Toward Automatic Distribution

Paul Feautrier
Laboratoire PRiSM
Université de Versailles
feautrier@prism.uvsq.fr

April 8, 2009

Abstract

This paper considers the problem of distributing data and code among the processors of a distributed memory supercomputer. Provided that the source program is amenable to detailed dataflow analysis, one may determine a *placement function* by an algorithm analogous to Gaussian elimination. Such a function completely characterizes the distribution by giving the identity of the virtual processor on which each elementary calculation is executed. One has then to "realize" the virtual processors on the PE. The resulting structure satisfies the "owner computes" rule and is reminiscent of two-level distribution schemes, like HPF's `ALIGN` and `DISTRIBUTE` directives, or the CM-2 virtual processor system.

1. Introduction

The emphasis in supercomputer architecture has recently shifted from vector processors to massively parallel computers. The main cause of this change is the availability of new RISC chips, which offer in a small package a processing power which is a significant fraction of the power of most vector processors. Building a supercomputer by assembling a moderate to large number of such chips – from 32 to more than a thousand – seems to make good sense, both in term of price and of computing power. For technical reasons, it is very difficult to equip such a computer with a global memory with uniform access time. The simplest possibility, from the hardware designer point of view, is to distribute the memory among processors. A network takes care of inevitable communications. There is generally more than an order of magnitude between the local memory access time and the network latency, hence the importance of good data placement for minimizing the communication overhead. This problem is usually left to the programmer. Computers built according to this scheme are *message passing* architectures.

The alternative is to hide the problem by providing a uniform address space with a non uniform access time. All such proposals – among which the most notable are distributed caches and Distributed Virtual Memory – rely heavily on data locality to obtain acceptable performances. We see that, on the whole, correct placement of the data, whether in a real or virtual global memory or among the distributed memories of a message passing architecture, is the critical factor in the overall performance of this type of supercomputer. When the placement is known, in a message passing

architecture, the compiler still has to build the communication code. This chore is taken care of dynamically by a combination of hardware and software in the case of global memory or DVM.

Several authors have proposed that the distribution be specified by the programmer [1,2,3,4]. The rationale is that most scientific computing programs are models of physical reality, and that locality is often obvious in the reality if not in the source code. The problem with this proposal is that, firstly, it does not apply to standard algorithms from numerical analysis, which are defined *in abstracto*, and second that in large codes, intuitive considerations may offer several conflicting distributions. In such a case, one needs a technique for either choosing between proposals, or, alternatively, for using them all for different phases of the computation, with redistribution operations in between.

In this paper, I will explore another direction, automatic distribution. The basic idea is, first, to analyze the source code to identify the communication patterns, and then to construct a *placement* for the program data, with the aim of minimizing – or, at least, of diminishing – the volume of communication.

In the next section, I will present the restrictions on the source programs and on the target architectures which are prerequisites for the application of the method. If these restrictions are met, it is possible to characterize at compile time all communication patterns of the source program. One may then seek a distribution of data among the processors with the aim of replacing most communications by local operations. Such a distribution is specified in term of a *placement function*. Section is devoted to the design of an algorithm for the selection of a good placement function. This algorithm makes use of an ordering of the communication patterns, which is discussed in section . The resulting technique, which is of a highly experimental nature, may be extended in several directions which are discussed in the conclusion.

2. Context

There are many types of distributed computers, with widely differing communication systems. An important characteristics is the so-called topology, i.e. the shape of the connection network. Here, I will consider only the "ether" communication model. All processors are supposed to be interconnected: the time to transmit a message is independent of the source and destination processors. The model is nearly exact for bus networks (for instance, a collection of workstations on an Ethernet). In many recent designs, the architects have taken pains to build fair approximations to the ether model – e.g. by randomizing the routing.

Suppose a processor needs a value. This value may reside in the memory of the processor, in which case the cost of access will be considered as null, or it may reside somewhere else, in which case the cost of communication will be considered as "very high". As a consequence, the quality of the distribution may be assessed simply by estimating the volume of remote data accesses.

