Automatic Distribution of Data and Computations

Paul Feautrier *

March 27, 2000

I said it in Hebrew – I said it in Dutch –
I said it in German and Greek,
But I wholly forgot (and it vexes me much)
That English is what you speak.

Lewis Carrol.

Abstract

The most critical factor in the performance of a distributed memory computer is the access frequency to remote data. This frequency may be reduced by a clever distribution of data and computations among processors and their memories. In the context of data parallel languages – as for instance, HPF – finding the proper distribution is the responsibility of the programmer. This paper explores another possibility, namely having the compiler determine the distribution using only information available in the source program. The paper shows that, with the help of elementary linear algebra techniques, one may find satisfactory placements provided the source program is limited to `DO` loops and arrays with affine subscripts.

*This report is a litteral translation by the author of "Distribution Automatique des Donnés et des Calculs", TSI vol. 15, pages 529-557, 1996.

Résumé

Pour un multiprocesseur à mémoire distribuée, le facteur de performance le plus critique est le taux d'accès à des informations éloignées. Ce taux d'accès peut être considérablement réduit si l'on distribue habilement les données et les calculs parmi les processeurs et leurs mémoires. Si l'on utilise un langage de programmation à parallélisme de données comme HPF, ce travail est de la responsabilité de l'utilisateur. On explore ici une autre possibilité, celle d'une distribution automatique à la compilation à partir des seules informations disponibles dans le texte source. On montre qu'il est possible, en utilisant des techniques peu complexes d'algèbre linéaire, de trouver des placements satisfaisants à condition que le programme source se limite à des boucles `DO` et à des tableaux indexés au moyen de fonctions affines.

1 Introduction

Modern computer applications need processing power far beyond what can be obtained from a commodity microprocessor. This is true in the field of High Performance Computation but also for Data Base Management and Discrete Event Simulation. One may try to implement High Performance Monoprocessors, but one is quickly limited by the performances of the then current technology. Moore law says that microprocessor performance increases by a factor of two each year and a half. There is no equivalent for mainframes and supercomputers, whose design cycle is much longer.

The obvious solution is the construction of supercomputers from standard parts. But this is by no means a simple task. Suppose that one needs, at a given time, twice the power of the most powerful current multiprocessor. One is tempted to build a biprocessor, but if this task takes more than 18 months, a better solution is to do nothing and wait for the next generation. A multiprocessor is interesting only if its degree of parallelism is high and if it can be implemented in a short time.

From that point of view, distributed memory computers have the advantage. They can be easily constructed by interconnecting commodity elements – for instance, ordinary workstations – using a communication network which can also be off the shelf, for instance an Ethernet or an ATM based switch.

The operating system can also be a standard Unix, although special libraries like PVM [BDG+91] have been specially designed for HPC applications. The programming model for such computers is *message passing*. Message passing occurs whenever a processor needs data which is located in the memory of another processor. Such a communication always has a high cost – most often, the equivalent of the execution time of several thousands of processor instructions. Hence, to obtain reasonable performances, the program must be designed in such a way that communications occurs very seldom.

Usually, parallel programs are classified according to their *grain*. Our contention is that there are several types of grains, and that comparing them allows a very crude evaluation of a parallel program performance.

The architecture grain, noted G in the following, is defined as the number of instructions which could have been executed in the time taken by a typical interaction between processors. To get a feeling for the importance of this parameter, consider a program which runs parallel phases on each processor, followed by a set of communication between all processors. Let N be the total number of instructions which are executed by the parallel phases. If we suppose that phases are perfectly balanced, the efficiency is easily seen to be:

$$\epsilon = \frac{1}{1 + PG/N}.$$

This clearly shows that the efficiency of the program decreases with the number of processors, which must be small with respect to N/G. The quantity N is is the grain of the parallel program. One can summarize these observations by saying that a good parallel program must have a grain which is much larger thant the grain of the architecture. Reducing the hardware grain is the task of the architect, while increasing the program grain is the problem of the programmer or of the compiler.

On the hardware side, high performance networks have been designed with a view of increasing throughput, and of reducing latency, and these efforts have had spectacular results. However, since at the same time the performance of processors has steadily increased, the hardware grain has stayed almost constant. The overal architecture of such supercomputers has not varied since the first Intel designs: the building block is an ordinary processor and its memory. Processors are connected through a network whose topology is chosen among a few basic possibilities: grids with dimension 1 to 3, hypercubes, multistage networks. The important point is that a

local memory access is faster by several order of magnitude than a message exchange. The actual performance of a program is governed by the frequency of inter-processors exchanges. Diminishing this frequency can be obtained by two methods:

- One starts from an arbitrary distribuation, and tries to improve it as the program is being executed, by migrating either programs or data. Some distributed operating systems provide tools for program migration. Shared virtual memory systemes move data as near as possible to the tasks which are using them.

- The other possibility is to build the program according to an optimal data distribution. This can be the responsibility of the programmer or of the compiler.

The first method has the advantage of simplicity: no preliminary analysis is needed. Besides, it is a dynamic solution, and there is no reason to believe that a unique solution can be optimal for the whole of a large program. The second method must be applied to reasonably sized kernels, with redistribution phases in between. This has the potential of giving better results, because the programmer or the compiler has, at least in theory, a complete view of the behaviour of the program, while the hardware or the operating system knows only its past. Even with complete information, finding a good distribution is a difficult optimization problem. The main goal of this paper is to explore ways and means of having the compiler choose an optimal (or nearly optimal) distribution.

1.1 An Elementary Example

Let us consider the following very simple example:

```
program A
x = y + z
```

To "distribute" this statement, we have to specify which processors hold `x`, `y` and `z`, and which processor execute the addition. We can for instance affect each variable to a different processor, and use a fourth processor for the computation. This distribution obviously entails three message exchanges. At the opposite, we can select a processor to hold all three variables and do

the calculation. In this case, no message passing is necessary. The second solution probably is the most efficient. It has no parallelism, but, after all, neither had the original program.

Consider now a similar example:

```
program B
do i = 1,n
    x(i) = y(i) + z(i)   {S}
end do
```

As a first try, one may consider that arrays `x`, `y` and `z`, and all iterations of statement S are to be distributed as a whole. One obtain results similar to those for program A, with either too many communications or not enough parallelism. Another choice is to handle the arrays on a word per word basis, and to consider each iteration of S independently. It is easy to see that if `x(i)`, `y(i)`, `z(i)` and iteration `i` of S are assigned to the same processor, no communication will be necessary. On the other hand, there is no constraint on the placement of words or iterations associated to distinct values of i. Let P be the number of available processors. One may for instance divide the interval $[1, n]$ into P (almost) equal segments, the corresponding strip of arrays `x`, `y` and `z`, and the corresponding iterations of S being attributed to the same processor.

This example shows the importance of another grain, the analysis grain, which characterize the smallest analyzable part of the source program as seen by the compiler. In this case, we have compared the results of a coarse grain and of a fine grain analysis, with an obvious advantage for the second solution. It is important to note that the parallel program grain is much larger (by a factor of n/P) than the grain of the analysis. There is a natural tendency to match the grain of analysis to the grain of the target architecture. Perhaps paradoxically, our example shows that it is better to run the analysis at the smallest possible grain and to leave it to the compiler to construct bigger chunks of operations by a process of *aggregation*. However, there is a level at which one gains nothing by further dissection of the operations. Experience shows that one execution of one high-level language statement (e.g. an assignment in Fortran or Pascal or C) is precise enough for automatic parallelisation. A finer grain might be needed for other optimizations, like ILP or register assignment or locality enhancement.

