Toward Automatic Distribution

Paul Feautrier

Laboratoire MASI, Institut Blaise Pascal
Université de Versailles St-Quentin
45 Avenue des Etats-Unis
78035 VERSAILLES CEDEX FRANCE

e-mail : `feautrier@masi.ibp.fr`

December 4, 1992

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 compute" rule and is reminiscent of two-level distribution schemes, like HPF's `ALIGN` and `DISTRIBUTE` directives, or the CM-2 virtual processor system.

Résumé

Dans cet article, on considère le problème de la distribution du code et des données d'un programme entre les processeurs d'un ordinateur à mémoire distribuée. Si le programme source est suffisamment simple pour permettre une analyse de flot, on peut déterminer une fonction de placement à l'aide d'un algorithme simple analogue à l'algorithme de Gauss. La fonction de placement caractérise complétement la distribution en donnant l'identité du processeur virtuel qui exécute chaque opération élémentaire. Il ne reste plus ensuite qu'à "réaliser" les processeurs virtuels sur les processeurs physiques de la machine cible. Le résultat de cette analyse satisfait

à la règle des écritures locales et ressemble à des schémas de distribution à deux étages bien connus, comme les directives `ALIGN` et `DISTRIBUTE` qui ont été proposées pour HPF, ou le système de processeurs virtuels de la CM-2.

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 between processors. 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; one gets in this way *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, while this is taken care of dynamically by a combination of hardware and software in the case of global memory and DVM.

Several authors have proposed that the distribution be specified by the programmer [HKK+91, ZBG88, Lov92, ZBC+92]. The rationale for this proposal 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 obeyed, 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 almost all communications by local operations. Such a distribution is specified in term of a *placement function*. Section 3 is devoted to the design of an algorithm for the selection of a good placement function. This algorithm make use of an ordering of the communication patterns, which is discussed in section 4. 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. The differences lie both in the programming style and in the performances. Programming style will be ignored in this paper. We will assume that, once a good distribution has been found, it will be the job of a later phase of the compiler to write the corresponding communication instructions, if necessary. Here, I will consider only the so-called "ether" communication model. All processors are supposed to be directly connected: the time to transmit a message is independent of the source and destination processors. The model is exact for bus networks (for instance, a collection of workstations interconnected by an Ethernet). In many recent designs, the architects have taken pains to build fair approximations of 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 a very high constant. As a consequence, the quality of the distribution may be assessed simply by estimating the volume of remote data accesses.

Automatic distribution is a difficult task, which 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 will be substantially the same as in [Fea91]:

- instructions are restricted to assigments and `DO` loops,
- 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.

Provided these constraints are satisfied, one may compute the Dataflow Graph of the source program.

An important distinction when discussing parallel programs is the one between statements and operations. A statement is a static entity, which may be identified with a syntactic unit of the program text. An operation is one particular execution of a statement in a particular context. Most often, a statement will be executed many times. Each execution will be considered as a distinct operation. In a static control program, an 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 instruction its iteration domain, which is a subset of $\mathbb{N}^d$, where d is the number of surrounding loops. The iteration domain of instruction R will be noted $\mathcal{D}_R$. If array subscripts depend only on loop counters, knowledge of an operation name allows one to compute the addresses of accessed memory cells, and hence to establish whether dependences exists.

For a static control program, it is possible to analyze the flow of data through the operations and the memory cells. The basic technique is presented in [Fea91]; a brief description follows. For each read access in the program, the set of all preceding write accesses to the same memory cell is characterized and its latest element is computed. The result is the source of the value obtained by the read access. A source is composed of an instruction name and an iteration vector. Both these elements may depend on the iteration vector of the read access. The method of [Fea91] yields *source functions* in the form of more or less complicated conditional expressions.

