lhs	Rhs	Communication Primitive	Cost on Hypercube
c_1	c_2	Transfer(m)	O(m)
i	i±c	Shift(m)	O(m)
i	c	OneToManyMulticast(m,seq)	O(m* log num(seq))
c	I	Reduction(m,seq)	O(m* log num(seq))
i or $f_1(i)$	j or $f_2(j)$	ManyToManyMulticast(m,seq)	O(m* num(seq))

In the original algorithm of component alignment problem[9], the node set of the component affinity graph will be partitioned into D disjointed subsets, where D is the largest dimensionality of all arrays. We use the following example to illustrate this partition.

Example 1:

```
for I=1 to m
    for J=1 to m
        A[I,J]=B[J+1]+C[J]
        D[I+1,J]=A[I,J]
        B[I]=A[J+1,I]
    endfor
    A[I,1]=B[I]
    B[I]=C[I+1]
```

endfor

The component alignment graph of this program is as follows:

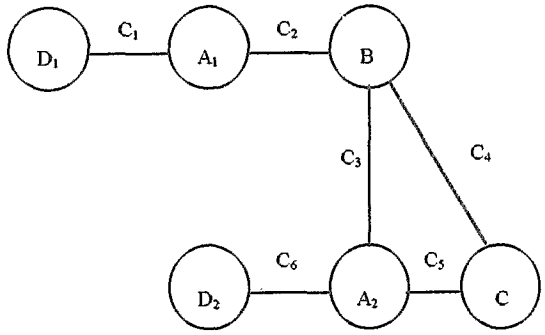

Fig. 1. CAG of example 1.

Suppose that A and D are two-dimensional arrays of m×m, B and C are one-dimensional arrays of size m, and the number of processors is equal to N. Then

$c_1=c_6$=ManyToManyMulticast(m^2/N,N)

$c_2=c_4=c_5$=ManyToManyMulticast(m/N,N)

Because there are two reference patterns having affinity relation between B and A_2, we have

c_3=ManyToManyMulticast(m^2/N,N)

+ManyToManyMulticast (m/N,N)

Since $c_3 + c_4 > c_4=c_2$, after applying the component alignment algorithm, we get two disjointed subsets as Fig. 2 shows. (A disjointed subset will be referred to the alignment group in the rest of this paper.) ■

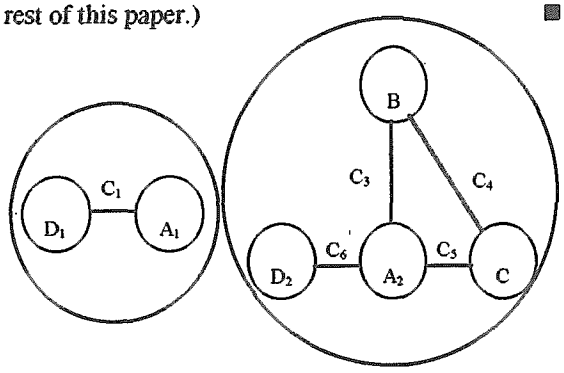

Fig. 2. CAG after applying the component alignment algorithm.

3. Data Alignment

In previous section, we had pointed out why we made the dimension of processor grid to be equal to the smallest dimensionality of all arrays. In the following, we will use an example to illustrate how to delete the redundant nodes.

Example 2:

Assume the reference patterns are

A[I,J]←B[J+1,I]

B[I,J]←C[I]

A[I+1,J]←C[I]

D[I]←C[I+1]

B[I,J]←D[I+1]

E[I]←F[I+1]

We can construct the following component alignment graph:

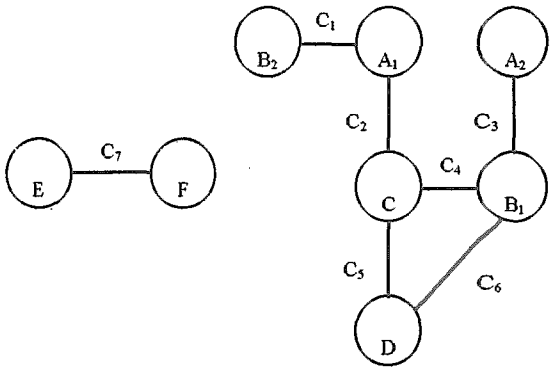

Fig. 3. CAG of example 2.