1.2 Related Work

Data distribution is the core problem in data-parallel programming languages [Bou93], and these languages have been recognized as the preferred method for programming distributed memory architectures. In the best known data-parallel language, [For94], the user has the responsibility of specifying the data distribution, and the compiler applies the "owner computes rule" to distribute the computations. The underlying hypothesis is that the user knows the data flow in his program, since this is to be an analogue of actual flows in the modeled system. The awkward consequence is that the compiler has to be able to translate any specification, whatever its efficiency or lack thereof. The reason is that the compiler has no control on the decisions of the programmer, and that human beings, while very good at finding efficient distributions when they exist, perform poorly when the problem is to find a compromise between necessary evils. This has lead to the search for automatic distribution algorithms.

Early work (see [JC90] for a typical example) used the constraint satisfaction paradigm. An analysis of the source program allows one to detect relations between array placement for an efficient execution. The result is the *affinity graph*, which is usually inconsistent. Various heuristics are used to satisfy as much constraints as possible. The placements which are found in this way are usually more limited than those which are dealt with in this paper.

The affine placement paradigm, which used here, was introduced by Ramanujam and Sadayappan in [RS91]. These authors did not give a universal method for solving the placement equations. In paper [Fea94], we introduced the technique of incremental Gaussian elimination, which will be described in Section 3.2.1. The authors of [BKK+94] linked the solution of the placement problem to the dimension of the kernel of a matrix known as the *communication matrix* of the program. However, their criterion is valid in the case of a perfectly nested loop only; more complicated tests are needed in the general case.

In a sequence of papers ([DR94a, DR93, DR94b]) Y. Robert et. al. attack the placement problem by techniques from linear algebra and matrix calculus. An important contribution of these papers is that several subproblems connected to placement are proved to be NP-hard. This is the justification for the use of heuristics in the solution. Papers [DR94a, DR93] are restricted

to the case where the subscript functions are translations in a perfect loop nest. More recently, paper [DR94b] handles arbitrary loop nests and arbitrary affine subscripts. The authors build an access graph, which is similar to the communication graph. The direction of each edge depends on whether the associated subscript function can be inverted or not. Finding a maximum weight branching for this graph gives the order in which placement function are to be computed. The edges which do not belong to the selected branching correspond to residual communications.

2 Formal Solution

Our objective is now to construct an efficient placement of data and computation at compile time. In this work, we will limit ourselves to the analysis of one procedure only. Finding a placement for several procedures is a very difficult problem, whether the placement is found automatically or "manually". In this section we will set up the problem and analyze its properties. The construction of a solution method is dealt with in the next section.

2.1 Notations and conventions

In the present state of the theory, there is no hope of finding a placement for an arbitrary program. In fact, even in the simplest languages – Fortran for instance – one can specifies dynamic access patterns, as for instance with subscript arrays:

```
program C
do i = 1,n
  a(b(i)) = ....
end do
```

In this example, one cannot characterize at compile time the relation between operations (one iteration of the loop body) and the array cells which it modifies, since one has to know the contents of array `b`. The only possibility is to restrict the allowed access patterns to a few simple constructions.

Firstly, one must be able to identify and to give a name to each *operation* of the program. An operation is the execution of one machine instruction or of one statement in a high level language, depending on the granularity

of the analysis. In ordinary programs, statements are repeated many times; each repetition has to be named. This is easy in the case of well structured programs with `DO` loops and conditionals. In this case, a repetition of a statement is named by giving the values of the surrounding loop counters. It is convenient to write these values, from the outermost loop to the innermost one, as a vector, the *iteration vector* of the statement. The name of the operation is a couple $\langle S, \vec{x} \rangle$ where S is the name of the statement and where $\vec{x}$ is the iteration vector. For each statement, the dimension of the iteration vector is fixed: it is equal to the number of surrounding loops. Each coordinate of the iteration vector cannot take arbitrary values: it is constrained by the bounds of the corresponding loop. The set of integer vector which satisfies the loop bounds is the *iteration domain* of the given statement. The iteration domain of S is $\mathcal{D}_S$. $\mathcal{D}$ is the set of all operations of the program, i.e. the disjoint union of all iteration domains.

If statement S is controlled by one or more tests, its iteration domain may be a subset of the iteration domain of the surrounding loops. In this case, a precise determination of its iteration domain may be impossible at compile time if the tests predicates are complex or depend on the input data of the programs. In this case, one has to resort to approximations [CBF95]. On the contrary, there is no difficulty at all if the conditional tests the sign of an affine form in the loop counters: we will suppose this is the case in all programs to be considered in this paper.

Consider:

```
program D
do i = 1,n
  do j = 1,i-1
    a(j,i) = a(i,j)       {S}
  end do
end do
```

The iteration domain of S is the set of integer points belonging to the triangle:

$$1 \leq i \leq n, \;\; 1 \leq j \leq i-1.$$

The objective of the compiler is, roughly, to distribute each operation to the processor which holds the maximum number of its operands. It thus has

to deduce from the program text the set of memory cells which are accessed by each operation. This is very complicated in general. The only simple case is that in which data structures are arrays or scalar. Furthermore, one has to restrict subscripts to affine functions of the surrounding loop counters. To simplify notations, the subscripts of an array are gathered into a *subscript vector*. With this convention, for each array access, there exist an affine transformation which associate a subscript vector to each iteration vector. Scalars are handled as 0-subscript vectors.

> Consider again example D. The transformation associated to the reference to `a` in the right hand side of S is:
>
> $$f(\vec{x}) = \begin{pmatrix} x_2 \\ x_1 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \vec{x},$$
>
> where $\vec{x}$ is the iteration vector. In its last version, this formula shows that f is associated to the matrix: $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$.

The placement problem for data and computations can now be formalized in the following way. The computation placement function Π associates to each operation $\langle S, \vec{x} \rangle$ a processor number $\Pi(S, \vec{x})$. Without loss of generality, one may suppose that processors are numbered, and hence that the value of Π is an integer (we will see later that it is sometime useful to have *integer vectors* as processor numbers).

In the same way, $\Pi(\mathtt{A}, \vec{i})$ is the number of the processor which holds the memory cell whose subscripts are $\vec{i}$ in array `A`. Π is the *placement function* of the program.

> If q is some processor number, then:
>
> $$\mathcal{D}_q = \{u \mid u \in \mathcal{D}, \Pi(u) = q\} \tag{1}$$
>
> is the set of operations which are executed by q. One can define in the same way the subarray of `A` which is resident in q memory. Placement functions must be such that these sets can be described in simple terms. Besides, the sets $\mathcal{D}_q$ must be of roughly the same size for load balancing.

A more general formulation can be obtained if Π is supposed to be a relation rather than a function. This convention allows the representation of redundancy, both for data and computations. This aspect of the placement problem is not well understood at present, and will be the subject of future research. A special case of the problem is solved in [BKK+94]: the duplication of read-only data.

2.2 Cutting Conditions

In this framework, it is easy to write the conditions that are to be satisfied by Π if one wants to remove all communications. Let S be a statement, $\vec{x}$ its iteration vector, $\mathtt{A}[f(\vec{x})]$ a reference to $\mathtt{A}$ in S. Operation $\langle S, \vec{x} \rangle$ is executed by processor $\Pi(S, \vec{x})$. Memory cell $\mathtt{A}[f(\vec{x})]$ is in the memory of processor $\Pi(\mathtt{A}, f(\vec{x}))$. No communication is necessary if these two processors are the same:

$$\Pi(S, \vec{x}) = \Pi(\mathtt{A}, f(\vec{x})). \tag{2}$$

These equations are called cutting conditions, since they have to be satisfied for communications between cell $\mathtt{A}[f(\vec{x})]$ and operation $\langle S, \vec{x} \rangle$ to be "cut".