> The program of Fig. 1 computes the Cholesky decomposition of a symmetric matrix. This program has only one size parameter, n.
>
> Let us consider the reference to `s` in statement 6. There are four *a priori* sources for the value of `s`: statements 1, 2, 4 and 5. Statements 1 and 2 may be eliminated outright by observing that for each instance of 6 there is a preceding instance of 4 which will destroy any previous value. In the case of instruction 5, the reasoning is as follows. For $\langle 5, i', j', k' \rangle$ to be a source for `s` in $\langle 6, i, j \rangle$, one must satisfy the following constaints:
>
> - there is no index constraint, since `s` is a scalar;
> - the definition must precedes the use:
>
> $$i' < i \vee (i' = i \wedge j' < j) \vee (i' = i \wedge j' = j),$$
>
> - the loop counters must be within the loop bounds:
>
> $$1 \le i' \le n, i' + 1 \le j' \le n, 1 \le k' \le i' - 1.$$

```
        program choles

        integer i, j, k
        real x
        real a(100,100), p(100)

        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
        end
```

Figure 1: Cholesky Decomposition

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`

and the correct source is the latest $\langle 5, i', j', k' \rangle$ operation which satisfies these constraints. The obvious solution is:

$$i' = i, j' = j, k' = i - 1.$$

However, this is a legitimate iteration vector only if it belongs to the iteration domain:

$$i - 1 \geq 1, j \geq i + 1.$$

If $\langle 6, i, j \rangle$ is a proper operation, the second condition is always satisfied but the first one is not. We conclude that the source of `s` in $\langle 6, i, j \rangle$ is $\langle 5, i, j, i - 1 \rangle$, provided that $i \geq 2$. It is easy to see that in the case $i = 1$, the source is $\langle 4, i, j \rangle$.

This reasoning may be reproduced (with suitable modifications), for all right-hand side (*rhs*) references in any static control program. I have shown elsewhere [Fea91] that the process is completely mechanical, the basic tool being an algorithm for solving parametric linear programing problems in integers [Fea88]. The corresponding program has been implemented partly in Lisp and partly in C, and runs on various computers including many workstations and a personal computer.

The result of the analysis may be presented as the so-called *Dataflow Graph* or DFG for short. Table 1 gives the DFG of the Cholesky example[1]. The DFG has one node per instruction in the original program. There is an edge from instruction R (*the source*) to instruction S (*the sink*) for each rhs 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 in the sink instruction. In the Cholesky example, as we have seen, the governing predicate of the $\langle 5, i, j, i - 1 \rangle \rightarrow \langle 6, i, j \rangle$ edge is $i \geq 2$. The governing predicate of edge e is associated to a subset $\mathcal{P}_e$ of the iteration domain of the sink instruction $\mathcal{D}_{\delta(e)}$.

- The *sink-to-source transformation* which allows one to compute the iteration vector of the source instruction in terms of the iteration vector of the sink instruction. In all practical cases we have encountered so far, this transformation is affine, but there is a possibility of encountering quasi-affine transformations as the result of the source computation (see [Fea88] for a somewhat artificial example). 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 $\mathcal{D}_{\sigma(e)}$.

[1] Edges are numbered arbitrarily by the analysis software

The DFG is a synthetic representation of communication patterns in the source program. If the actual values of the size parameters are known, one may construct an expanded version, whose vertices are the operations of the program:

$$E = \bigcup_S \{\langle S, x\rangle \mid x \in \mathcal{D}_S\}.$$

e being a an edge of the DFG from R to S, with governing predicate $\mathcal{P}_e$ and sink-to-source transform h_e, the expanded DFG has an edge from $\langle R, h_e(x)\rangle$ to $\langle S, x\rangle$ for all $x \in \mathcal{P}_e$. In realistic cases, the extended DFG is much too big to be used effectively. Observe for instance that on a 1 Megaflop machine, a program which runs for one second has an expanded DFG with one million nodes! Nevertheless, the expanded DFG is a useful conceptual tool for the design of scheduling and placement algorithms.

Knowledge of the Dataflow Graph enables one to construct a Single Assignment equivalent of the source program. The recipe has been given in [Fea91]. Basically, one associates a new array $\mathtt{v}_S$ to each instruction S. In the left hand side of the instruction, this array is subscripted by all components of the iteration vector of S. In the right hand side, each reference to a variable is replaced by its source, as given by the dataflow graph.

Single Assignment conversion greatly simplifies the construction of distributed programs. As is well known, dependences may be classified into flow dependences, anti-dependences and output dependences. When a flow dependence exists between two operations which are executed by different processors, a data transfer operation must be executed. In the case of an anti-dependence, a value must be transfered and the processors must be synchronized. In the case of an output dependence one may either synchronize then invalidate the obsolete copies, or transmit the new value to the other processors. In Single Assignment programs, there exists neither anti-dependences nor output dependences. Since each value is produced only once, one may decide that a value resides on the originating processor which is then responsible for transmitting it "to whom it may concern" – this is the "owner compute rule".