Automatic distribution necessitates a global analysis of the source program. With present day techniques, this is only possible if constraints are imposed to the source code. These constraints – which define static control programs [5] – are the following:

- statements are restricted to assigments and `DO` loops,

```
        do i=1,n
1         s = a(i,i)
          do k = 1, i-1
2           s = s - a(i,k)**2
          end do
3         p(i) = 1.0/sqrt(s)
          do j = i+1, n
4           s = a(i,j)
            do k=1,i-1
5             s = s - a(j,k) * a(i,k)
            end do
6           a(j,i) = s * p(i)
          end do
        end do
```

Figure 1: Cholesky Factorization

- the only data structures are scalar and arrays of unrestricted dimension,
- subscripts and loop bounds are affine functions of surrounding loop counters and size parameters.

Programs which do not meet these requirements are very difficult to analyze at compile time. It might be possible to handle them by a combination of approximation techniques and run-time analysis, but this must be left for future research.

In this paper, I will use the program in Fig. 1 as a running example. It is easy to check that this code – a straightforward implementation of the Cholesky Factorization algorithm – has static control.

In a static control program, each statement execution or *operation* may be identified by specifying the executed statement and the values of the surrounding loop counters: the *iteration vector*. Analysis of loop bounds allows one to associate to each statement its iteration domain, which is a subset of N^d, where d is the nesting level for the statement. The iteration domain of R will be noted $\mathcal{D}_R$.

Furthermore, it is possible to analyze the flow of data through the operations and the memory cells of a static control program. For each read access in the program, the set of all preceding write accesses to the same memory cell is characterized and its latest executed element is computed. The result is the source of the value obtained by the read access. A source is composed of a statement name and an iteration vector. Both these elements may depend on the iteration vector of the read access. A method has been defined elsewhere[5] which yields *source functions* in the form of more or less complicated conditional expressions.

The result of this analysis may be presented as the so-called *Dataflow Graph* or DFG for short. The DFG has one node per statement in the original program. There is an edge from statement R (*the source*) to statement S (*the sink*) for each read reference in S which may use a value produced by R. The source and sink of edge e will be written $\sigma(e)$ and $\delta(e)$.

Each edge is labelled by the following information:

- The *governing predicate*, which must be true for the value to be actually used

Edge	Source	Destination	Dimension	Predicate
101	$\langle 2, i, k-1\rangle$	$\langle 2, i, k\rangle$	2	$k-2 \geq 0$
102	$\langle 1, i\rangle$	$\langle 2, i, k\rangle$	1	$1-k \geq 0$
103	$\langle 2, i, i-1\rangle$	$\langle 3, i\rangle$	1	$i-2 \geq 0$
104	$\langle 1, i\rangle$	$\langle 3, i\rangle$	0	$1-i \geq 0$
105	$\langle 3, i\rangle$	$\langle 6, i, j\rangle$	1	
106	$\langle 5, i, j, k-1\rangle$	$\langle 5, i, j, k\rangle$	3	$k-2 \geq 0$
107	$\langle 4, i, j\rangle$	$\langle 5, i, j, k\rangle$	2	$1-k \geq 0$
108	$\langle 5, i, j, i-1\rangle$	$\langle 6, i, j\rangle$	2	$i-2 \geq 0$
109	$\langle 4, i, j\rangle$	$\langle 6, i, j\rangle$	1	$1-i \geq 0$
110	$\langle 6, k, i\rangle$	$\langle 2, i, k\rangle$	2	
111	$\langle 6, k, j\rangle$	$\langle 5, i, j, k\rangle$	2	
112	$\langle 6, k, i\rangle$	$\langle 5, i, j, k\rangle$	2	

Table 1: The Dataflow graph of program `choles`

in the sink. The governing predicate of edge e is associated to a subset $\mathcal{P}_e$ of the iteration domain of the sink $\mathcal{D}_{\delta(e)}$.

- The *sink-to-source transformation* which allows one to compute the iteration vector of the source in terms of the iteration vector of the sink. In most practical cases, this transformation is affine, but there is a possibility of encountering integer divisions as the result of the source computation. The source-to-sink transform of edge e will be denoted by h_e. Its domain is $\mathcal{P}_e$, and its codomain is included in $\mathcal{D}_{\sigma(e)}$.

