Automatic Data Distribution for Massively Parallel Computers

Alexis PLATONOFF

Centre de Recherche en Informatique, École Nationale Supérieure des Mines de Paris,
35, rue Saint-Honoré, 77305 FONTAINEBLEAU Cedex, FRANCE.
platonof@cri.ensmp.fr

Abstract

When programming distributed memory machines, users are compelled to deal with the data distribution. However, this is a very complex problem. An automatic method has been proposed to determine automatically a placement function that gives the virtual processor executing each operation of the source program. But it does not take into account the particular types of data communication available on these machines. In order to fill this gap, a new extension is proposed. It is based on the detection before hand of the types of communication that can be replaced by their optimized counterparts. It has been implemented in the parallelizer PIPS developed at École des mines de Paris. Experiments have been done on a CM-5 and a CRAY-T3D.

Introduction

A new type of computer architecture, the so called *massively parallel architecture*, has appeared in the last few years. In these computers the memory is physically distributed, so data communications between the local memories of the processors have to be handled with care. Moreover, on these machines, well optimized types of communication have been implemented. They are worth using because bad performances are often due to the high number of expensive communications. However, the determination of the best distribution and the use of these "fast" communications were until recently left to the user. Many extensions to Fortran introduce some directives to do it ([Pla93]). This problem being very complex [Mac87, LC90, DR94], many studies try to solve it automatically.

Some projects suggest methods that align arrays [GB92, KLS90], but with no program transformations. Anderson & Lam introduce an algorithm that computes tasks and data decomposition [AL93] and try to find the coarsest grain parallelism with unimodular transformations. Darte & Robert provide a model based on a communication graph that defines the data distribution with an affine function and they try to optimize some communications (broadcasts, message vectorization) [DR93, DR94].

With a similar idea Feautrier introduces the computation of a *placement function* which completely characterizes the data and tasks distribution of a program [Fea93]. The placement gives the identity of the virtual processor that executes each instruction. The goal of the placement function is to minimize the volume of communication.

Compared to the above methods, this approach works on any loop nest where the array accesses are affine functions and it is part of a global parallelization method that computes an affine schedule and transforms the program according to it and the placement function.

But it does not take into account the type (and so the cost) of the communications. For that purpose, we provide an extension of the method which is based on a special treatment for the potential communications that can be optimized.

In the first section, we describe the technique introduced by Feautrier. In Section 2, we present three kinds of "fast" data movements and methods to detect them. The third section gives the conditions under which these special communications can be taken into account to compute the placement function. Section 4 provides the new algorithm that computes the placement function. Finally, the experiments we have done on a CM-5 and a CRAY-T3D are compared to those from the initial method in Section 5.

1 Framework

Our study is limited to a certain class of programs called "static control programs" [Fea91], based on the work of Feautrier's team of the PRiSM laboratory (University of Versailles, France).

In such programs the authorized control structures are only the DO *loop* and the IF *test*; the instructions are either *assignment* or *input/output* statements; the loop bound expressions and the array subscript expressions must be *integer linear expressions* function of surrounding loop indices and *structure parameters*[1].

Figure 1 gives an example of a static control program (program `FMM`, for *Floating point Multiplication of Matrices*). Instructions `s1` and `s2` have been added in order to explicitly show the initialization of both arrays.

1.1 Data flow graph

The *data flow graph* is computed from a static control program (noted DFG, [Fea91]). The DFG is in fact a kind of dependence graph in which only true dependences appear.

To each node of the DFG are associated an *instruction* and an *execution domain*. The execution domain is a system of constraints specifying the variation domain of the surrounding loop indices of the instruction.

Nodes are connected by oriented edges which represent true data dependences. To each edge of the DFG are associated the *reference* of the dependence, a *transformation* function in which each index of the source's iteration domain is expressed as a linear function of the indices of the sink's iteration domain, and a *governing predicate* that specifies the sub-space of the sink's iteration domain on which the edge exists (it is a system of constraints upon the indices of the sink's iteration domain).

Let us come back to the program `FMM`. The DFG of this program has four nodes, one for each instruction, and four edges (see Figure 2). Table 1 contains a detailed description of each edge. In this table, the transformation, given in the *source* column,

[1]A structure parameter is an integer variable defined only once in the program, typically a constant that defines the array size.

```
            program FMM
            integer i, j, k, n
            real a(n,n), b(n,n), c(n,n), a'(n,n), b'(n,n)
            do i=1,n
              do j=1,n
s1               a(i,j) = a'(i,j)
s2               b(i,j) = b'(i,j)
              end do
            end do
            do 4 i=1,n
              do 4 j=1,n
s3               c(i,j) = 0.
                do 4 k=1,n
s4                 c(i,j) = c(i,j) + a(i,k) * b(k,j)
                end do
              end do
            end do
            end
```

Figure 1: Program FMM

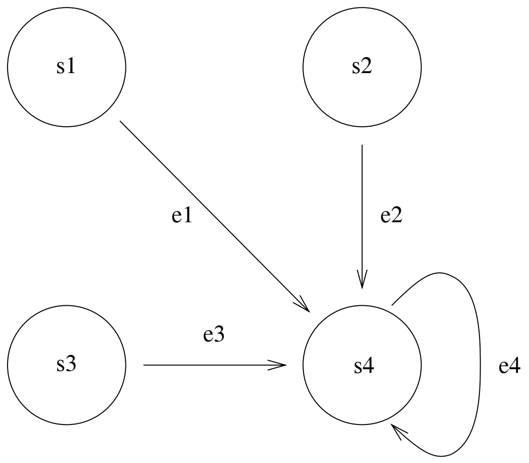

Figure 2: DFG of program FMM

is function of the indices in the *destination* column. The *predicate* column is empty when the edge exists for all the values in the execution domain of the destination.