The placement problem is thus seen to be equivalent to finding a function Π which satisfies all of the cutting conditions, for all statements in the program and all references therein. If such a solution does not exists, the set of residual communications due to reference $\mathtt{A}[f(\vec{x})]$ in statement S as the set:

$$\mathcal{R}_{S\mathtt{A}} = \{\vec{x} \mid \vec{x} \in \mathcal{D}_S, \Pi(S, \vec{x}) \neq \Pi(\mathtt{A}, f(\vec{x}))\} \tag{3}$$

and the problem is to minimize the sum of the sizes of all these sets[1]

An important point is that if we do not add constraints, the problem has a trivial solution, in which all data and all computations are assigned to one and only one processor. When constructing a distributed program, this *collapse* must be avoided at all costs. We have already encountered the necessary constraint: it is in fact the load balancing constraint, i.e. the stipulation that all sets $\mathcal{D}_q$ (see Equ. 1) have the same size.

[1] It is clear that our formalization is approximative. We are ignoring phenomena such as the reuse of the same value by several operations, or the possibility of moving a value from processor to processor as the computation proceeds.

2.3 Formal solution

In the above form, the problem has a simple solution. Let us consider the following bipartite graph. The vertices are operations and array cells. There is an edge between an operation and an array cell if the operation access the cell, whether as a read or as a write. This un-oriented graph is the *communication graph* of the program. Finding a placement is equivalent to associating a processor number to each vertex, and the cutting conditions express the fact that neighbours must reside on the same processor . By transitivity, this implies that all vertices on a path are on the same processor, and finally that all vertices in one connected component are the same processor.

One may thus find the *principal solution* of the placement problem by computing the connected components of the communication graph and assigning one processor to each component. If ϖ is the principal solution, all other solutions are found by composition of an arbitrary function:

$$\Pi = \xi \circ \varpi. \tag{4}$$

> When a placement is given, as above, by the composition of two functions, it is usual to say that function ϖ gives a *virtual processor number*, and that ξ specifies the distribution of virtual processors on real ones. This convention has been introduced by the designers of the Connexion Machine and is found, with a slightly different terminology in the HPF language.

A consequence of these observations is that the maximum degree of communication-free parallelism is an intrinsic characteristics of a program. In fact, the use of a folding function ξ is a way of reducing this degree (by assigning several connected components to the same processor) but it cannot be used to increase it. In fact most programs have a connected communication graph, hence no communication-free parallelism. Our aim is now to understand why, and to find ways of improving the situation.

2.4 Some elementary examples

Let us consider first the following transposition code:

```
program E
do i = 1,n
  do j = 1,i-1
    r = t(i,j)          {S1}
    t(i,j) = t(j,i)     {S2}
    t(j,i) = r          {S3}
  end do
end do
```

The connected component of `r` obviously includes all operations from S_1 and all operations from S_3. It then follows from the text of S_1 that this component include also all `t(i,j)`, for $j < i$, then, according to S_3, all `t(i,j)` for $j > i$. Lastly, all operations from S_2 are in the connected component of `r`. Hence, the communication graph of this program is connected, and the only way of obtaining a communication-free object program is to use only one processor. One sees clearly that this situation is due to the use of a unique scalar `s`. There is a well known remedy: one must *expand* `r` to, e.g., a two dimensional array :

```
program EE
do i = 1,n
  do j = 1,i-1
    r(i,j) = t(i,j)     {S1}
    t(i,j) = t(j,i)     {S2}
    t(j,i) = r(i,j)     {S3}
  end do
end do
```

This transformation can be done mechanically by using the results of the array dataflow analysis of the original program [Fea88, Fea91, MAL93, PW93]. In the communication graph of program EE, from the vertex representing cell `r(i,j)` for $j < i$, one may reach cells `t(i,j)` et `t(j,i)`, and also operations $\langle S_1, i, j\rangle, \langle S_2, i, j\rangle$ and $\langle S_3, i, j\rangle$. On the other hand, an operation whose vector is $\langle i, j\rangle$ accesses only array cells whose subscript are $\langle i, j\rangle$ or $\langle j, i\rangle$. There

can be a communication between iterations $\langle i,j\rangle$ and $\langle i',j'\rangle$ only in the case $i=j', j=i'$ which is excluded by the loop bounds[2].

All in all, the principal solution for program EE can be written:

$$\begin{aligned}\varpi(S_1,i,j) &= \varpi(S_2,i,j)=\varpi(S_3,i,j)=\varpi(\mathtt{t},i,j)=\varpi(\mathtt{r},i,j)= \\ &= \textbf{if } i\geq j \textbf{ then } \begin{pmatrix} i \\ j \end{pmatrix} \textbf{ else } \begin{pmatrix} j \\ i \end{pmatrix}\end{aligned}$$

One may observe that in this case, connected components have vector names. The corresponding program is communication-free and can be executed in constant time on $O(n^2)$ processors. This number can be reduced throught the use of a well choosen folding function.

Let us now consider a single assignment version of the matrix product code:

```
program F
do i = 1,n
  do j = 1,n
    c(i,j,0) = 0                                    {S1}
    do k = 1,n
      c(i,j,k) = c(i,j,k-1) + a(i,k)*b(k,j)         {S2}
    end do
  end do
end do
```

The cutting conditions are:

$$\begin{aligned}\Pi(S_1,i,j) &= \Pi(\mathtt{c},i,j,0) &(5)\\ \Pi(S_2,i,j,k) &= \Pi(\mathtt{c},i,j,k) &(6)\\ &= \Pi(\mathtt{c},i,j,k-1) &(7)\\ &= \Pi(\mathtt{a},i,k) &(8)\\ &= \Pi(\mathtt{b},k,j) &(9)\end{aligned}$$

[2]Remark that this reasoning depends on the hypothesis that two accesses to the same array with differing subscripts are accesses to differing memory cells, or, equivalently, that subscripts are always within the array dimensions. Almost all research in automatic parallelization implicitly accepts this hypothesis, which belongs to the *garbage in, garbage out* category.

One easily sees that (8) et (9), entails that $\Pi(S_2, i, j, k)$ does not depend on either i or j by using the rule: "if $\phi(x) = \psi(y)$ where x et y are distinct independent variables then ϕ and ψ are constant functions". For the same reason, (6) et (7) entails that $\Pi(\mathtt{c}, i, j, k)$ does not depend on k. All in all, these result imply that all Π functions are constant, that the communication graph is connected, and consequently that there is no communication-free parallel version of program F. Furthermore, since the object program was in single assignment form, the situation cannot be improved by expansion. It is clear, however, that many communications are linked to the use of read only variables, namely `a` and `b`. If one ignore the cutting conditions (8) and (9), one finds the principal solution :

$$\varpi(S_1, i, j) = \varpi(S_2, i, j, k) = \begin{pmatrix} i \\ j \end{pmatrix},$$

This solution can only be used if all or parts of `a` et `b` are replicated beforehand. This coupling by constant sharing occurs quite frequently and can be solved by data replication.