Table 1 is a representation of the DFG of the Cholesky Factorization program in Fig. 1. Line 108, e.g., indicates that provided that $i \geq 2$, iteration $\langle i, j, i-1\rangle$ of statement 5 provides a value for `s` in iteration $\langle i, j\rangle$ of statement 6.

3. Data and code distribution

3.1. Introduction

There are many types of parallel computers, and each type has its own programming style and introduces its particular kind of overhead. Here, we are interested in asynchronous distributed memory machines. These architectures are programmed according to the message passing paradigm. Data are distributed among the memories of the computer. Each elementary processor acts independently on local data. When it encounters an operation which uses a non-local datum, it has to exchange message with the owner processsor, either to send or receive the needed information.

A common convention when constructing distributed programs is the *owner computes rule*. According to this rule, all computations are done by the processor which owns the result cell. We may consider then that all communications are in the form of remote reads – albeit it is possible to do better at the implementation level. Our main concern is the minimization of the number of remote reads. Now, if the owner computes rule is obeyed, the only cause for communication is the fact that an operation may use a value which has been produced elsewhere. The Dataflow

Graph is a precise description of all such situations. If operations $\langle S, h_e(x)\rangle$ and $\langle R, x\rangle$ are connected in the DFG, a communication will be necessary *unless both operations are executed by the same processor*. This suggests that the distribution pattern be defined statically, i.e. that the same operation is always executed on the same processor on every execution of the program. This may be implemented by postulating the existence of a *placement function* $\Pi(S, x)$ which gives the name of the processor on which $\langle S, x\rangle$ is to be executed. If the computer has P processors, which are numbered from 0 to $P-1$, Π is a function from $\mathcal{D}_S$ to $[0, P-1]$.

To eliminate all communications implied by edge e of the DFG, the placement function has to satisfy the following *placement equation*:

$$x \in \mathcal{P}_e \Rightarrow \Pi(\delta(e), x) = \Pi(\sigma(e), h_e(x)). \tag{1}$$

The first observation is that if we insist that the placement equation be verified everywhere, then the program may well end up being executed on only one processor. As an example of this phenomenon, consider the following kernel:

```
do i = 1,n
  do j = 1,n
     do k = 1,n
       a(i,j,k) = a(i-1,j,k)
              + a(i,j-1,k) + a(i,j,k-1)
     end do
  end do
end do
```

The placement equations are:

$$\Pi(i, j, k) = \Pi(i-1, j, k) = \Pi(i, j-1, k) = \Pi(i, j, k-1)$$

and their only solution is the trivial one $\Pi(i, j, k) = 0$.

A possible way out is to compute the size of the set of operations for which the placement equations are verified, and to select the placement functions which maximize this size, subject to the constraint that the solution is not trivial. In the "ether" model, this makes good sense, since communication overhead depends only on the volume of data to be transferred, not on the position of communicating processors. This is, however, a very difficult problem; an idea of its difficulty can be had by observing that simply computing the size of the iteration space is difficult. One may solve approximately this optimization problem by classifying edges in two categories:

- *cut edges*, for which equation 1 is verified,
- *uncut edges*, for which it is not,

and then maximizing the set of cut edges under the condition that all placement functions are non trivial. This is the solution we are going to explore, with the added refinement that each edge will be assigned a weight, and that we will try to cut edges with large weights first. The problem of selecting the weights will be postponed to the next section.

Placement functions are used to construct a distributed program according to the following recipe: replace each statement S by:

```
if   Π(S, x) = q
then remote-read all non local data for S
     S
end if
```

where x is the iteration vector and q is the processor number. This version is highly inefficient. Optimization is easy only if Π is affine. There is however a difficulty. Consider a program with size parameter n. A typical domain will be of the form

$$\mathcal{D}_S = \{x \mid Mx \geq nb\}.$$

Suppose that $\{x \mid Mx \geq b\}$ is full dimensional. It contains a ball of diameter Δ, and the range of any linear function $\Pi(S, x) = a.x + b$ in $\mathcal{D}_S$ will be at least $n\Delta|a|$. If we insist that a is integral, $|a| \geq 1$, and, for a sufficiently large value of n, Π will have more values than there are processors. The solution is to express Π as the composition of two functions:

$$\Pi = \chi \circ \pi.$$

where π is affine. It maps the computation onto a set of "virtual" processors whose size will depend on the size parameters. χ is a "folding" function with range $[0, P-1]$. The primary objective will be to minimize communication between virtual processors. Since communication between virtual processors which are implemented on the same real processor is simply a copy operation, proper choice of the folding function will offer some opportunities for further reduction of the traffic. This two-tier mapping system is reminiscent of the Connection Machine software [6], or of the templates in HPF [3], the main difference being that templates or so-called *geometries* are multidimensional objects. I will return to that point later.