Edge	Source	Destination	Reference	Predicate
e1	$\langle s1, i, k\rangle$	$\langle s4, i, j, k\rangle$	a(i,k)	
e2	$\langle s2, k, j\rangle$	$\langle s4, i, j, k\rangle$	b(k,j)	
e3	$\langle s3, i, j\rangle$	$\langle s4, i, j, k\rangle$	c(i,j)	$1 - k \geq 0$
e4	$\langle s4, i, j, k-1\rangle$	$\langle s4, i, j, k\rangle$	c(i,j)	$k - 2 \geq 0$

Table 1: Edges of the DFG of program `FMM`

1.2 Scheduling function

The scheduling function is computed from the DFG [Fea92a]. The schedule provides for each operation of a program the logical date at which it must be executed. For an instruction s, it is expressed as a linear function θ_s of the loop indices and the structure parameters.

For a given time t, the scheduling function defines a set of operations which are executed between time t and time $t+1$:

$$F(t) = \{u \mid \forall s, \theta_s(u) = t\}$$

This set is called a *front*. All the operations of the same front are independent so they can be executed simultaneously. On the contrary, the execution of two successive fronts must be sequential.

Even with the previous restrictions, it is not possible to always have a unidimensional linear schedule. Typically, this happens when there are two or more sequential loops in the same loop nest. In that case, a multidimensional linear schedule is computed [Fea92b].

The schedule of program `FMM` is unidimensional:

$$\begin{array}{ll} \theta_{s1}(i,j) & = 0 \\ \theta_{s2}(i,j) & = 0 \\ \theta_{s3}(i,j) & = 0 \\ \theta_{s4}(i,j,k) & = k \end{array}$$

This means that the first three instructions are fully parallel and the last instruction has only two parallel loops (the innermost one on `k` is sequential).

1.3 Placement function

The placement function associates to each instruction a multidimensional affine function of the loop indices and the structure parameters [Fea93]. It specifies explicitly the placement of the instruction on a virtual processor grid, *i.e.*, gives the identity of the virtual processor that executes each operation of the instruction.

For a given instruction s, each dimension of its placement function is represented with a prototype π_s which is an affine function of the loop indices ($x_s = \{x_s^i\}$) and the structure parameters ($c = \{c^i\}$):

$$\pi_s(x_s) = \sum_{i=1}^{|x_s|}(\lambda_s^i.x_s^i) + \sum_{i=1}^{|c|}(\mu_s^i.c^i) + \nu_s$$

The goal is to find the values to all these unknown coefficients: $\lambda_s^i, \mu_s^i, \nu_s$.

Each dimension of the placement function defines a distribution direction for the instruction and all these directions constitute the distribution space of the instruction. The number of dimensions to compute is arbitrary with a maximum equal to the dimension of iteration space minus the dimension of the scheduling function.

Initially, the goal of the algorithm that computes the placement function is to reduce the number of communications. For an edge e of the DFG (transformation h_e, see above Section 1.1), there is a potential communication between its source $\sigma(e)$ and its destination $\delta(e)$, which can be represented by a distance:

$$d_e(x) = \pi_{\delta(e)}(x) - \pi_{\sigma(e)}(h_e(x))$$

If such an equation is equal to zero (the edge is "cut") then the source and the destination will be mapped onto the same processor, and there will be no communication.

The principle of the method is to nullify as much distance as possible. To each edge is associated what is called a *cutting condition* represented by a system of equalities. The distance of an edge is nullified if its cutting condition is satisfied, *i.e.* its equalities are satisfied. These equalities corresponds to the nullification of the factors of the variables (loop indices and structure parameters) appearing in the distance. In most cases, satisfying the cutting conditions of all the edges will lead to the trivial solution: some null dimensions, *i.e.* everything is on the same processor. To avoid this, some edges must not be cut. To choose them, a greedy algorithm has been proposed that treats the edges by decreasing importance [Fea93]. The importance of an edge is represented by its weight which is equal to the volume of data which have to be sent if the edge is not cut.