As a last example, consider the following Gaussian elimination code:

```
program G
do i = 1,n
  do j = i+1,n
    do k = i+1,n
      a(j,k) = a(j,k) - a(j,i)*a(i,k)/a(i,i)     {S}
    end do
  end do
end do
```

The cutting conditions are:

$$\Pi(S, i, j, k) = \Pi(\mathtt{a}, j, k) = \Pi(\mathtt{a}, j, i) = \Pi(\mathtt{a}, i, k) = \Pi(\mathtt{a}, i, i), \qquad (10)$$

and this also implies that Π is a constant: G has a connected communication graph. Here, it is neither a problem of scalar expansion nor of constant sharing, but an intrinsic property of the Gaussian algorithm. Finding a non trivial placement for G – and for a lot of similar programs - entails ignoring some of the cutting conditions, and this will generates residual communications. In the next section, I will present and discuss an algorithm for the

selection of residual communications and for the calculation of associated placement functions.

3 Solving the placement problem

In the simple examples above, the cutting conditions could be solved without recourse to any hypothesis on the shape of the placement functions. In the general case, the cutting conditions are very complex. Beside, the objective of this work is to find a placement *algorithm*; a collection of special techniques is not an adequate solution. There is small hope of reaching this goal unless we restrict the input programs, especially in the matter of the subscript functions. We will suppose here that all subscript functions are affine.

$$f(\vec{x}) = F\vec{x} + \vec{h}. \tag{11}$$

If $|S|$ is the number of loops surrounding statement S, and if $|\mathtt{A}|$ is the rank of array `A`, then in the above formula F is a matrix of dimension $|\mathtt{A}| \times |S|$ and $\vec{h}$ is a vector of dimension $|\mathtt{A}|$.

This hypothesis is frequently made by automatic parallelizers. Programs with only `DO` loops and affine subscripts were named *static control programs* in [Fea88]. There are reasons to believe, firstly that this is the only class of programs which have a well defined compilation algorithm toward parallel computers. Less constrained programs can be handled either by approximate methods [CBF95], or by run-time parallelization methods.

Secondly, the authors of [SLY89] have shown that a large proportion of numerical programs – about 80% – belongs to the static control class. An important research domain deals with methods for converting some subclasses of non static control programs to static control. Relevant methods include elimination of `GOTO` [Amm92], identification of inductive variables [ASU86], identification of `DO` loops and others.

But this hypothesis is not enough for solving the placement problem. One must also suppose that the unknown functions Π also are affine:

$$\Pi(S, \vec{x}) = P_S\vec{x} + \vec{q}_S, \tag{12}$$

$$\Pi(\mathtt{A}, \vec{x}) = P_{\mathtt{A}}\vec{x} + \vec{q}_{\mathtt{A}}. \tag{13}$$

Within this framework, the unknown are now the matrices P_S, $P_{\mathtt{A}}$ and the vectors $\vec{q}_S$ and $\vec{q}_{\mathtt{A}}$.

It is clear that this hypothesis is rather *ad hoc*. One may observe that a placement is uninteresting if it cannot be used as the blueprint for a parallel program. The present state of the art applies only to affine (or piecewise affine) placements [AI91, CFR95, Xue94, KP94].

It is clear nevertheless that one may build very simple examples with very complicated placements:

```
program H
do i = 1,n
  a(i) = a(2*i)
end do
```

The cutting condition for this program is:

$$\Pi(\mathtt{a}, i) = \Pi(\mathtt{a}, 2i).$$

All iterations whose counter is of the form $(2p+1).2^k$ clearly are in the same connected component. The principal solution of the placement problem is thus the function which associates to i its largest odd factor. This function is neither affine nor piecewise affine.

In formula (12), the dimensions of matrix P_S are $g \times |S|$. g is the dimension of the placement. The choice of the value of g depends both on the structure of the interconnection network and on the degree of parallelism of the program.

Most interconnection networks have a regular structure, since this allows the use of identical processors everywhere. Among the most frequently used regular networks are the *grids*: the processors have coordinates in $\mathbb{N}^d$, and each processor is connected to its $2d$ nearest neighbors. d is the dimension of the grid. Usual values are 1 (linear arrays), 2 and 3. On a d dimensional grid, the natural choice for processor names are d dimensional vectors, and the natural choice for g is d. One looks for a placement with the same dimension as the processor grid.

However, there is another point to consider. The interconnection network may not be a grid, or be a grid with adjustable dimension, as is the case for the hypercube. Let us suppose that in the source program, a loop generates about n iterations. An affine form on the loop counters will also have about n different values, whatever the number of independent variables. If one uses a g dimensional placement, the number of virtual processors will be of the order of n^g. This value must be compared on one side to the number of physical processors on the target architecture, and on the other side to the maximum degree of parallelism of the source code.

Let us consider the case of the inversion of a matrix of dimension $n = 1000$ on a distributed memory machine with about 100 processors. A one dimensional placement generates 1000 virtual processors, or a *vp-ratio* of 10.

Suppose now that the computer is somewhat like a CM-1, with 64k small processors. One has to switch to a two dimensional placement if the vp-ratio is to stay large enough. Lastly, if the dimension of the matrix is only 100, it is useless to generate a three dimensional placement, since the mean degree of parallelism of matrix inversion is only n^2.

Our conclusion is thus that the choice of the placement dimension is a complex problem, in which one has to take into account both the source program and the target computer. We feel that in most case, experimental evidence is the only way of selecting the best dimension. All we can do here is accept the value of g as a parameter of the placement problem; our algorithm has to be general enough not to depend on its precise value.

3.1 The Communication Matrix: the Unidimensional Case

In order to simplify the notations, we will first present the case of a one dimensional placement ($g = 1$). Such a placement is adapted to a linear or ring network. Matrices P_S and $P_{\mathtt{A}}$ of (12,13) becomes vectors $\vec{p}_S$ ans $\vec{p}_{\mathtt{A}}$. Similarly, vectors $\vec{q}_S$ and $\vec{q}_{\mathtt{A}}$ becomes scalars. The cutting condition (2) becomes:

$$\vec{p}_S.\vec{x} + q_S = \vec{p}_{\mathtt{A}}.(F\vec{x} + \vec{h}) + q_{\mathtt{A}}.$$

This equation summarizes as many constraints as there are points in the iteration domain of S. In fact, most of these constraints are redundant. It is easy to see that it is enough to have them satisfied at $d + 1$ points $\vec{x}_0, \ldots, \vec{x}_d$ to have them satisfied at all points of the affine subspace generated by these points. Most of the time, there are enough points in the iteration domain of S to generate the $|S|$ dimensional space itself. Among others, this space has for elements the origin and the vectors of a canonical basis. If we write the cutting condition at those points, we obtain:

$$\vec{p}_S = \vec{p}_{\mathtt{A}}F, \tag{14}$$

$$q_S = \vec{p}_{\mathtt{A}}.\vec{h} + q_{\mathtt{A}} \tag{15}$$

It may happen that the iteration domain of some statement is not of full dimension. An example is:

```
do i = 1,n
  do j = 1,n
    if(i.eq.j) then
      a(i,j) = 1.        {S}
    else
      a(i,j) = 0.
    end if
  end do
end do
```

It is always possible to handle this problem by constructing the supporting subspace of the iteration domain, then rewriting the loop in this subspace by a simple change of basis.