Since the dimension of processor grid is equal to one, we must delete one redundant node for each of two-dimensional arrays A and B. These two redundant nodes will be one of these four combinations: (A_1,B_1), (A_1,B_2), (A_2,B_1), and (A_2,B_2). When a node is removed, all edges associated with this node must also be removed. Thus, the costs for these four combinations can be computed as follows:

(A_1,B_1) : $c_1+c_2+c_3+c_4+c_6$

(A_1,B_2) : c_1+c_2

(A_2,B_1) : $c_3+c_4+c_6$

(A_2,B_2) : c_1+c_3

Since $c_1 + c_2 < c_1 + c_3 < c_3 + c_4 + c_6 < c_1 + c_2 + c_3 + c_4 + c_6$, the pair of ($A_1$,$B_2$) is deleted. After this, we get two alignment groups as Fig. 4 shows. ■

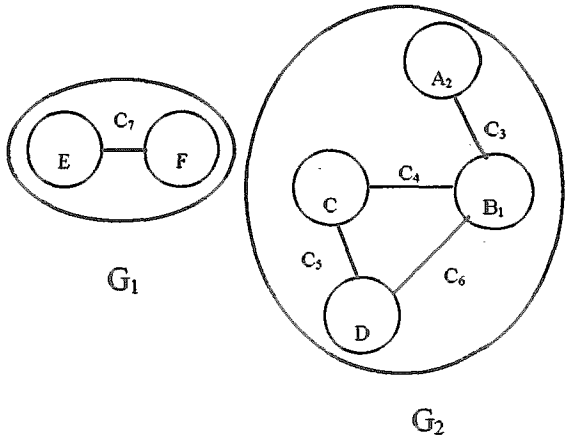

Fig. 4. CAG after deleting A1 and B2.

Since the dimension of template is equal to the dimension of processor grid, which is one in example 2, and the two alignment groups G_1 and G_2 contain no common array, we can align the nodes in G_1 to a template T, and align the nodes in G_2 to another template T'. In the following, we will introduce how to determine the alignment function for all dimensions in an alignment group.

3.1. Alignment function

The alignment function we consider will be limited to the form of "$\alpha X+\beta$" in this paper. Since there is at least one affinity relation between two dimensions if an edge exists, there may have more than one affinity relation associated with an edge. For different affinity relations, they may have different costs. In this situation we will remove all the affinity relations aside from the one whose cost is the largest among all of the affinity relations.

In an alignment group, there may exist cycles. Before determining the alignment function for all dimensions in an alignment group, we must break the cycles. Otherwise, the alignment functions may conflict with each other. For convenience, the affinity relation between the i-th dimension of array A and j-th dimension of array B is represented by the reference pattern $A_i[aI+b] \leftarrow B_j[cI+d]$. In what follows, we will explain how to construct a maximum spanning tree from an alignment group containing cycles, and then find the alignment functions for each array.

Given the reference patterns in a alignment group and their costs as below:

Reference pattern	Cost
$A_1[I] \leftarrow B_1[I+1]$	ManyToManyMulticast(m^2/N,N)
$A_1[I] \leftarrow B_1[2I+1]$	ManyToManyMulticast(m^2/N,N)
$A_1[I] \leftarrow B_1[I+1]$	ManyToManyMulticast(m^2/N,N)
$A_1[I] \leftarrow C[I+2]$	ManyToManyMulticast(m/N,N)
$C[2I] \leftarrow B_1[3I+2]$	ManyToManyMulticast(m^2/N,N)
$B_1[I] \leftarrow D[I-1]$	ManyToManyMulticast(m^2/N,N)

The alignment group is shown in Fig. 5. For the convenience of reference, we associate each edge with a reference pattern. As mentioned above, the reference pattern associated with an edge is that with maximum cost. Since there are two affinity relations between A_1 and B_1, and the reference pattern $A_1[I] \leftarrow B_1[I+1]$ appears two times, we associate the edge between A_1 and B_1 with this reference pattern.