For the computation of the placement function of program `FMM`, we create a prototype for each instruction:

$$\begin{aligned}
\pi_{s1}(i,j,n) &= \lambda_2.i + \lambda_3.j + \lambda_4.n + \lambda_1 \\
\pi_{s2}(i,j,n) &= \lambda_6.i + \lambda_7.j + \lambda_8.n + \lambda_5 \\
\pi_{s3}(i,j,n) &= \lambda_{10}.i + \lambda_{11}.j + \lambda_{12}.n + \lambda_9 \\
\pi_{s4}(i,j,k,n) &= \lambda_{14}.i + \lambda_{15}.j + \lambda_{16}.k + \lambda_{17}.n + \lambda_{13}
\end{aligned}$$

From Table 1, we have four distances:

$$\begin{aligned}
d_{e1}(i,j,k) = &\ (\lambda_{14} - \lambda_3).i + \lambda_{15}.j + (\lambda_{16} - \lambda_3).k + (\lambda_{17} - \lambda_4).n + \lambda_{13} - \lambda_1 \\
d_{e2}(i,j,k) = &\ \lambda_{14}.i + (\lambda_{15} - \lambda_7).j + (\lambda_{16} - \lambda_6).k + (\lambda_{17} - \lambda_8).n + \lambda_{13} - \lambda_5 \\
d_{e3}(i,j,k) = &\ (\lambda_{14} - \lambda_{10}).i + (\lambda_{15} - \lambda_{11}).j + \lambda_{16}.k + (\lambda_{17} - \lambda_{12}).n + \lambda_{13} - \lambda_9 \\
d_{e4}(i,j,k) = &\ \lambda_{16}
\end{aligned}$$

The instruction with a three dimensional execution domain (`s4`) has a non null unidimensional schedule. And, all the others (`s1, s2, s3`) have a bidimensional execution domain and a null schedule. Hence, the maximum number of dimensions we are able to compute is two.

1.4 Code generation

The final phase builds the parallel program using all the results of the preceding phases. At a given time step, the parallel program has to execute the corresponding front (see above), synchronize and pass to the next time step. This induces the following general architecture of the parallel program [RWF91]:

```
do t = 1, number of fronts
  execute simultaneously all operation of F(t)
  synchronize
end do
```

The code generation is based upon three transformations. The first is a *total expansion* which transforms the initial program into single assignment form [Fea88]. The second is a *loop reordering* which rearranges the iteration domain of the initial loops according to the scheduling and placement functions (the first gives the sequential loops, the second the parallel ones). The reordering is equivalent to scanning polyhedra with do loops [AI91]. The third is a *reindexing* which substitutes all the array access functions with new ones computed according to the new loops. A general method for generating parallel code from the results of the preceding phases has been proposed by Collard [CF93b].

2 Data movements

Li & Cheng ([LC91]) distinguish five types of data movement that can be executed on distributed-memory machines: permutation, translation (or uniform communication) which is a special case of permutation, aggregate communication (broadcast and reduction), general communication and asynchronous communication. However, most of the distributed memory massively parallel machines (e.g. CM-5 and CRAY-T3D) have efficiently implemented the broadcast, the reduction and the translation, and offer several high-level primitives that use them.

To clarify this, we have compared the cost of the reduction, the broadcast, the translation and the general communication on a CM-5 and for the same amount of data. Table 2 shows the execution time of each kind of communication relatively to each other, supposing that the reduction is done in one unit of time. We easily deduce from this table that the use of the first three data movements is extremely advantageous compare to the last one, which should be avoided as much as possible.

Reduction	Broadcast	Translation	General communication
1	1.5	2.5	90

Table 2: Comparison of data movements

Thus, if we could detect the instructions that may produce this kind of data movements, we could take them into account to compute the placement function, and generate the code using these primitives for a more efficient execution.

2.1 Broadcast

A broadcast corresponds to the fact that the same memory cell is read by distinct instances (or operations) of an instruction. If these instances are executed on distinct processors then it is necessary to send this value to all of them, which may induce a great cost if an optimized broadcast communication is not used. Figure 3 gives the communication schemes of the broadcast.

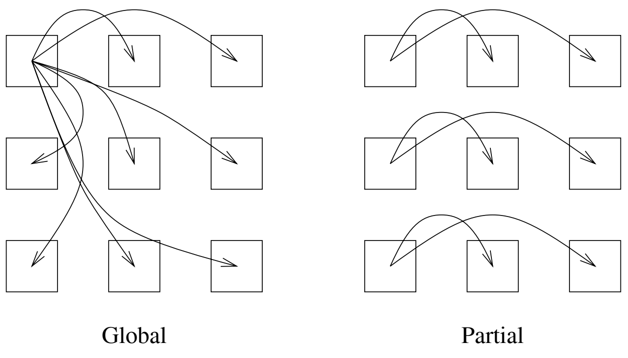

Figure 3: Broadcast communication scheme

An edge of the DFG is a broadcast if, the values of the indices of the source being fixed, there exist several possible values for the indices of the destination. The transformation (T) associated to each edge of the DFG (see Section 1.1) expresses the values of the indices of the source (i') as a function of the indices of the destination (i):

$$T * i = i'$$

Thus, for a given value of i', we are looking for all the possible values of i. This is equivalent to looking for the solutions of a system of integer linear equations. The solutions of such a system are in the space defined by the kernel of T, $Ker(T)$. The basis of $Ker(T)$ gives the broadcast directions.