One builds the set of conditions for a communication-free placement by collecting all equations (14,15) for all references in the program. In the resulting system, the unknowns are the (components of) the vectors $\vec{p}_S$, $\vec{p}_{\mathtt{A}}$ and the scalars q_S et $q_{\mathtt{A}}$. The other terms, the F matrices and the $\vec{h}$ vectors, can be extracted from the program text, by syntactic analysis in simple cases, or by a more complex analysis, inductive variable detection for instance. The important point is that from the present point of view, each reference generates a linear *homogeneous* system of equations.

The two equations (14,15) are in fact quite different. The quantity:

$$d(S, \mathtt{A}, \vec{x}) = \Pi(S, \vec{x}) - \Pi(\mathtt{A}, f(\vec{x}))$$

is the "distance" from the processor executing iteration $\langle S, \vec{x} \rangle$ to the processor holding $\mathtt{A}[f(\vec{x})]$. This distance may have a physical interpretation — in the case of a grid — or not. However, it always gives a indication on the transmission delay. In the case of an affine placement, we get:

$$d(S, \mathtt{A}, \vec{x}) = (\vec{p}_S - \vec{p}_{\mathtt{A}} F).\vec{x} + q_S - \vec{p}_{\mathtt{A}}.\vec{h} - q_{\mathtt{A}}.$$

This formula shows that if (14) is satisfied, the communication distance does not depend on $\vec{x}$. On most interconnection networks, this kind of constant distance communication or *shift* is much faster than an arbitrary point to point communication. The ratio has been measured to be more than 30 on the CM-5 [Pla95]. If the equations (14) are satisfied, then one may try to satisfy (15). If this is possible,

all communications are eliminated. Nevertheless, this is not our primary objective: one may still have some constant distance communications without doing much harm to the program performance. Furthermore, the problems of satisfying (14) and (15) are separated here only for expository purposes; in our implementation, their handling is similar.

One can get a clearer appreciation of the problem to be solved by collecting all unknowns in a unique vector in which all $\vec{p}_S$ and $\vec{p}_{\mathtt{A}}$ are concatenated. The *placement vector* obtained in this way is of dimension:

$$N = \sum_S |S| + \sum_{\mathtt{A}} |\mathtt{A}|$$

The order in which statements and arrays are enumerated is arbitrary. We will write, for instance: $S < T$ to indicate that statement or array S comes before statement or array T in this enumeration.

With these definitions, equation (14) becomes:

$$\vec{p} C_{S\mathtt{A}} = 0,$$

where $C_{S\mathtt{A}}$ is the following bloc matrix:

$$C_{S\mathtt{A}} = \begin{pmatrix} Z_1 \\ I \\ Z_2 \\ -F \\ Z_3 \end{pmatrix}.$$

I is the unit matrix of dimension $|S| \times |S|$ and F is the subscript matrix of (11). The Z_i are null matrices with appropriate dimensions. For instance, Z_1 is of dimension $N_1 \times |S|$, where:

$$N_1 = \sum_{R<S} |R|.$$

Matrix $C_{S\mathtt{A}}$ is the *elementary communication matrix* for the reference to $\mathtt{A}$ in statement S. One then collect all such equations in the form:

$$\vec{p} C = 0. \tag{16}$$

Matrix C is obtained by concatenating matrices $C_{S\mathtt{A}}$[3] and is the *communication matrix* of the whole program. When formulated in this way, the solution is obvious.

[3]Our formulation is predicated on the hypothesis that each statement has at most one reference per array. The general case just requires more complicated notations.

The placement vector $\vec{p}$ can be chosen arbitrarily in the kernel of C. An eventual collapse on one processor occurs when C is of full row rank, and its kernel is the trivial subspace $\{\vec{0}\}$, which corresponds to a one processor placement.

> There is no collapse for example D above. The problem is to place statement S and array `a`, whose domains are two-dimensional. Hence, $\vec{p}$ has dimension 4. There are two subscript matrices:
>
> $$F_1 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \; F_2 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}.$$
>
> Substituting these matrices into (14) and concatenating, one obtains the following communication matrix:
>
> $$C = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & -1 & -1 & 0 \\ -1 & 0 & 0 & -1 \end{pmatrix}.$$
>
> The reader will readily find that the kernel of C is generated by the vector $(1, 1, 1, 1)$.

3.2 A Greedy Algorithm

When the communication matrix has a null kernel, we cannot find communication free-parallelism. Either the program stays sequential, or we have to accept *residual communications*, which is equivalent to ignoring some of the elementary communication matrices. One obtain in this way a partial communication matrix C', and it seems plausible that if one ignore enough references, C' will have a non trivial kernel. As an extreme solution, if all references are ignored, C' becomes empty, its kernel is the whole space, and the placement can be chosen arbitrarily.

It might seem that the correct approach would be to compute the communication volume according to (3) and to find its minimum. But this is a useless refinement. In fact, the set of all iterations giving rise to a communication is:

$$\mathcal{R}_{S\mathtt{A}} = \mathcal{D}_S - \{\vec{x} \mid \vec{x} \in \mathcal{D}_S, d(S, \mathtt{A}, \vec{x}) = 0\}.$$

If the communication distance is null, then $\mathcal{R}_{S\mathtt{A}}$ is empty. Otherwise, let n be the order of magnitude of the size of loops. The size of $\mathcal{D}_S$ is $O(n^{|S|})$. On the other hand, since the equation $d = 0$ define an hyperplane, the size of the second term is $O(n^{|S|-1})$, which is negligible when n is large. The outcome of these

estimates is that our reasoning can be "all or nothing" : either the elementary communication matrix $C_{S\mathtt{A}}$ is part of C', and the corresponding communications are eliminated[4] or $C_{S\mathtt{A}}$ is not taken into account, and this generates a volume of communication which has to be estimated, but which is nearly independent of the chosen placement. A technique for estimating this volume is given in section 3.2.2.

Let us summarize our findings. Our problem is to select which elementary communication matrices are going to be part of C', the constraint being that C' kernel is non trivial and the objective being that the volume of residual communications is minimal. This suggests the use of a greedy algorithm, which is described below. In the body of the algorithm, we suppose that all references in the program have been numbered from 1 to L, that the elementary communication matrix corresponding to reference number k is C_k, and that reference are listed by order of decreasing communication volume.

Algorithm E

1. Initially, C' is the empty matrix.
2. For $k = 1, L$:
   (a) Construct $C'' = \begin{pmatrix} C' & C_k \end{pmatrix}$
   (b) If $\ker(C'') \neq \{0\}$, then $C' = C''$.
3. Any vector in $\ker(C')$ can be used as a placement vector.

   The test in step 2b, which decide whether $\ker(C')$ has a non zero vector, will be called the *triviality test* in the following.

As for any greedy algorithm, there is no guarantee that the solution will be an optimum. Algorithm E can readily be transformed into a branch and bound one in the following way. We suppose that to each reference is associated a weight w_k which is a measure (in some sense) of the volume of communication generated by the corresponding reference. The algorithm build a solution tree in which decision are taken sequentially. To each node of the problem tree are associated the sets A (of accepted references) and D (of discarded references). References are here identified to their communication matrices. In a node of depth k, all references from 1 to k have been classified:

$$A \cup D = [1, k],\ A \cap D = \emptyset.$$

[4] or, in the worst case, transformed into shifts.

The value of the node is $\sum_{i \in D} w_i$, i.e. the volume of the residual communications. In the course of the algorithm, one computes the best current solution, of value W.

Algorithm B

1. If the communiction matrix of the current node has a trivial kernel, this node is a failure.

2. If the current node has height L, this node is a success. Adjust the best value W accordingly.

3. If the value of the current node is larger than W, it is a failure.

4. Otherwise, let k be the height of the current node. One constructs its left son by adding C_{k+1} to A, and its right son by adding it to D.