In this paper, I will be mainly concerned with the determination of the virtual mapping. Some indications on the choice of the folding function will be given in section .

3.2. A Practical Algorithm

The problem is to find a system of functions $\pi(S, x)$ which cuts as many edges as possible. To each edge is associated a distance function:

$$d_e(x) = \pi(\delta(e), x) - \pi(\sigma(e), h_e(x)). \tag{2}$$

The edge is cut if the distance is identically zero in $\mathcal{P}_e$. $\pi(S, x)$ is supposed to be an affine form in x:

$$\pi(S, x) = a_S.x + b_S. \tag{3}$$

For given a_S and b_S, one may compute the $d_e(x)$ and test whether they are null everywhere in $\mathcal{P}_e$. If $\mathcal{P}_e$ contains enough affinely independent points, (i.e., if it is full dimensional), this can happen only if all coefficients in $d_e(x)$ are zero. If not, one may construct a parametric representation of $\mathcal{P}_e$ in term of new independent variables y. One then rewrites $d_e(x)$ in term of the new variables, whose coefficient must also be null. In both cases, the edge cutting condition translates to a system of equations:

$$C_e a = 0,$$

where a is the vector whose components are all the unknown coefficients a_S and b_S. As a rough estimate of the size of the problem, if there are N statements whose mean nesting level is d, there will be $N(d+1)$ unknowns in a, while if there are E edges in the DFG, there will be about $E(d+1)$ equations, some of which may be trivial. Since for practical codes $E \approx 2.R$, where R is the number of read references, the problem will in general be overdetermined.

The matrix C_e is obtained by straighforward algebraic manipulations from h_e. The union of all such systems will be written:

$$Ca = 0. \tag{4}$$

Let us return to the code of Fig. 1. Let us write:

$$\pi(5, i, j, k) = b_5 + a_{5,1}i + a_{5,2}j + a_{5,3}k,$$

for the placement function of statement 5, with similar notations for other placement functions. Let us consider first edge 106 in table 1. The source is $\langle 5, i, j, k-1\rangle$ and the sink is $\langle 5, i, j, k\rangle$. As a consequence, the placement equations is simply: $a_{5,3} = 0$.

Consider now edge 107, whose source and sink are respectively $\langle 4, i, j\rangle$ and $\langle 5, i, j, k\rangle$. The placement equation is:

$$b_4 + a_{4,1}i + a_{4,2}j - b_5 - a_{5,1}i - a_{5,2}j - a_{5,3}k = 0.$$

In that case, however, the set $\mathcal{P}_{107}$ is not fully dimensional. In fact the governing predicate $k \leq 1$ and the constraint $k \geq 1$ in the domain imply $k = 1$. As a consequence, the placement equation reduces to:

$$\begin{aligned} b_4 - b_5 - a_{5,3} &= 0, \\ a_{4,1} - a_{5,1} &= 0, \\ a_{4,2} - a_{5,2} &= 0. \end{aligned}$$

The union of all such equations is a linear, homogeneous system which, in general, has only the trivial solution $a = 0$. The problem is to select a subset of this system which cuts as many important edges as possible, and which gives a non trivial distribution. It would be possible to use an exhaustive search algorithm, but the following greedy algorithm has been found to be quite satisfactory in most cases.

The idea is to order the rows of matrix C by decreasing importance, and to solve the system $Ca = 0$ by successive Gauss-Jordan elimination. The algorithm is as follows:

Algorithm E

1. Suppose that a partial solution has been found, in the form of a substitution σ. Initially, σ is the empty substitution.