Constructing distributed versions of Single Assignment programs is thus seen to be much simpler than tackling arbitrary programs. The drawback, obviously, is that Single Assignment programs needs much more memory than their conventional counterparts. In the Cholesky case, memory size which is $O(n^2)$ in the original version, grows up to $O(n^3)$ in the single assignment version. In my mind, single assignment programs are not meant to be executed as written. In many cases, when distribution is done, one may optimize memory usage; see [RWF91] for an outline of the technique. Another interesting property of single assignment programs is that their iteration domain is identical to their data space. Hence, if one follows the owner compute rule, data distribution and code distribution are one and the same process.

The best way of exhibiting the parallelism in a static control program is to construct a *schedule* from its Dataflow Graph. A schedule for instruction S is a function θ from the set of operations E to N. $\theta(S,x), x \in \mathcal{D}_S$ may be interpreted as the earliest date at which operation $\langle S, x\rangle$ may be executed on a computer with an unlimited number of processors. One may define more general schedules, whose codomains are other totally ordered sets – for instance, N^d under lexicographical ordering.

When executing a parallel program, one has to insure that when an operation start executing, all its arguments have been already computed. This may be written as the following *causality condition*:

$$\forall e \in DFG, x \in \mathcal{P}_e : \theta(\delta(e), x) > \theta(\sigma(e), h_e(x)). \tag{1}$$

Finding *causal schedules* has received much attention, both in the context of systolic array design and in the context of loop nest scheduling [Qui87, QR81, MQRS90, Fea92a, Fea92b, SQ92]. The *free schedule* is such that each operation is executed at the earliest possible time. It can be computed by a toplogical sort of the expanded DFG. For a more practical algorithm, one postulates the form of the solution – most of the time, an affine form in x – and translates (1) into a set of constraints for the unknown coefficients. One then solves these constraints according to some quality criterion, minimum latency for instance. The minimum latency schedule for program B is given by table 2.

Supposing a causal schedule θ has been found, one may construct the set of operations which are scheduled at time t:

$$\mathcal{F}(t) = \{u \mid u \in E, \theta(u) = t\},$$

the *front* at time t. From the causality condition, one sees that there is neither dependences nor data transfer between operations in the same front. Operations in a front may be arbitrarily distributed between processors. The corresponding parallel program may be described by the following sketch:

```
do t = 0, L
   parallel do
      F(t)
   end parallel do
   barrier
end do
```

L is the latency of the schedule. The `barrier` instruction insures that proper synchronization between fronts is observed. It compiles to a `NOP` on synchronous machines, like the CM-2 or the Maspar.

Instruction	Loop counters	Schedule	Domain
1	i	0	$\begin{pmatrix} n-i \\ i-1 \end{pmatrix} \geq 0$
2	i, k	$3k$	$\begin{pmatrix} n-i \\ i-1 \\ i-k-1 \\ k-1 \end{pmatrix} \geq 0$
3	i	$3i-2$	$\begin{pmatrix} n-i \\ i-1 \end{pmatrix} \geq 0$
4	i, j	0	$\begin{pmatrix} n-i \\ i-1 \\ n-j \\ j-i-1 \end{pmatrix} \geq 0$
5	i, j, k	$3k$	$\begin{pmatrix} n-i \\ i-1 \\ n-j \\ j-i-1 \\ i-k-1 \\ k-1 \end{pmatrix} \geq 0$
6	i, j	$3i-1$	$\begin{pmatrix} n-i \\ i-1 \\ n-j \\ j-i-1 \end{pmatrix} \geq 0$