Briefly, in order to detect if an edge e is a broadcast, you only have to consider the transformation matrix (T_e), compute its kernel ($Ker(T_e)$) and deduce from its basis the broadcast space.

Let us come back to program `FMM` and take edge `e1` (see Section 1.1). The edge is from `s1` to `s4` and the transformation is as follows:

$$T_{e1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

A basis of $Ker(T_{e1})$ can easily be found: $\{(0,1,0)\}$. This means that the broadcast is unidimensional and done along `j`.

2.2 Reduction

A reduction is the reverse operation of a broadcast; it corresponds to the computation of a value using the values computed by different instances of instructions in an associative operation.

Formally, it is an order one recurrence with one equation. A few methods for the detection of reductions already exist, as the ones proposed by Jouvelot & Dehbonei [JD89] or by Redon [RF93]. The latter uses the informations contained in the Dfg and detects general recurrences of many equations and of all order.

Practically, in order to take into account the reductions we would have to modified the schedule of the program because otherwise the reduction is sequential. The problem of computing the scheduling function with the reductions have been solved by Redon [RF94] and its exploitation on the computation of placement function is currently being studied by Barreteau [BF95]. So the reduction is not treated in our present method.

2.3 Translation

A translation is a data movement in which all the processors send a data (or a set of data) on the same grid direction and at the same distance. Figure 4 gives the communication scheme of the translation.

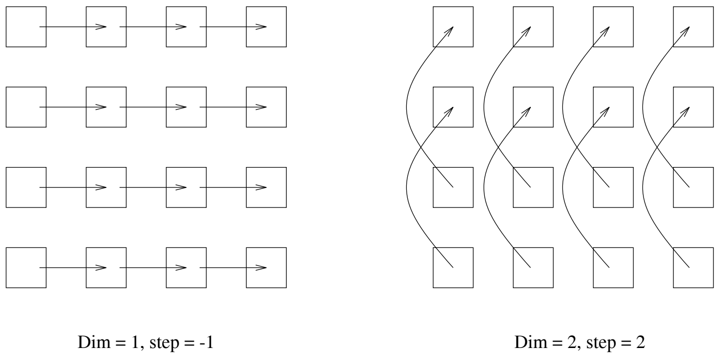

Figure 4: Translation communication scheme

Let us come back to the edges of the Dfg. If the distance of such an edge is constant (but not null) it means that it represents a data movement in which the source and destination are always at the same distance. Thus, it is a translation.

An edge distance is expressed as a linear function of the loop indices and the structure parameters. So a constant distance means that its expression is independent of the loop indices.

3 Placement conditions

The great majority of programs to be executed on massively parallel machines will have to perform communications. So, it is better if these compulsory data movements can be optimized. In this section, we study the conditions to satisfy during the computation of the placement function in order to be able to use optimized communications such as broadcasts and translations.

3.1 Broadcast conditions

A broadcast communication distributes some data on several processors, hence the destination array must be mapped according to the broadcast directions. So, for a given edge detected as a broadcast, the conditions are put on the prototype of the destination of the edge.

It is important to notice that a broadcast is always done along one or more dimensions of the virtual processor grid. Then, we distinguish the *global* broadcast in which the data movement is done along all the dimensions of the grid, and the *partial* broadcast in which only some dimensions are concerned.

Let us take as an example a contrived instruction with a three dimensional iteration space (I, J, K) and for which we have determined a two dimensional distribution space (P, Q) such as:

$$\begin{cases} P = I + J \\ Q = I - J \end{cases}$$

Let us now suppose that the instruction is the destination of an edge on which we have detected a broadcast. Figure 5 shows the broadcast communication that we will effectively be able to generate for four different broadcast spaces. The first broadcast can not be generated because it is partial and not parallel to one of the axis. General communications must be used instead. The second one is global because all the distribution space is included in the projection of the broadcast space. In the third one, the projection of the broadcast direction is parallel to one of the distribution space axis, so it is a partial broadcast which can effectively be done. Finally, the fourth one can only be executed on one dimension because the other dimension is collapsed (all the destinations of the broadcast on the collapsed dimension are mapped to the same processor).

Global broadcast In general, an edge represents a global broadcast when the distribution space of its destination is included in the projection of its broadcast space. The condition is expressed by a system of equalities on the unknown coefficients of the prototype of the destination of the edge. If all these equalities are satisfied we are sure to be able to generate a global broadcast.