2. Extract the next line of C and apply σ to it. There is nothing to do if the result is $0 = 0$. If not, write the resulting equation in the form:

$$x = f,$$

   where x is some component of a which has not yet been eliminated. Let τ be the elementary substitution $[x \leftarrow f]$; compute $\sigma' = \tau \circ \sigma$.

3. Apply σ' to all prototype placement functions and test whether any of them becomes trivial.

4. If there is no trivial prototype, replace σ by σ'.

5. Start again at step 2 until all rows of C have been used.

We have still to explain how to detect a trivial placement function. Since the original problem is homogeneous, the right hand sides in σ are homogeneous too. The uneliminated variables – those which do not occur in the left hand side of σ – may take arbitrary values. One easily proves that by giving proper values to the uneliminated variables, one may give non zero values to all variables with the exception of those which are explicitly set to zero by σ. As a consequence, we see that a placement function is not trivial provided one of its coefficients at least is not set to zero by the current solution σ.

Let us consider again the Cholesky solver, starting with edge 106. As we have seen earlier, the corresponding equation is $a_{5,3} = 0$. The first solution is then: $\sigma = [a_{5,3} \leftarrow 0]$. The next edge to be cut is 112. The associated equations are:

$$a_{5,3} = a_{6,1}, a_{5,2} = 0, a_{5,1} = a_{6,2}, b_5 = b_6.$$

The solution is now:

$$\sigma' = [a_{5,1} \leftarrow a_{6,2}, a_{5,2} \leftarrow 0, a_{6,1} \leftarrow 0, b_5 \leftarrow b_6, a_{5,3} \leftarrow 0].$$

If we try to cut edge 111 next, some of the placement functions become trivial. One of the equations for this edge is: $a_{6,2} = a_{5,2}$, which implies $a_{6,2} = 0$. At that time, all coefficients in the placement function for statement 6 are set to 0. As a consequence, we ignore the equation and try again with the next one.

The algorithm continues until all equations have been tested. At the end, we are left with only two arbitrary coefficients, $a_{6,2}$ and b_6. We may set the first to 1 and the second to 0. The end result is:

$$\begin{aligned} \pi(1,i) = i \quad &, \quad \pi(2,i,k) = i, \\ \pi(3,i) = i \quad &, \quad \pi(4,i) = i, \\ \pi(5,i,j,k) = i \quad &, \quad \pi(6,i,j) = j. \end{aligned} \tag{5}$$

Comparison with table 1 shows that all edges are cut except 108, 109 and 111. Execution of algorithm E is very fast: a Lisp-based implementation takes a few seconds on a low-end workstation.

4. Heuristics

In this section, I will address three problems. The first one is the determination of the order in which edges should be processed. The second one is the problem of distribution on a multidimensional grid of processors. The last one is the question of the selection of the folding function.

The following proposals will be justified by heuristics arguments. These will stem from an asymptotic analysis of the workload and communication volume. To simplify this analysis, we will suppose that the source program has only one size parameter, n, and that all dimensions of iteration domains, etc. are proportional to n. As a consequence, we will admit that the number of integer points in a d dimensional polyhedron of parameter n is $O(n^d)$.

4.1. Edge ranking

It is quite clear that an edge has a better chance of being cut if it is processed early by algorithm E. Hence, one should start with the edges which induce the largest traffic. The proposal is that we associate to each edge the volume of data which is exchanged if this edge is not cut, and that we rank edges by decreasing volume. However, we do not need a precise value for the said volume: any consistent estimate will suffice.

From the definition of the DFG, we see that the set of values which are sent along edge e is isomorphic to the image of $\mathcal{P}_e$ by the function h_e. We propose to use the dimension of this "emitter set" as a characterization of this volume. This suppose that the target computer has broadcasting facilities, i.e. that a value has to be sent only once even if it is used by many PE.

The emitter set is:

$$E_e = \{y \mid \exists x \in \mathcal{P}_e, y = h_e(x)\}.$$

One first eliminates x by a combination of Gauss-Jordan and Fourier-Motzkin algorithms. This gives a definition of E_e by a system of inequalities. One then constructs the set of implicit equalities which is satisfied by E_e; the dimension of E_e is the dimension of y minus the number of implicit inequalities.