Table 2: A schedule for program B

3 Data and code distribution

3.1 Introduction

In the present context, the most interesting property of fronts is that their operations may be distributed arbitrarily among the available processors. In the case of a shared memory machine, the only constraint is load balancing. If one distributes the operations in a front as fairly as possible among the available processors, and if the source program has any significant parallelism, one get an asymptotically efficient parallel program [Fea89]. On the other hand, for a distributed memory architecture, the main concern must be minimizing data communications. Now, if the owner compute rule is followed, 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 expanded DFG, a communication will be necessary *unless both operations are executed by the same processor*. Since fronts may be distributed arbitrarily, we should arrange the program in such a way that operations $\langle R, x\rangle$ and $\langle S, h_e(x)\rangle$, which do not belong to the same front, 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 $[0, P-1]$. One may say that placement functions are to space what a schedule is to time. A placement function being given, one may define the subset of each front which is allotted to processor $q \in [0, P-1]$:

$$\mathcal{F}(t, q) = \{u \mid u \in E, \theta(u) = t, \Pi(u) = q\}.$$

The distributed program will consist in an initialization phase, after which processor q will start executing the following code:

```
do t = 0, L
   exchange data
   do
      F(t, q)
   end do
   barrier
end do
```

The analogue of the causality condition is obtained by writing that two operations which exchange data are executed by the same processor. For each edge e of the DFG:

$$x \in \mathcal{P}_e \Rightarrow \Pi(\delta(e), x) = \Pi(\sigma(e), h_e(x)). \tag{2}$$

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:

```
program C

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

end
```

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 [Taw91]. One may solve approximately this optimization problem by classifying edges in two categories:

- *cut edges*, for which equation 2 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 section 4.

The problem may be formally solved along the following lines. Let us consider the undirected graph which is obtained from the expanded DFG by forgetting the orientation of edges. This graph will be called the *communication graph* of the program. Now it is clear from (2) that any placement function is constant on the *connected components* of this graph. We may define a canonical placement function, in which each connected component is allocated to a different processor. Any other placement may be obtained by composing the canonical solution with an arbitrary function.

In the case of program C, it is a simple matter to see that the communication graph is connected. Hence the conclusion that the canonical solution is constant. It seems that this situation is the normal one for real programs: after all, if the data in a program may be split into several totally unrelated sets, it is good sofware engineering to split the program in several phases. Hence, the main problem is to decide which edges are going to be cut and which ones are going to be kept as residual communications.

Constructing the connected components when the set of cut edges is known is still a difficult process. Furthermore, placement functions are used to construct the sub-fronts $\mathcal{F}(t, q)$, and that is easy only if they are affine forms in the loop counters. 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 fully dimensional. It contains a ball of diameter d, and the range of any linear function $\Pi(S, x) = a.x + b$ in $\mathcal{D}_S$ will be at least $nd|a|$. 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:

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

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 [CMF89], or of the template in HPF [Lov92], 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 4.

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:

$$d_e(x) = \pi(\delta(e), x) - \pi(\sigma(e), h_e(x)). \tag{3}$$

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:

$$\pi(S, x) = a_S.x + b_S. \tag{4}$$

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 $\mathcal{P}_e$ is fully 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 rewrite $d_e(x)$ in term of the new variables, whose coefficient must also be null. In both cases, the edge cutting condition translate 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. The matrix C_e is obtained by straighforward algebraic manipulations from h_e. The union of all such system will be written:

$$Ca = 0.$$

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

$$\pi_i(x) = b_i + \sum_k a_{ik} x_k,$$

for the placement function of statement i. 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 $k \leq 1$ together with the constraint $k \geq 1$ in the domain imply $k = 1$. As a consequence, the placement equation reduce 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 which express a subset of a in term of the remaining variables. Initially, σ is the empty substitution.

2. Extract the next line of C and apply σ to it. Write the resulting equation in the form:

$$x = f,$$

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

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 have the same property. The uneliminated variables – those which do not occur in the left hand side of σ – may take arbitrary values. One easily prove 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 set to zero by σ.

> Let there be n uneliminated variables, and let λ be a positive number. Select an arbitrary rhs, select an arbitrary variable x in it, and give it the value λ^n. Select now another rhs in which x does not occur, select another variable and give it the value λ^{n-1}. Continue in this way until all rhs contain at least one valued variable. Give the

value 1 to all remaining variables. It should be clear that, for λ large enough, all rhs are non zero. ∎

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 associted equations are:

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

The new 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, part 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 instruction 6 are set to 0. As a consequence, we ignore the foregoing equation and try again with the next one.

The algorithm continue 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:

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

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 stems from an asymptotic analysis of the worload 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 been sent only once even if it is used by many PE.