Limited experience with this algorithm has shown that results are not significantly better than those of the greedy algorithm. The greedy algorithm itself is just the first part of algorithm B (the search for the first feasible solution, provided one explore the solution tree depth first, the left son being developed first).

These algorithms are just skeletons. To flesh them up, one has to explain how to efficiently construct the kernel of C, how to order the references by decreasing communication volume, and, lastly, how to select the offset constants when the placement vector is known.

3.2.1 Computing Kernels

From the point of view of efficiency, the important point in the computation of kernels is to make use of the fact that communication matrices are constructed incrementally. The best solution is to reformulate the problem as the solution of a system of linear and homogeneous equations [Fea94]. One introduces variable names to represent the components of $\vec{p}$, let us say $p_1, \ldots, p_N$. The current solution can be represented as a substitution σ which replaces some of the p_i by linear forms in the other unknowns. In what follows, symbol $\mathcal{C}$ represents the system of equations:

$$(p_1, \ldots, p_N)C = 0$$

associated to the matrix C. At any given step of the algorithm, let σ be the solution associated to the current communication matrix. A base for its kernel is obtained simply by applying σ to the vector $(p_1, \ldots, p_N)$ and separating the

coefficients of the remaining unknowns. Hence, the dimension of the kernel is the number of independent unknowns in σ.

Let $\mathcal{C}_k$ be the system associated to elementary communication matrix C_k and let $\mathcal{C}$ be the current communication matrix with solution σ. The first step is to apply σ to $\mathcal{C}_k$ and to eliminate trivial rows ($0 = 0$) and redundant rows. The remaining rows are a system of linear homogeneous equations in the remaining unknowns, which is solved by Gaussian elimination. One obtain a new substitution, τ, and one build $\sigma' = \sigma \circ \tau$. The kernel of the new communication matrix is trivial if and only if σ' assign the value 0 to all variables.

> Let us return to program D. Here the two references have the same communication volume and can be solved in any order. The system which is associated to reference 1 is:
>
> $$\begin{aligned} p_1 - p_4 &= 0 \\ p_2 - p_3 &= 0 \end{aligned}$$
>
> and its solution is $\sigma_1 = [p_1 \leftarrow p_4, p_2 \leftarrow p_3]$. Since there are two residual unknowns, the corresponding kernel is of dimension 2.
>
> The next system is:
>
> $$\begin{aligned} p_1 - p_3 &= 0 \\ p_2 - p_4 &= 0 \end{aligned}$$
>
> If one applies σ_1 to it, one obtain the equation $p_4 - p_3 = 0$ repeated twice. to which corresponds $\tau = [p_3 \leftarrow p_4]$, then $\sigma_2 = [p_1 \leftarrow p_4, p_2 \leftarrow p_4, p_3 \leftarrow p_4]$. This gives the following placement vector: $p_4(1, 1, 1, 1)$, which is equivalent to the kernel we found earlier.

3.2.2 Ordering References

To apply algorithm E, the references in the program have to be ordered by decreasing communication volume. Here, we need some information on the target architecture. The simplest case is that of a NUMA machine (Non Uniform Memory Access): each processor can read and write in the memory of of other processors, but a remote memory access takes much longer than a local memory access. In the object code, there is no difference between between local and remote accesses. The communication volume of a reference is thus equal to the number of iterations of this reference, i.e. to the volume of[5] $\mathcal{D}_S$.

[5]Here, the volume of a subset of $\mathbb{N}^d$ is taken to be the number of points with integer coordinates which belong to this subset.

In fact, most NUMA machines have coherent caches. When a memory cell is accessed, a copy is stored in the cache, and successive accesses are local. If another processor modify the distinguished memory cell, the coherence mechanism insure that the local copy is invalidated, which implies that the next access will be distant. The behavior of a message passing architecture is similar. When a memory cell has been accessed by a processor, it can be kept in local memory as long as it is not modified by another processor. When this happens, a new message exchange is necessary. One may summarize this analysis by saying that what must be counted is not the number of memory cells but the number of values.

> Let us evaluate the communications volumes associated to the references in program G. A detailed study of the program shows, for instance, that the value in `a(i,i)` does not change while the loops on `j` and `k` are executed. This is a consequence of the values of the lower bounds of these loops. On the other hand, the value of `a(i,i)` changes at each iteration of the `i` loop; the corresponding communication volume is thus n.
>
> On the contrary, it is easy to see that each read access to `a(j,k)` returns a new value, hence the traffic is $O(n^3)$. One may also observe that, since we cannot have any information on the coefficients of the matrix, one has to suppose that the values written into `a` are distinct. Hence, the communication volume of the left hand side reference in any statement is always taken to be equal to the volume of $\mathcal{D}_S$. We leave it to the reader to deduce that the communication volume associated to the two remaining references is $O(n^2)$.

The important point is that this analysis is not a direct syntactic outcome of the source program. It would be a gross error to say that since reference `a(j,k)` depends on two subscripts whose range is $[1, n]$, the generated traffic is $O(n^2)$. What is needed here is a dataflow analysis in the spirit of [Fea91], in which the interested reader will find a complete study of program G. This analysis gives, for each read, the name of the operation which produced the corresponding value, as a function of the name of the read operation. In the present framework, this function is affine or piecewise affine.

In all cases, one is left with the problem of counting the points in a subset of $\mathbb{N}^d$. This is feasible since the loop bounds are affine, and hence since the relevant subsets are included in polyhedra [Taw91]. However, the counting algorithm is quite complex, the result is a polynomial on the loop bounds, and may be difficult to interpret. Lastly, it is not evident that such precision is really necessary. Experience has shown that evaluating the order of magnitude of the communication

volume is quite sufficient. This order of magnitude is directly related to the dimension of the polyhedron which bounds the communication points. In the case where this polyhedron is $\mathcal{D}_S$, this dimension is $|S|$ with the exception of some pathological cases. If the communication volume is bounded by a set of the form $f(\mathcal{D}_S)$, where f is the source function as given by dataflow analysis, the computation of its dimension is more complex since one has to decide whether f is one-to-one or not. But since, by hypothesis, f is an affine function, this is another case of Gaussian elimination. One finds that one obtains good results if, for the use of algorithm E, one order references by decreasing dimension[6]..

3.2.3 Computing Shift Constants

The same reasoning as above can be applied to the resolution of [15]. This is a non homogeneous linear system which may or may not be overdeterminate, as in the following case:

```
program K
do i = 2,n
  a(i) = a(i-1)
end do
```

Here again, the method is to sort the references in order of decreasing traffic, and then to solve this system incrementally, with backtracking when an impossibility is found. This sub-algorithm can easily be integrated into algorithm E: one only has to order equations in the proper way.

3.3 The "Owner Computes" Rule and the Single Assignment Form

The "owner computes" rule edicts that a computation is always done on the processor which holds the results. In our formalism, this rule translate to the rule that left hand side references must always be included in the communication matrix.

> We have seen earlier a justification for this rule: in all cases, the communication volume for the left hand side reference is at least equal to the volume of any other reference in the same statement. The other justification is that if one complies with the "owner computes" rule,

[6]Note added by the translator: Nowadays, I would probably amend this discussion to take into account the revival of Ehrhardt polynomials due to Philippe Clauss.

the target program is simplified since one does not have to implement a remote write protocol.