Consider as an exemple edge 108 in table 1. The emitter set is:

$$\begin{aligned} E_{108} = \{i', j', k' \mid \exists i, j, k \quad : \quad & i' = i, j' = j, k' = i - 1, \\ & 2 \leq i \leq n, i + 1 \leq j \leq n\}. \end{aligned}$$

Obviously, all points in E_{108} satisfy $i' - 1 = k'$. Hence, the dimension of this set is 2.

The dimension of all emitter sets are given in the fourth column of table 1. From these, one may deduce that the volume of residual communications when using placement (5) will be $O(n^2)$.

4.2. Multidimensional Placement

The scheme we have just proposed has one major drawback: in some cases, the size of the virtual processor set may be less than the available parallelism, and less than the number of physical processors. Consider for instance the case of statement 5 in the Cholesky solver. Its $O(n^3)$ iterations are partitioned in about n fronts: hence, the mean paralellism is $O(n^2)$. In contrast the placement function (5) will generate only n virtual processors. Suppose that n is of the order of 100 and that there are about 1000 processors: there will be a severe loss of processing power.

Since in large distributed memory machines, the processors are most often organized as a multidimensional grid, one is naturally led to the consideration of multidimensional placement functions, each component of the function giving one coordinate of the virtual processor in the grid. There is a difficulty in this scheme: namely, that some fronts in the program may not have sufficient dimension to fill the grid. Suppose for instance that we try to implement Cholesky on a two dimensional grid. This is simple for statement 5, which has a two dimensional front, but what are we to do for 6, whose front is one dimensional?

There are two solutions here. The first one, if the hardware or software permits, is to rearrange the grid according to the statement – changing the geometry in Connection Machine parlance. In this way, there will be no loss of processing power. The drawback is that since changing the geometry is a non linear transform, minimizing communications will become very difficult.

The other possibility is to use the same geometry for the whole program, some processors being kept idle if necessary. One may note that this scheme is already used for one dimensional placement. Consider the case of statement 3: its fronts contain just one operation. Hence, only one virtual processor among n is in use at that time. The situation will be the same for statement 6 in a two dimensional grid: at each time tick, active processors will belong to a one dimensional subset.

Choosing between the two schemes is likely to depend strongly on detailed performances of the target computer, and should be the subject of further experiments. Here, I will explore the feasibility of the second proposal.

Computing higher dimensional placement may be done by an extension of algorithm E. The only change is in the triviality test. One requires that each candidate placement function depends on enough parameters that one may construct the required number of linearly independent solutions by giving suitable numerical values to the parameters. Obviously, one cannot impose this condition for statements whose fronts do not have the requested dimension. For instance, one cannot ask for a two dimensional placement function for statement 6 in Cholesky. In that case, one will obtain a placement function whose two components are not linearly independent. For Cholesky, the maximum possible dimension is two. Algorithm E with the new triviality test gives a prototype with two independent parameters:

$$\begin{aligned} \pi(1,i) = \alpha i \quad &, \quad \pi(2,i,k) = \alpha i, \\ \pi(3,i) = \alpha i \quad &, \quad \pi(4,i) = \alpha i + \beta j, \\ \pi(5,i,j,k) = \alpha i + \beta j \quad &, \quad \pi(6,i,j) = \alpha j. \end{aligned}$$

A two dimensional placement function may be obtained by successively setting $\alpha = 1, \beta = 0$, which gives again (5), and $\alpha = 0, \beta = 1$:

$$\begin{aligned} \pi'(1,i) = 0 \quad &, \quad \pi'(2,i,k) = 0, \\ \pi'(3,i) = 0 \quad &, \quad \pi'(4,i) = j, \\ \pi'(5,i,j,k) = j \quad &, \quad \pi'(6,i,j) = 0. \end{aligned}$$