Let us examine an example of global broadcast condition on program `FMM`. We have seen that edge `e1` has a broadcast space defined by $\{(0, 1, 0)\}$ (see Section 2.1). The destination of the edge is `s4` that has a three dimensional iteration space (see Section 1.3):

$$\pi_{s4}(i, j, k, n) = \lambda_{14}.i + \lambda_{15}.j + \lambda_{16}.k + \lambda_{17}.n + \lambda_{13}$$

In this case, we easily find the condition for which the distribution space is included in the projection of the broadcast space:

$$\begin{cases} \lambda_{14} = 0 \\ \lambda_{16} = 0 \end{cases}$$

Partial broadcast If a broadcast can not be global, we can make it partial. An edge represents a partial broadcast when the space defined by some of the directions of the distribution space of its destination is included in the space defined by the projection

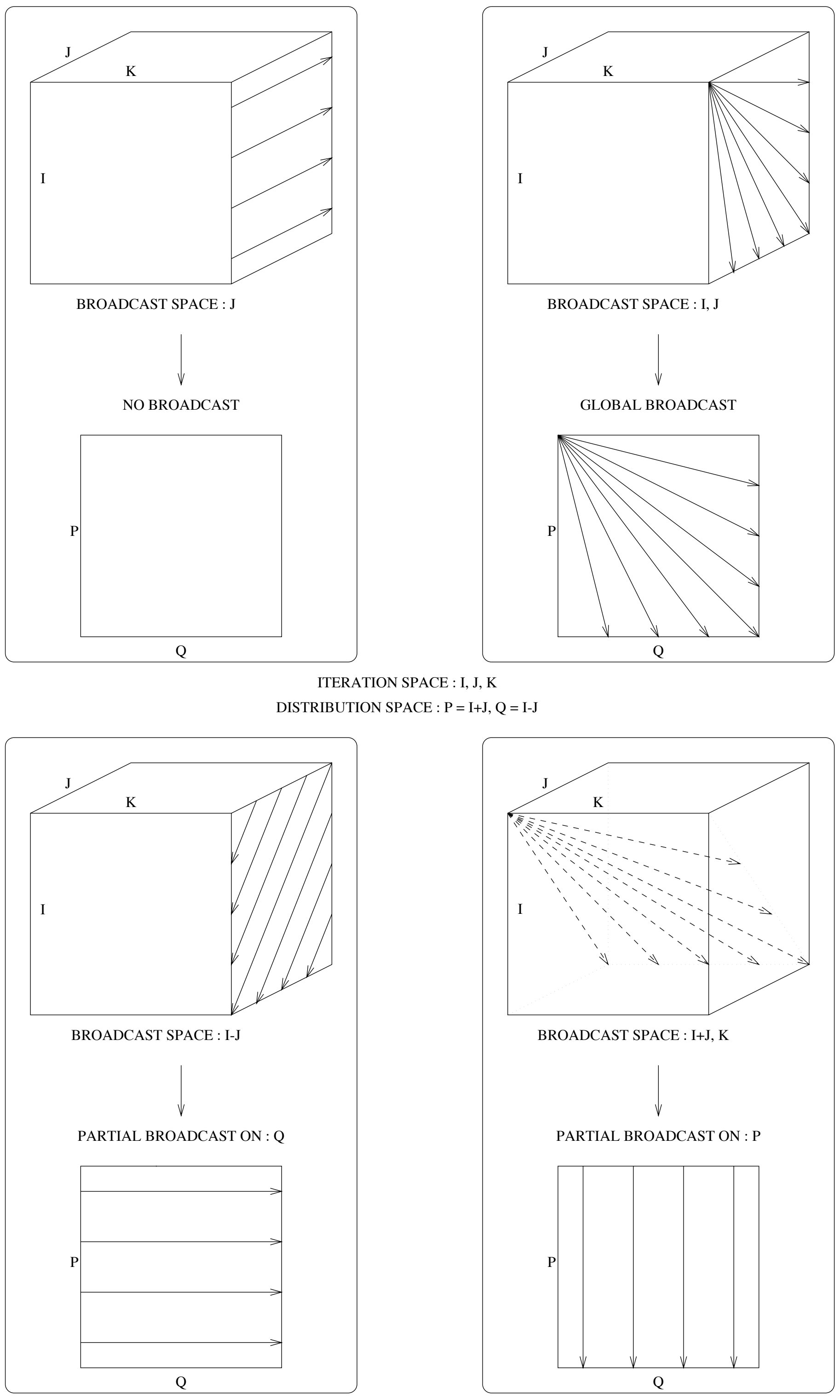

Figure 5: Global and partial broadcasts

of some of the broadcast directions. The idea is to keep for each instruction all the non-global broadcasts and build a family of broadcasts directions. Then, our goal is to try to have a projection of these broadcast directions on the distribution directions. The condition for this is to associate a new unknown coefficient to each broadcast direction and substitute them in the placement prototype. These new coefficients are such that if one of them is set to 1 and all the others to 0 then the distribution direction is equal to the projection of the corresponding broadcast direction.

Suppose that, for program `FMM`, the global broadcast condition of edge `e1` has not been satisfied. We associate a new unknown coefficient (μ_1) to the broadcast direction and introduce it in the prototype of `s4`. In our case, the substitution is simple:

$$\mu_1 = \lambda_{15} \Leftrightarrow \pi_{s4}(i,j,k,n) = \lambda_{14}.i + \mu_1.j + \lambda_{16}.k + \lambda_{17}.n + \lambda_{13}$$

3.2 Translation conditions

We said above that an edge corresponds to a translation if its distance is constant, *i.e.* is independent of the subscripts of the surrounding loops. The goal of the direct method of computing the placement function is to nullified as much edge distances as possible. The distance is constant if the factors of the loop indices are nullified, so these are the first equalities we try to satisfied. In this case it will be easier to make a constant distance than a null distance. Moreover, in certain cases, it will result in two constant distances instead of a null distance and a general (not constant) distance. In that case, it is a better result because one general communication is much more expensive than two translations.