The costs of edges in the alignment group can be computed as below:

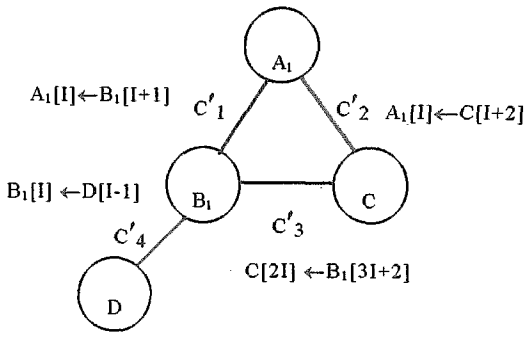

Fig. 5. An alignment group containing a cycle.

c'_1 = 2*ManyToManyMulticast(m^2/N,N)

c'_2 = ManyToManyMulticast(m/N,N)

c'_3 = ManyToManyMulticast(m^2/N,N)

c'_4 = ManyToManyMulticast(m^2/N,N)

We can see that there is a cycle in this alignment group. Because $c'_1 > c'_3 > c'_2$, after eliminating the edge between A_1 and C, the maximum spanning tree is obtained and shown in Fig. 6.

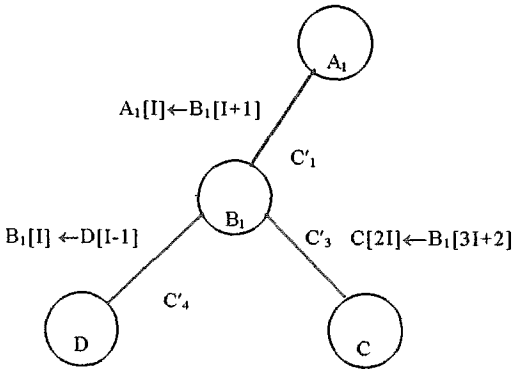

Fig. 6. The maximum spanning tree in Fig. 5.

In the next step, we will determine the alignment functions from the reference patterns appearing in the maximum spanning tree, and use these alignment functions to align each pair of the active elements of these reference patterns to the same place of template. In the maximum spanning tree, we can choose any node to be the root. For the above example, we let A_1 be the root. We will determine the alignment function from A_1, and then the descendent nodes one by one.

[Definition] Relative value

For a reference pattern $A_i[aI+b] \leftarrow B_j[cI+d]$, let $c'/a' = c/a$ such that $\gcd(c',a')=1$. We say that the relative value of B_j to A_i is c', and A_i to B_j is a'. ■

Fig. 7 shows the relative values of each edge in Fig. 6, where the relative value is for the descendent node relative to its ancestor.

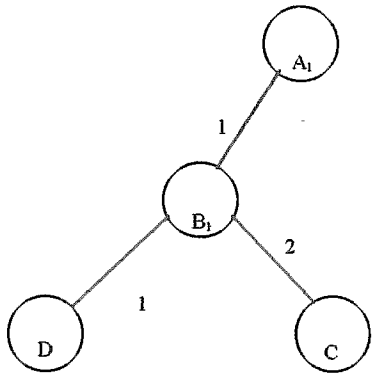

Fig. 7. The relative values in Fig. 6.

We assume that the root's alignment function is of this form "$\alpha_{root}*X+\beta_{root}$", where β_{root} is a non-negative integer, α_{root} is equal to the absolute value of LCM(least common multiple) of all relative values in the maximum spanning tree for an alignment group. In this sense, for above example, the alignment function for A_1 will be "$f_{A1}(X)= \alpha_{A1}X+\beta_{A1}$", for B_1 will be "$f_{B1}(X)= \alpha_{B1}X+\beta_{B1}$", for C will be "$f_C(X)= \alpha_C X+\beta_C$", and for D will be "$f_D(X)= \alpha_D X+\beta_D$".

Since A_1 is the root, we compute the alignment function from A_1. The α_{A1} would be equal to $|lcm(1,1,2)|=2$. And according to the reference pattern $A_1[I] \leftarrow B_1[I+1]$, we have $f_{A1}(I)= f_{B1}(I+1)$. Therefore,

$$\alpha_{A1}I+\beta_{A1}= \alpha_{B1}(I+1)+\beta_{B1}$$

Since $\alpha_{A1}=2$, this becomes

$$2I+\beta_{A1}=\alpha_{B1}I+\alpha_{B1}+\beta_{B1}$$

Comparing the coefficients of I, we have α_{B1} to be equal to 2. And, we get

$$\beta_{A1}=2+\beta_{B1}$$

Assume that β_{B1} is zero, then we have $\beta_{A1}=2$. Therefore, the alignment function for A_1 is 2X+2, and for B_1 is 2X.

