Optimizing Communications by Using Compile Time Analysis

Mourad Raji Werth and Paul Feautrier

Laboratoire PRiSM-CNRS, Université de Versailles,
45, avenue des États-Unis, 78035 Versailles Cedex, France
{Mourad.Raji-Werth , Paul.Feautrier}@prism.uvsq.fr

Abstract. It is well known that communication time is one of the most limiting factors for obtaining high performance programs for parallel architectures.
In our parallelization approach, we first express communications using the `get` primitive since it is natural and easy to generate automatically. This solution is unfortunately inefficient. This paper addresses the transformation of the `get` communication schema into a less-costly one. Two tracks are explored : 1. transforming `gets` into `sends`, where *diffusion* is detected and generated. 2. detection of `shift` communications which are efficiently implemented due to their regular character. We also discuss the combination of these two optimizations and present some experimental results.

1 Introduction

In a parallel program generated by our parallelization method (summarized later), when a datum is not local, it has to be `got` from the processor which owns it. This is accomplished naturally by using the `get` primitive.
This solution has the advantage of being simple[1], but unfortunately it is inefficient. Indeed, the source processor pointed out by the `get` instruction can be entirely arbitrary; knowing that all active processors execute the `get` at the same time and in parallel, conflicts on the communication links may occur during the forwarding of the messages. Moreover, on certain machines (such as the CM-2), the target processor has first to send its address to the source processor in order to allow the former to send it the desired datum. As a consequence, `gets` are twice as expensive as `sends`.

The aim of this paper is to address the issue of communications optimization by proposing ways to replace the costly `get` communication schema by a cheaper equivalent one.
In fact, this amounts to a decomposition of the `get` operation into one or more elementary operations. Our goal is to detect the most economical decomposition.

After presenting an overview of our parallelization method in section 2, we first of all study the transformation of `gets` into `sends` in section 3, dealing with communications where all or some of the active processors ask for the same data.

[1] one can find the same choice in other works, see for example [1]

These communications, called **spreads**, are efficiently implemented on most architectures.
A second opportunity, in order to reduce the cost of communications, is the detection of the regular communications between neighbor processors called **shift**[2] communications. Owing to the regular communication pattern involved, these communications are also very efficiently implemented. This topic is presented in section 4, as well as a discussion concerning the possibility of combining the two optimizations together.
Many examples are given throughout the paper in order to illustrate our propositions. Section 5 deals with some experimental results. Finally we group our conclusions in section 6.

Before going farther, let us mention a similar work done by Li & Chen [2] where pattern matching techniques are used to detect the economical decomposition. Our approach, for out part, is based on linear algebra.

2 A Framework for a Systematic Parallelization Method

Our parallelization method starts with a program written in a conventional language –e.g. FORTRAN–, then produces the parallel form of the program using a systematic transformation scheme.
This form is then adapted to generate code for a given parallel architecture.
The underlying idea of our systematic transformation scheme is to retrieve a *space-time* mapping of the program. –a similar approach is used by the systolization community [3]–
Starting with a powerful tool, the *Data Flow Graph* [4][5] which provides, for every value manipulated in the program, its exact source, i.e. the exact instruction and the corresponding iteration (in the presence of a loop nest) that produces the value. Then the *schedule* [6] and the *placement function* [7] of the program instructions are derived. They are linear functions in terms of the surrounding loop indexes. The first one expresses the logical time at which a given iteration must be scheduled, the last one provides the coordinates of the virtual processor –within a multi-dimensional grid– that will execute the corresponding computation.
Afterwards the program is expanded in order to obtain its *single assignment form* [8] and the new loop nests are constructed. The logical time and the grids axes represent the new indexes. Each original loop nest is incarnated by a multi-dimensional virtual processors grid.
The generated program contains a single global sequential loop over the time having an entirely parallel body [9][10].

Unfortunately, the resulting body incorporates several overheads that limit the performances of the generated code. These overheads are due to the presence of instruction guards and conditional operands and to the high cost of communications . The first problem is dealt with in [11]. In this paper we deal with the second one.

[2] they are called **NEWS** communications too.