The DFG of program `FMM` has four edges for which we have computed distances (see Section 1.3). For instance, from distance d_{e1} we deduce the cutting condition for edge `e1`:

$$d_{e1}(i,j,k) = (\lambda_{14} - \lambda_3).i + \lambda_{15}.j + (\lambda_{16} - \lambda_3).k + (\lambda_{17} - \lambda_4).n + \lambda_{13} - \lambda_1$$

$$\Rightarrow \begin{cases} \lambda_{14} - \lambda_3 = 0 \\ \lambda_{15} = 0 \\ \lambda_{16} - \lambda_3 = 0 \\ \lambda_{17} - \lambda_4 = 0 \\ \lambda_{13} - \lambda_1 = 0 \end{cases}$$

The first three equations result from the nullification of the factors of the loop indices. Hence, we only have to consider these equations so as to make this distance constant.

4 New algorithm

In [Fea93], Feautrier presents an algorithm for the computation of the placement function. We keep its principle of it but for the edges treatment. The goal of our extension is to take into account both our detection of broadcast that can be attached to each edge of the DFG and the translation conditions [Pla95]. Thus, the order in which we consider the edges depends on the type of communication they induce. We want to minimize the number of communications and optimize those to be done. Hence, the edges with optimized communication have priority and are treated first.

In the remainder of this section, we give the new algorithm for computing the placement function:

1. Compute the weight of the edges.
2. Associate a placement prototype to each instruction.
3. Detect the broadcasts; this finds the broadcast space of each edge.
4. Satisfy the conditions of global broadcast on each edge. The edges for which at least one equality is not satisfied (it means the broadcast can not be global) are kept for the cutting conditions treatment.
5. Satisfy the conditions of partial broadcast on each instruction that is destination of edges with non-global broadcast.
6. Satisfy the cutting conditions on each edge except those with global broadcast (translation condition: try to have constant distances first).
7. valuate the remaining unknowns in order to build each dimension of the placement function.

5 Experiments

All this method and the code generation module have been implemented in the parallelizer Pips of the École des mines de Paris [IJT91]. And we have done some experiments on two massively parallel computers the Cm-5 [TMC92] and the Cray-t3d [Mel94].

5.1 Placement efficiency

In order to measure the efficiency of our new algorithm, we have compared it to the initial method for eight programs (19 to 72 lines of Fortran) from different domains of application.

We have classified the edges in four categories depending on the type of communication it will generate at the execution. The type of communication is given by the value of the distance after the computation of the placement function: no communication if the distance is null (noted `ND`); a translation if the distance is constant (noted `CD`); a broadcast if the distance is not constant and the edge has been detected as such (noted `BD`); a general communication otherwise (noted `GD`).

We have summarized our results in Table 3 which gives the mean ratio of each type of edges for these eight programs. It is important to notice that the placement function computed with the initial method produces no edges of type `BD` because it does not detect them. This table shows that our new method produces a little more null distance edges (`ND`), *i.e.* there are less communications. More significantly, the number of edges of type `GD` (general communications) have greatly decreased. This is because several general communications have been replaced by broadcasts.

From this sample of eight programs, we see that our new algorithm for the computation of the placement function allows, in a great majority of cases, a good communication optimization, and thus a decrease of the global communication cost.

Placement with optimized communications				Placement : initial method			
ND	CD	BD	GD	ND	CD	BD	GD
67%	8%	15%	10%	60%	12%	0%	28%

Table 3: Mean ratio of the type of edges

5.2 Execution efficiency

Experiments were done for a program of multiplication of squared matrices. We have compared the execution time of three versions of the parallel program, the first one called `FMM1` is the parallelized version using the initial method of placement, the second one called `FMM2` is the parallelized version using the new method of placement, and the last one called `matmul` uses the intrinsic function `MATMUL`. We have chosen to give the speed-up of these three versions, computed with the execution time of a sequential version on one processor.