The "owner computes" rule can always be enforced, provided there is no conflict with the desired placement dimension. For instance, there is no non-trivial placement for:

```
do i = 1,n
  s = ...
end do
```

if the "owner computes" rule is enforced. This is never a problem if the program arrays have been extended up to a single assignment form, which can always be done as a by-product of array dataflow analysis. In this case, all assignment are of the form:

do $\vec{x} \in \mathcal{D}_S$
  $\mathtt{A}_S[\vec{x}] = \cdots$
end do

and each write is to a distinct array cell. The cutting condition for the left hand side reference is:

$$\Pi(S, \vec{x}) = \Pi(\mathtt{A}_S, \vec{x}).$$

Enforcing the "owner computes" rule is equivalent to using this equation to eliminate either $\Pi(S, .)$ or $\Pi(\mathtt{A}_S, .)$.

Nevertheless, the "owner computes" rule is one more constraint on the placement, and, as all additional constraints, it may lower the quality of the result. The reader will find an example of this phenomenon in [DR93].

3.4 Multidimensional Placement

One has already seen that in some cases, using a scalar placement function may not result in enough parallelism for the target architecture. The obvious solution (which is suggested by the structure of grid networks) is to use multidimensional placement functions; let g be the dimension of the placement. All findings of the preceding section are still valid. In placement prototypes [12,13], one replace vectors $\vec{p}_S, \vec{p}_{\mathtt{A}}$ by matrices $P_S, P_{\mathtt{A}}$. One may consider a global placement matrix P of dimension $N \times g$, which must satisfies the analogue of [16]:

$$PC = 0.$$

The communication matrix C is the same as in the one-dimensional case. It is clear that the row vectors of P are solutions to [16], hence belong to the kernel of C. The solution of the problem is then obvious: to construct a g-dimensional placement, one selects g linearly independent vectors in $\ker(C)$.

But this condition, while necessary, is not sufficient. If the chosen solution is such that the row vectors of one of the P_S matrices are linearly dependent, statement S will be executed only by a subset of the available processors, and this entails a loss of processing power. We have to use a stronger triviality test in the greedy algorithm. The C' matrix is satisfactory for step 2b of algorithm E if its kernel is spanned by at least g vectors whose projection in each of the iteration spaces of the program statements are linearly independent. Note however that this is impossible if the iteration space has dimension less than g. The first conclusion is that it is useless to chose a g which is larger than the maximum nesting level of the program. But, even in cases where this constraint is satisfied, it may happen that some statement has nesting level less than g. One must accept a performance loss in this case. If the kernels are computed by the incremental Gauss algorithm we have defined above (see section 3.2.1), this extended triviality test is quite simple. It is enough to check that the placement vector of each statement depends at least on g independent variables.

It is clear that the triviality test will grow more stringent when g increases. It follows from this remark that the amount of residual communication increases with g. Whether this effect is compensated by increased communication bandwidth and increased parallelism can only be judged experimentally.

3.5 Example

Let us select a placement for example G. There is one statement at depth 3, and one array of rank 2, hence 5 unknown coefficients p_1 to p_5. On the other hand, there are 5 array references, but two of them are the same, hence there are only four cutting conditions. We can in that case give the explicit counterpart of [16]:

$$\begin{array}{lll} p_1 = 0, & p_2 = p_4, & p_3 = p_5 \\ p_1 = p_5, & p_2 = p_4, & p_3 = 0 \\ p_1 = p_4, & p_2 = 0, & p_3 = p_5 \\ p_1 = p_4 + p_5, & p_2 = 0, & p_3 = 0 \end{array}$$

The communication volumes associated to each reference have been computed in section 3.2.2. One finds that the left hand side and the first right hand side reference generates a communication volume of order n^3, the next two reference a volume of the order of n^2, and the last reference a volume of order n. As a

consequence, it is natural to follow the owner compute rule. Applying Gaussian elimination to the first three equations gives the solution:

$$\sigma_1 = [p_1 \leftarrow 0, p_2 \leftarrow p_4, p_3 \leftarrow p_5].$$

There are still two free variables. Hence, we can construct a placement of dimension 2. If the above solution is applied to the next equation one find:

$$0 = p_5,\ 0 = 0,\ p_5 = 0.$$

whence the new solution:

$$\sigma_2 = [p_1 \leftarrow 0, p_2 \leftarrow p_4, p_3 \leftarrow 0, p_5 \leftarrow 0].$$

There is only one free variable left. Hence, the placement is only one-dimensional. The reader will easily show that, if one attempts to satisfy the remaining equations, the placement becomes trivial. The computation of shift constants is left to the reader: they are found to be 0 in all cases.

We conclude that example G, has, first of all, a two dimensional placement:

$$\Pi(S, i, j, k) = \begin{pmatrix} j \\ k \end{pmatrix},$$

which generates two residual communications of volume n^2 and a communication of volume n, and a one-dimensional placement: $\Pi(S, i, j, k) = j$, which generates only one communication of volume n^2 and one communication of volume n. If the elimination order is changed, one may also find the symmetric placement: $\Pi(S, i, j, k) = k$.

3.6 Distribution

The number of points in the range of the placement function is equal to the number of virtual processors that is needed for running the target program. There is no reason for this number to be equal the number of physical processors in the target machine. One has to use a *folding function* which can be quite arbitrary. The choice of a folding function corresponds to the selection of a distribution in data parallel languages. One uses simple distributions, like block distributions:

$$\xi(x) = x \div B,$$

cyclic distributions:

$$\xi(x) = x \bmod P,$$

or block cyclic distribution:

$$\xi(x) = (x \div B) \bmod P.$$

One criterion is the study of residual communications. If those are uniform, the use of a block distribution allows an additional reduction of the communication volume. On the other hand, block distributions have no particular advantages in the case of general distributions. One should select cyclic distribution for better load balance.

One may suggest the following rules:

- If most of the residual communications have no particular patterns, use a cyclic distribution.
- If the residual communications are uniform, select a bloc cyclic distribution. The size of each bloc must be larger than the length of the communication vectors.

3.7 Code Generation

The problem of generating the parallel code for a given placement is beyond the scope of this paper. There are in fact many active researchers in the field.

One of the simplest methods [ZBG88] consists in having each processor execute a copy of the original program, each statement S being guarded:

$$\textbf{if } \Pi(S, \vec{x}) = q \textbf{ then } S$$

where q is the name of the current processor. Similarly, to each reference is associated a conditional which decides if the reference is local, or if it must be sent to or received from another processor. In both these cases, one must write a message passing routine. This task might be simplified by calling a message passing library like [BDG+91]. The resulting program is usually very inefficient, but it can be optimized by polyhedra scanning techniques [AI91, CFR95, Xue94, KP94].

4 Conclusion

The proposals in this paper can be summarized as follows:

- Have the program submitted to a preprocessing phase, not formalized here, in which arrray accesses are normalized, the granularity of parallelism and the loop size is determined, the dataflow is computed, and arrays and scalars are expanded if necessary.

- Chose the dimension of the placement functions, taking into account both the parameters of the target architecture (number of processors, network topology), and the characteristics of the source program.
- Apply algorithm E to find a placement with minimal residual communications.
- Chose a distribution function according to Section 3.6 and generate the target program according to Section 3.7.

There has been several implementations of this method for various target architectures and target languages. One of them starts from Fortran and generate data parallel code like CM-Fortran or CRAFT for the Cray T3D. Compilation times are acceptable and the performances of object programs are good, giving in some cases better results than the T3D library. There is obviously a lot of work to do for transforming these pilot implementations into useful compilers.