Now, we can compute the alignment function for C as follows:

$$f_{B1}(3I+2)= f_C(2I)$$

This becomes

$$2(3I+2)+0=\alpha_C(2I)+\beta_C$$

It yields

$$6I+4=2\alpha_C I+\beta_C$$

Comparing the coefficients, we have $\alpha_C=3$ and $\beta_C=4$. This means the alignment function for array C is 3X+4.

By the similar discussion, we can find the alignment function for array D to be $f_D(I)$ is 2X+2.

Using instruction formulation, we can describe these alignments as follows:

Align A[I,*] with T[2I+2]
Align B[I,*] with T[2I]
Align C[I] with T[3I+4]
Align D[I] with T[2I+2]

As for the data distribution, if the alignment function for an array is $\alpha X+\beta$ and the template will be distributed to N processors with block size k, then its distribution function would be $f(X)= \lfloor(\alpha X+\beta)/k\rfloor \bmod N$.

4. Communication evaluation

To evaluate the performance of our scheme, we analyze the amount of communication of the following three array statements in different distribution fashions and different number of processors. These three array statements are as follows:

Array statement 1: A[1:601:2]=f(B[0:900:3]) (R_1)

Array statement 2: A[1:601:2]=f(B[4:904:3]) (R_2)

Array statement 3: A[2:902:3]=f(B[1:601:2]) (R_3)

The alignment function for array A can be computed to be "3I" and for array B to be "2I+3". Table 2 shows the amount of communication between processors for each array statement and four different block size. From Fig. 8 we can see that no communication arises in R_1 for any block sizes, because the alignment function is derived from the reference pattern in this statement. The amount of communication decreases when block size increases for array statement R_2. As for R_3, the amount of communication is not affected very much by the block size.

Next, let's observe the relation between amount of communication and number of processors. In this situation, the block size is fixed to 16. From Fig. 9 we can see that array statement R_1 still has no communication in any number of processors. For array statement R_2 , the amount of communication remains unchanged for different number of processors because the block size is fixed. In array statement R_3, the amount of communication increases relatively with the number of processors.

Finally, we use BLOCK distribution to distribute the template. The result is shown in Fig. 10. We find that R_1 still has no communication whatever the number of processors is. For R_2, the amount of communication is slightly increased when the number of processors rises. The relation between amount of communication and number of processors is irregular for array statement R_3.

From these results, we can see that we can find an alignment function for R_1 such that R_1 is always free communication regardless of the distribution of template and the number of processors.

Table 2. The amount of communication between processors.

		p_0				p_1				p_2				p_3			
		4	8	16	32	4	8	16	32	4	8	16	32	4	8	16	32
	R_1	75	76	77	77	0	0	0	0	0	0	0	0	0	0	0	0
p_0	R_2	0	0	38	58	0	0	0	0	76	0	0	0	0	75	38	19
	R_3	19	20	19	20	19	19	20	23	19	18	19	20	18	18	17	16
	R_1	0	0	0	0	75	76	74	75	0	0	0	0	0	0	0	0
p_1	R_2	0	76	39	19	0	0	37	56	0	0	0	0	75	0	0	0
	R_3	19	19	20	20	19	19	18	16	19	19	18	19	19	19	20	23
	R_1	0	0	0	0	0	0	0	0	76	74	75	74	0	0	0	0
p_2	R_2	75	0	0	0	0	76	37	19	0	0	38	56	0	0	0	0
	R_3	19	19	19	19	19	20	19	19	18	19	20	18	17	18	18	16
	R_1	0	0	0	0	0	0	0	0	0	0	0	0	75	75	75	75
p_3	R_2	0	0	0	0	75	0	0	0	0	74	37	18	0	0	37	56
	R_3	19	18	20	19	18	18	18	16	19	19	17	18	19	19	19	19

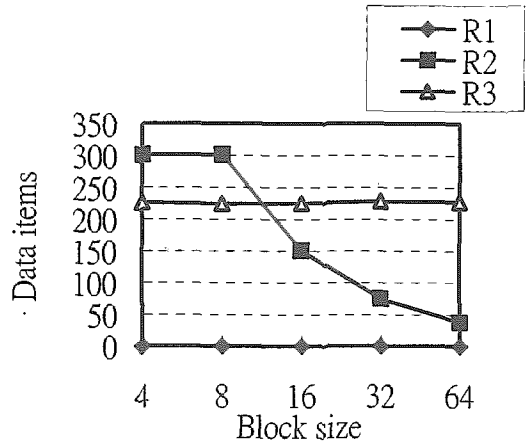

Fig. 8. The relation between amount of communication and different block size.