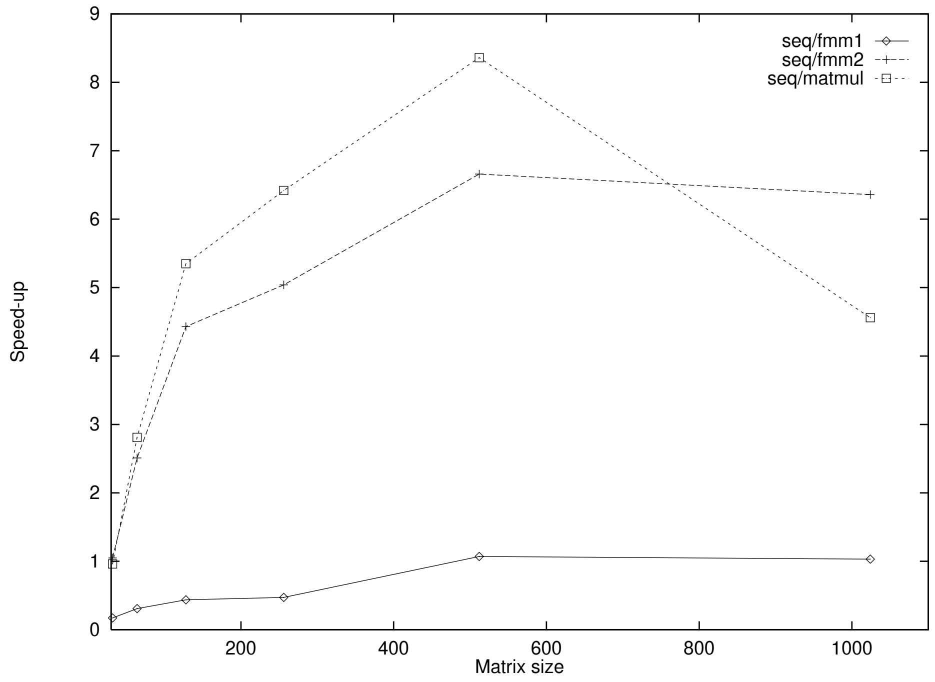

Figure 6: Speed-up on CM-5

Figure 6 shows the speed-up obtained on a CM-5 with 32 nodes and no vector units[2]. The speed-up is a function of the array size. We see that our new method have better performances than the initial one and is not that far from the `MATMUL`. This proves

[2]Without vector units we were able to compared our parallel executions with a sequential one on the front-end.

that the optimization of communications have greatly influenced the performances. However, the speed-up remains low even for the should-be-optimized `MATMUL` since, in the best case, the efficiency is 20% for `FMM2`.

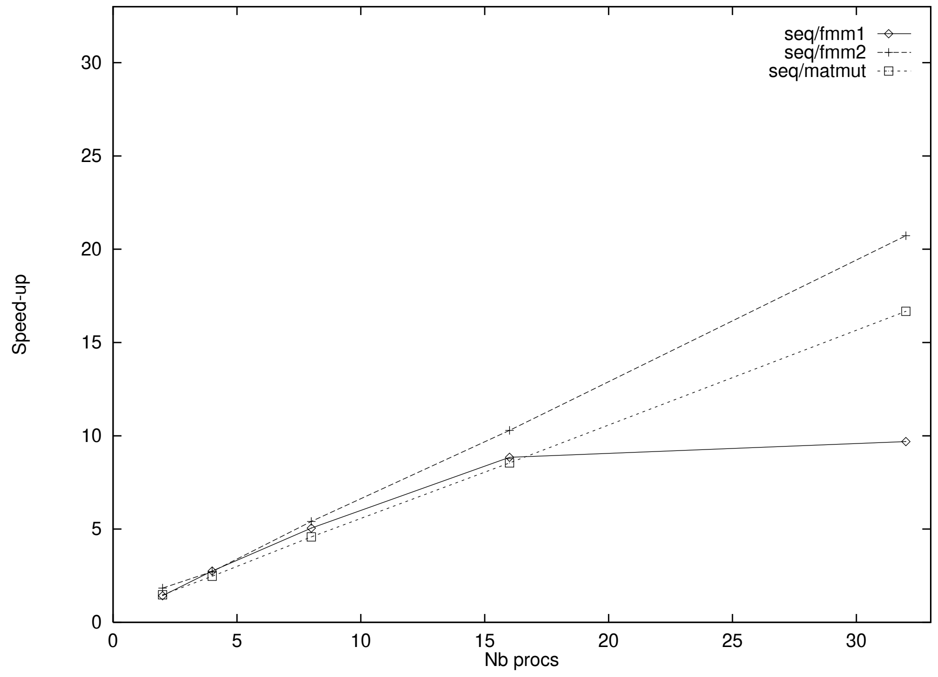

Figure 7: Speed-up on Cray-t3d

Figure 7 shows the speed-up obtained on a Cray-t3d with a 32 nodes partition. Here, the speed-up is a function of the number of processors and the array size is fixed to 256×256. While the number of processors is not very high (*i.e.* less than 16) the communication cost is not significant. But we see that with 32 processors, the expensive communications induced by `FMM1` induce a degradation of the performances compared with those of `FMM2`. The overall speed-up is good since the efficiency is about 70% for `FMM2`.

6 Conclusion and future work

In this paper, we present an extension of an affine placement method that deals with the well optimized type of communication (broadcasts, translations) implemented on massively parallel computers. We show how to detect the broadcasts and we introduce the conditions under which the broadcasts and the translations are taken into account to compute the placement function. We then give a general algorithm including these conditions.

This work has been implemented in C language and is integrated into the Pips project (Interprocedural Parallelizer of Scientific Programs). We prove the efficiency of the new method by comparing its results to those given by the initial method, firstly on the placement itself and secondly at the execution on two massively parallel machines: the CM-5 and the CRAY-T3D.