The above technique can be explained in the context of recent research on automatic parallelization, [Fea92c, Fea95], in which a parallel program is represented as a partial order on its operations. Scheduling techniques [Fea92a, Fea92b] look for sets of unordered operations (anti-chains of the parallel order), and are well adapted to synchronous architectures. Placement, on the contrary, looks for chains (sets of totally ordered operations), and corresponds to distributed architectures. Both methods are full-fledged compilation techniques. Their behaviors are exactly opposite: scheduling adds edges to the given partial order until it can be represented as a sequence of parallel constructs. This is the `SEQ of PAR` form of [Bou93]. Conversely, the placement method *remove* edges until the execution order takes the form of a parallel composition of sequential processes – the `PAR of SEQ` form. Since edges have been removed, they must be reinserted as residual communications.

There is no a priori reason for preferring one or the other method. Some orders are brought to the `SEQ of PAR` form by adding very few edges, others become `PAR of SEQ` by suppressing very few edges. On the other hand, each parallel architecture has a prefered form, and in some cases – SIMD machines and systolic arrays - one needs both a schedule and a placement for the generation of the parallel program. The reader is referred to [Fea95] for a discussion of these cases.

In the rare cases where the original program is already in the `PAR of SEQ` form, the algorithm should find directly all independent chains. As example D shows, this is not really the case. A straightforward analysis gives a piecewise bidimensional placement analogous to the one in example EE, giving $O(n^2)$ chains.

The proposed method gives only one placement vector, giving only $O(n)$ chains. It would be interesting to extend the present method to more general placement functions. Example H shows that simple problems may have very complicated placements functions, thus limiting progress in this direction.

Another constraint results from the fact that placements and distributions are only intermediates for the ultimate construction of the parallel code. With present knowledge, this is possible only for affine and piecewise affine placements. Example H shows that very simple subscript functions generate complex placements, which cannot be converted into parallel programs in a simple and regular way. It thus seems hopeless to handle complex subscripts. A compromise solution is to ignore complex subscripts when selecting the placement, and to have them reappear as residual communications. One can only hope that they will not result in too much performance loss.

It may be possible to construct a placement from an approximate analysis of the source program [CBF95], but this is a fully unexplored avenue.

Some distributed architectures have been optimized for handling efficiently some types of communications. This is often the case for uniform communications, but there are also broadcasts (a communication from one to many processors), and reductions (a communication from many to one processor). Recent studies [Pla95] have shown that taking these peculiarities into account results in greatly increased performance.

Lastly, it seems clear that when the number of statements in the program increases, the number of constraints increases faster than the number of free parameters. Hence, the quality of the placement decreases. It thus seems interesting to dissect a program into phases which are processed independently, with redistribution phases in between if necessary. Redistribution operations are provided in data parallel languages like HPL. If the basic technology is simple (one just has to rename arrays), the choice of cut points is difficult and is a worthwhile subject for further research.

References

[AI91] Corinne Ancourt and François Irigoin. Scanning polyhedra with DO loops. In *Proc. third SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 39–50. ACM Press, April 1991.

[Amm92] Zahira Ammarguellat. A control-flow normalization algorithm and its complexity. *IEEE Transactions on Software Engineering*, 18(3):237–251, March 1992.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, Reading, Mass, 1986.

[BDG+91] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. A user guide to pvm: Parallel virtual machine. Technical Report ORNL/TM-11826, Oak Ridge National Laboratory, July 1991.

[BKK+94] David Bau, Indupras Kodukula, Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. Solving alignment using elementary linear algebra. In *Seventh Annual Workshop on Languages and Compilers for Parallel Computing*, pages 46–60. Springer-Verlag, LNCS 892, August 1994.

[Bou93] Luc Bougé. Le modèle de programmation à parallélisme de donnés : une perspective sémantique. *T.S.I.*, 12(5):541–562, 1993.

[CBF95] Jean-François Collard, Denis Barthou, and Paul Feautrier. Fuzzy array dataflow analysis. In *ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming.* ACM, July 1995.

[CFR95] Jean-François Collard, Paul Feautrier, and Tanguy Risset. Construction of `do` loops from systems of affine constraints. *Parallel Processing Letters*, 5(3):421–436, 1995.

[DR93] Alain Darte and Yves Robert. A graph-theoretic approach to the alignment problem. Technical Report 93-20, LIP-IMAG, July 1993.

[DR94a] Alain Darte and Yves Robert. Mapping uniform loop nests onto distributed memory architectures. *Parallel Computing*, 20:679–710, 1994.

[DR94b] Michèle Dion and Yves Robert. Mapping affine loop nests: New results. Technical Report 94-30, LIP, 1994.

[Fea88] Paul Feautrier. Array expansion. In *ACM Int. Conf. on Supercomputing*, pages 429–441, 1988.

[Fea91] Paul Feautrier. Dataflow analysis of scalar and array references. *Int. J. of Parallel Programming*, 20(1):23–53, February 1991.

[Fea92a] Paul Feautrier. Some efficient solutions to the affine scheduling problem, I, one dimensional time. *Int. J. of Parallel Programming*, 21(5):313–348, October 1992.

[Fea92b] Paul Feautrier. Some efficient solutions to the affine scheduling problem, II, multidimensional time. *Int. J. of Parallel Programming*, 21(6):389–420, December 1992.

[Fea92c] Paul Feautrier. Techniques de parallélisation. In M. Cosnard, M. Nivat, and Y. Robert, editors, *Algorithmique Parallèle*, pages 243–257. Masson, May 1992.

[Fea94] Paul Feautrier. Toward automatic distribution. *Parallel Processing Letters*, 4(3):233–244, 1994.

[Fea95] Paul Feautrier. Compiling for massively parallel architectures: a perspective. *Microprogramming and microprocessing*, 1995. à paraître.

[For94] High Performance Fortran Forum. High performance fortran language specification, version 1.1. Technical report, Rice University, November 1994.

[JC90] 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. lEEE, October 90.

[KP94] Wayne Kelly and William Pugh. Selecting affine mappings based on performance estimations. *Parallel Processing Letters*, 4(3):205–220, September 1994.

[MAL93] Dror E. Maydan, Saman P. Amarasinghe, and Monica S. Lam. Array dataflow analysis and its use in array privatization. In *Proc. of ACM Conf. on Principles of Programming Languages*, pages 2–15, January 1993.

[Pla95] Alexis Platonoff. *Contribution à la distribution automatique des données pour machines massivement parallèles.* PhD thesis, Université P. et M. Curie, March 1995.

[PW93] William Pugh and David Wonnacott. An evaluation of exact methods for analysis of value-based array data dependences. In *Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing*, pages 546–566. Springer-Verlag LNCS 768, August 1993.

[RS91] 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.

[SLY89] Zhiyu Shen, Zhiyuan Li, and Pen-Chung Yew. An empirical study on array subscripts and data dependencies. In *1989 Int. Conf. on Parallel Processing*, pages II 145–152, 1989.

[Taw91] Nadia Tawbi. *Parallélisation Automatique : Estimation des Durées d'Exécution et Allocation Statique de Processeurs.* PhD thesis, Université P. et M. Curie, Paris, 1991. 19 Avril 1991.

[Xue94] J. Xue. Automating non-unimodular loop transformations for massive parallelism. *Parallel Computing*, 20(5):711–728, May 1994.

[ZBG88] H. P. Zima, H. J. Bast, and M. Gerndt. SUPERB : A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.