The emitter set is:

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

One first eliminate 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 construct 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 edge 108 in table 1. The emitter set is:

$$E_{108} = \{i', j', k' \mid i' = i, j' = j, k' = i' - 1, 1 \leq i \leq n, i + 1 \leq j \leq n\}.$$

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 processor. Consider for instance the case of instruction 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 instruction 5, which has a two dimensional front, but what are we to do for instruction 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 instruction – 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 instruction 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 instruction 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 depends 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 instructions whose fronts do not have the requested dimension. For instance, one cannot ask for a two dimensional placement function for instruction 6 in Cholesky. In that case, one will obtain a placement function whose two components are not linearly independent.

> In the case of Cholesky, the maximum possible dimension is two. Algorithm E with the new triviality test gives a prototype with two independent parameters:

$$\pi(1, i) \;=\; \alpha i,$$

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

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

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

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 belongs to different virtual processors which are folded to the same PE. Let us consider the distance $d_e(x)$ of equation (3). 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 Conclusions and Future Work

Our proposal has two characteristics:

- Each instruction use the same "geometry", whatever the dimension of its iteration space. This means that for lower dimensional instructions, 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 on these two problems. The selection of rearrangement points will be specially critical.

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 instructions should be taken into account at this stage. One should also experiment with other edge ranking schemes, giving for instance a much lower weight to fixed distance communication like edges 101 and 106.

Lastly, the placement functions and the schedule will constitute a space-time basis for the construction of the parallel code. As several author have noticed, the code generation phase is greatly simplified if this basis is unimodular [BL92]. Whether the above technique can be adapted to take the unimodularity constraint into account remains to be seen.

References

[BL92] Michael Barnett and Christian Lengauer. Loop parallelization and unimodularity. In Michel Cosnard, Maurice Nivat, and Yves Robert, editors, *Algorithmique Parallèle*, Paris, May 1992. Masson.

[CMF89] Thinking Machine Corp., Cambridge, MA. *CM Fortran Reference Manual, Version 5.2*, 1989.

[Fea88] Paul Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22:243–268, September 1988.

[Fea89] Paul Feautrier. Asymptotically efficent algorithms for parallel architectures. In M. Cosnard and C. Girault, editors, *Decentralized System*, pages 273–284. IFIP WG 10.3, North-Holland, December 1989.

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

[Fea92a] Paul Feautrier. Some efficient solution to the affine scheduling problem, part i, one-dimensional time. Technical Report 92.28, IBP/MASI, May 1992.

[Fea92b] Paul Feautrier. Some efficient solution to the affine scheduling problem, part ii, multidimensional time. Technical Report 92.78, IBP/MASI, October 1992.

[HKK+91] S. Hirandani, 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.

[Lov92] David Loveman. High performance fortran. In Hans P. Zima, editor, *Proceedings of the Third Workshop on Compilers for Parallel Computers*, Vienna, July 1992.

[MQRS90] Christophe Mauras, Patrice Quinton, Sanjay Rajopadhye, and Yannick Saouter. Scheduling affine parameterized recurrences by means of variable dependent timing functions. Technical Report 1204, INRIA, April 1990.

[QR81] Patrice Quinton and Yves Robert. *Algorithmes et Architectures Systoliques*. Masson, Paris, 1981.

[Qui87] Patrice Quinton. The systematic design of systolic arrays. In F. Fogelman, Y. Robert, and M. Tschuente, editors, *Automata networks in Computer Science*, pages 229–260. Manchester University Press, December 1987.

[RWF91] Mourad Raji-Werth and P. Feautrier. On parallel program generation for massively parallel architectures. In M. Durand and F. El Dabaghi, editors, *High Performance Computing II*. North-Holland, October 1991.

[SQ92] Yannick Saouter and Patrice Quinton. Computability of recurrence equations. *TCS*, 1992. To appear.

[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.

[ZBC+92] Hans Zima, Peter Brezany, Barbara Chapman, Piyush Mehrotra, and A. Schawald. Vienna foetran – a language specification. Technical Report 21, ICASE, 1992.

[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.

```
@TECHREPORT{Feau:92c,
AUTHOR = "Paul Feautrier",
TITLE = "Toward Automatic Distribution",
INSTITUTION = "IBP/MASI", NUMBER = "92.95",
YEAR = 1992, MONTH = Dec}
```