We assume that the method could be even more enhanced with the integration of the reductions in the computation of the placement function. Moreover, it could be interesting to test the method on other architectures, such as the SP2 or the PARAGON, which only requires a very small addition to our parallel code generator.

7 Acknowledgment

All this work was done at the Centre d'Étude de Limeil-Valenton of the Commissariat à l'Énergie Atomique (CEA, France) and was part of a research project done in collaboration with the École des mines de Paris and the University of Versailles (France).

Our experiments were done on the CM-5 of the Site Expérimental en Hyperparallélisme (SEH) of the Établissement Central Technique de l'Armement (ETCA) at Arcueil (France) and on the CRAY-T3D of the Centre d'Étude de Grenoble (CEA, France).

References

[AI91] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *PPOPP'91*, 1991.

[AL93] J.M. Anderson and M.S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI)*, Albuquerque, New Mexico, June 1993.

[BF95] M. Barreteau and P. Feautrier. Automatic mapping of scans and reductions. To appear in HPCS'95, February 1995.

[CF93a] B.M. Chapman and T. Fahringer. Automatic support for data distribution on distributed memory multiprocessor systems. Technical Report 93-2, Institute for Software Technology and Parallel Systems, University of Vienna, 1993.

[CF93b] J.-F. Collard and P. Feautrier. Automatic generation of data parallel code. In *Fourth International Workshop on Compilers for Parallel Computers*, Delft University of Technology, The Netherlands, December 1993.

[DR93] Alain Darte and Yves Robert. A graph-theoretic approach to the alignment problem. Technical Report 93-20, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, July 1993. To appear in Parallel Processing Letters. Available via anonymous ftp on `lip.ens-lyon.fr`.

[DR94] Alain Darte and Yves Robert. Mapping uniform loop nests onto distributed memory architectures. *Parallel Computing*, 20:679–710, 1994.

[Fea88] P. Feautrier. Array expansion. In *ACM Int. Conf. on Supercomputing*, Saint-Malo, July 1988.

[Fea91] P. Feautrier. Dataflow Analysis of Array and Scalar References. *Int. Journal of Parallel Programming*, 20(1):23–53, February 1991.

[Fea92a] P. Feautrier. Some Efficient Solutions to the Affine Scheduling Problem, Part I : One-dimensional Time. *Int. J. of Parallel Programming*, 21(5):313–348, October 1992.

[Fea92b] P. Feautrier. Some Efficient Solutions to the Affine Scheduling Problem, Part II : Multidimensional Time. *Int. J. of Parallel Programming*, 21(6):389–420, December 1992.

[Fea93] P. Feautrier. Toward Automatic Partitioning of Arrays on Distributed Memory Computers. In *ACM ICS'93*, pages 175–184, Tokyo, July 1993.

[GB92] M. Gupta and P. Banerjee. Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compilers on Multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):179–193, March 1992.

[IJT91] F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization: An overview of the PIPS project. In *ACM International Conference on Supercomputing*, Köln, Germany, 16-21 June 1991.

[JD89] P. Jouvelot and B. Dehbonei. A unified semantic approach for the vectorization and parallelization of generalized reductions. In *Procs. of the 3rd Int. Conf. on Supercomputing*, pages 186–194. ACM Press, 1989.

[KLS90] K. Knobe, J.D. Lukas, and G.L. Steele, Jr. Data Optimization: Allocation of Arrays to Reduce Communications on SIMD-Machines. *Journal of Parallel and Distributed Computing*, 8:102–118, 1990.

[LC90] J. Li and M. Chen. Index Domain Alignment: Minimizing cost of cross-referencing between distributed arrays. In *Frontiers'90, 3rd Symp. Frontiers Massively Parallel Computation*, College Park, MD, October 1990.

[LC91] J. Li and M. Chen. Compiling communication-efficient programs for massively parallel machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):361–376, July 1991.

[Mac87] M. Mace. *Memory Storage Patterns in Parallel Processing.* Kluwer Academic Publishers, 1987.

[Mel94] A. Meltzer. Programming for Performance in CRAFT on the T3D. Technical report, Cray Research Inc., Eagan, Minnesota, July 1994.

[Pla93] A. Platonoff. Which Fortran for Massively Parallel Programming ? Technical Report 93.09, IBP/MASI, Université P. et M. Curie (Paris 6), February 1993. Available via anonymous ftp on `ftp.ibp.fr`.

[Pla95] A. Platonoff. *Contribution à la Distribution Automatique des Données pour Machines Massivement Parallèles.* PhD thesis, Université P. et M. Curie, Paris, France, March 1995.

[RF93] X. Redon and P. Feautrier. Detection of reductions in sequentials programs with loops. In M. Reeve A. Bode and G. Wolf, editors, *5th Int. Parallel Architectures and Languages Europe*, pages 132–145, June 1993.

[RF94] X. Redon and P. Feautrier. Scheduling reductions. In ACM Press, editor, *Procs of the 8th ACM International Conference on Supercomputing*, pages 117–125, July 1994.

[RWF91] M. 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.

[TMC92] Thinking Machines Corporation, Cambridge, Massachussetts. *Connection Machine CM-5 Technical Summary*, January 1992.