5. Conclusions

In this paper, we analyze the reference patterns that have affinity relation, and determine the alignment functions automatically for all dimensions of arrays. For each reference pattern between two dimensions of two arrays in the maximum spanning tree, we can obtain the alignment functions for them such that there is no communication for these reference patterns.

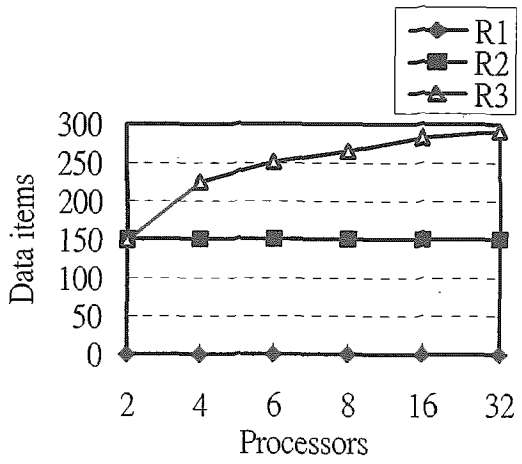

Fig. 9. The relation between amount of communication and number of processors (block size=16).

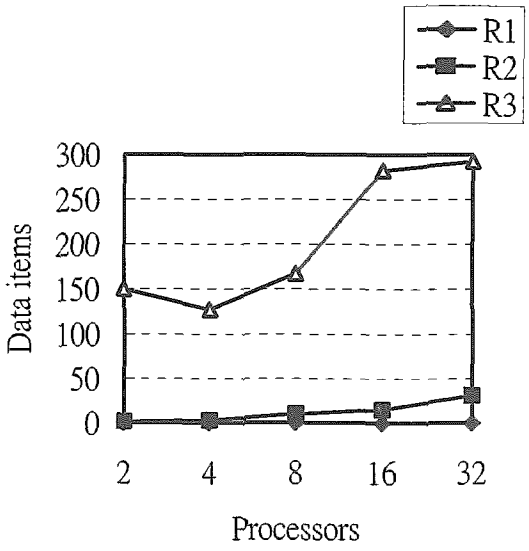

Fig. 10. The relation between amount of communication and number of processors using block distribution.

This method will significantly reduce the communication overhead if the reference patterns between two dimensions of arrays are few or there is a reference pattern whose cost is much greater than the others. From the result of simulation, we prove that we can find the alignment functions for given reference patterns. The reference pattern with maximum cost is free communication.

References

[1] M. Chen, Y. I. Choo,and J. Li, "Compiling parallel programs by optimizing performance,"

The Journal of Supercomputing, vol.2, pp.171-207, 1988.

[2] T. S. Chen and J. P. Sheu, "Communication-free data allocation techniques for parallelizing compilers on multicomputers," IEEE Transaction Parallel and Distributed Systems, pp.924-938, vol.5, no.9, Sep. 1994.

[3] M. Gupta and P. Banerjee. "Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers," IEEE Transaction on Parallel and Distributed Systems, vol.3, no.2, pp.179-193, Mar.1992.

[4] Gupta and P. Banerjee. "Compile-time estimation of communication costs in multicomputers," Technical Report CRHC-91-16, University of Illinois, May. 1991.

[5] E. T. Kalns, H. Xu, and L. M. Ni, "Evaluation of data distribution patterns in distributed-memory machines," In Proc. of International Conference on Parallel Processing, pp.175-183, Aug. 1993.

[6] P. Z. Lee, "Techniques for compiling program on distributed memory multicomputers," Institute of Information Science, Academia Sinica.

[7] P. Z. Lee, "Efficient algorithms for data distribution on distribution memory multicomputers," Institute of Information Science, Academia Sinica.

[8] P. Z. Lee and W. Y. Chen, "Compiler techniques for determining data distribution and generating communication sets on distributed-memory multicomputers, " the 29th IEEE Hawaii International Conference on System Science, Maui, Hawaii, January, 1996.

[9] J. Li and M. Chen, "The data alignment phase in compiling programs for distributed-memory machines," Journal Parallel and Distributed Computing. vol 13. pp.213-221, 1991.

[10] J. Ramanujam and P. Sadayappan , "Compile-time techniques for data distribution in distributed memory machines," IEEE Transaction Parallel and Distributed Systems, vol.2, no.4, pp.472-482, Oct. 1992.