4.3. Selection of the Folding Function

A smart choice of the folding function may help in reducing residual communication along uncut edges. This will happen if the source and sink operations belong to different virtual processors which are folded to the same PE. Let us consider the distance $d_e(x)$ of equation (2). If d_e is a constant (a constant vector in the case of multidimensional placement), one should choose a *block* folding function:

$$\chi(z) = z \div B,$$

whith a suitable block size B. If d_e depends on x, it does not seem possible to reduce traffic in this way. One should select a *cyclic* folding, which has better load equalization properties:

$$\chi(z) = z \bmod P.$$

5. Related work

The problem of automatically distributing arrays in a distributed memory computer has been widely discussed in the recent literature. Many authors works within the constraint satisfaction paradigm [7,8,9,10]. From an analysis of the source program, one deduce a constraint graph which indicates how the layout of the different arrays must be related in order to remove all communications. The set of constraints is usually inconsistent. An algorithm is then specified which aims at satisfying as many constraints as possible. Usually, the authors limit themselves to simple layouts; this is specially appropriate when the input language favors high level array operations, like Fortran 90 [8] or Alexi [11].

Nearest to our approach is the proposal of Ramanujan et. al. [12], which use affine placement functions and construct systems of equations like (4), but do not give a systematic method for solving them.

Lastly, the approach of Mace [13] is from a somewhat different point of view. The problem is how best to implement an array statement, given that there are several ways of distributing the data (storage pattern) each of which results in different costs for the operations. If one likens storage patterns to placement functions and loop nests to array statements, we see that our technique has the ability of providing the needed data to Mace's procedure, which works "in the large" while our own works "in the small".

6. Conclusions and Future Work

Our proposal has two characteristics:

- Each statement uses the same "geometry", whatever the dimension of its iteration space. This means that for lower dimensional statements, some processors will stay idle. As a compensation, communications are much easier to set up and optimize in this case.
- Each array has its own placement function, and this function is kept fixed for the entire execution of the program. This is in contrast with language which provide rearrangement directives.

Further research is needed to evaluate these two assumptions and explore alternatives. One should first experiment with other edge ranking schemes, giving for instance a much lower weight to fixed distance communications like edges 101 and 106. Next, the selection of a placement function should be more influenced by the particulars of the target architectures. Such problems as the existence of broadcast and partial broadcast mechanisms, or the provision of efficient reduction and scan primitives should be taken into account at this stage.

7. References

1. S. Hiranandani, Ken Kennedy, Charles Koelbel, Ulrike Kremer, and C-W. Teng. An overview of the fortran D programming system. Technical Report 91121, CRPC, Rice University, September 1991.
2. H. P. Zima, H. J. Bast, and M. Gerndt. SUPERB : A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.
3. David Loveman. High performance fortran. In Hans P. Zima, editor, *Proceedings of the Third Workshop on Compilers for Parallel Computers*, Vienna, July 1992.
4. Hans Zima, Peter Brezany, Barbara Chapman, Piyush Mehrotra, and A. Schawald. Vienna Fortran — a language specification. Technical Report 21, ICASE, 1992.
5. Paul Feautrier. Dataflow analysis of scalar and array references. *Int. J. of Parallel Programming*, 20(1):23–53, February 1991.
6. Thinking Machine Corp., Cambridge, MA. *CM Fortran Reference Manual, Version 5.2*, 1989.
7. Li Jinke and Marina Chen. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. In *Proc. Third Symp. on the Frontiers of Massively Parallel Computation*, pages 424–433. IEEE, October 90.
8. Kathleen Knobe, Joan D. Lukas, and Guy L. Steele. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *J. of Parallel and Distributed Computing*, 8, 1990.
9. Kathleen Knobe and Natarajan Venkataraman. Data optimization: Minimizing residual interprocessor data motion on SIMD machines. In *Proc. Third Symp. on the Frontiers of Massively Parallel Computation*, pages 416–423. IEEE, October 1990.
10. Manish Gupta and Prithviraj Banerjee. Demonstration of automatic data partionning techniques for parallelizing compilers on multicomputers. *IEEE Trans. on Parallel and Distributed Systems*, 3:179–193, March 1992.
11. Skef Wholey. Automatic data mapping for distributed-memory computers. In *ICS'92*, pages 25–34. ACM, 1992.

12. J. Ramanujan and P. Sadayappan. Compile-time techniques for data distribution in distributed memory machines. *IEEE Trans. on Paralell and Distributed Systems*, 2:472–482, October 1991.

13. Mary E. Mace. *Memory Storage Patterns in Paralell Processing.* Kluwer, 1987.