3 Transforming gets into sends

Recall that a typical instruction –say **r**–, within a transformed program, has the following form :

$$a_r(t, p_1, \ldots, p_n) = \ldots\ldots\ a_s(t-\tau, f_1(p_1, \ldots, p_n), \ldots, f_m(p_1, \ldots, p_n))\ \ldots\ldots$$

which gives, after distributing the iterations of **r** on a grid of dimension n and those of **s** on a grid of dimension m :

tempo = get $a_s(t-\tau)$ **from processor** $[f_1(p_1, \ldots, p_n), \ldots, f_m(p_1, \ldots, p_n))]$
$a_r(t) = \ldots\ldots$ **tempo** $\ldots\ldots$

This code is obviously executed by every active processor (those which verify a certain number of conditions, including the belonging of their coordinates to the intervals determined by the bounds of the spacial loop nest).

In this section we will seek to transform the **get** mentioned above into a **send**. By **send** we mean a 1-to-1 communication or a diffusion.

In the above schematic portion of code, a receiving processor $(p_1, \ldots, p_n)$ within the n-grid designates the sending processor by its coordinates $(q_1, \ldots, q_m) = [f_1(p_1, \ldots, p_n), \ldots, f_m(p_1, \ldots, p_n)]$ within the m-grid.
A processor within the n-grid is a receiving processor only if its coordinates verify the following inequations :

$$\begin{cases} l_1 \leq p_1 \leq u_1 \\ \ldots \quad \ldots \quad \ldots \\ l_n \leq p_n \leq u_n \end{cases} \tag{1}$$

where l_i and u_i are the bounds of the spatial loops.

Transforming **gets** into **sends** amounts to designating a set of processors $(q_1, \ldots, q_m)$ within the m-grid that will be in charge to send data to a set of processors within the n-grid. In order to achieve this, two things must be specified :

1. *The context* : to designate which of the processors belonging to the m-grid might be active.
2. *The destination* : each active processor $(q_1, \ldots, q_m)$ must know to which processor(s) it has to send its data.

From the inequalities 1 we define a set of processors within the m-grid, called *a priori active processors*, which verify the following conditions :

$$\begin{cases} L_1 \leq q_1 \leq U_1 \\ \ldots \quad \ldots \quad \ldots \\ L_m \leq q_m \leq U_m \end{cases} \tag{2}$$

where the L_i and the U_i may be computed for instance by using the Fourier-Motzkin elimination algorithm. These conditions form a part of the context. The a priori active processors are those which potentially will be in charge to send

their data. We will see later that in certain cases, the context will be augmented with other conditions, so as to reduce the sending processors to a subset of the a priori active processors.

Recall that the functions $f_1, \ldots, f_m$ are linear. The sending processors coordinates may be expressed in matrix form as follows :

$$\begin{pmatrix} q_1 \\ \vdots \\ q_m \end{pmatrix} = M_{m,n} \begin{pmatrix} p_1 \\ \vdots \\ p_n \end{pmatrix} + K_{m,1}, \text{ or in a condensed way : } \mathbf{q} = \mathbf{Mp} + \mathbf{k}.$$

Searching for the destination is then summarized by the search of the solution of a parametric linear system of m equations and n unknowns, the parameters are $q_1, \ldots, q_m, t$ and the so-called *structure parameters* (the symbolic constants of the program).

In fact, our goal is to determine, for a processor $\mathbf{q}$ belonging to the m-grid, the addressee processor(s) within the n-grid. This amounts to expressing $\mathbf{p}$ in terms of $\mathbf{q}$ which is a classical linear algebra problem.
The general solution $\mathbf{p}_{gen}$ is given by : $\mathbf{p}_{gen} = \mathbf{p}_{hom} + \mathbf{p}_{par}$, where $\mathbf{p}_{par}$ is a particular solution of the system and $\mathbf{p}_{hom}$ is the solution of the homogeneous system : $\mathbf{Mp} = \mathbf{0}$.

It is well known that $\mathbf{p}_{hom}$ is nothing else than the kernel of the linear application characterized by $\mathbf{M}$. Let r be the rank of $\mathbf{M}$, $\mathbf{p}_{hom}$ is a sub-vector space of dimension $n - r$ for which it is easy to build a basis [12][13]. $\mathbf{p}_{gen}$ is simply the linear sub-space parallel to $\mathbf{p}_{hom}$ passing by $\mathbf{p}_{par}$. It is thus sufficient to find a particular solution to build the solution of the general system because we know how to characterize $\mathbf{p}_{hom}$.

One can note that the only solutions we are interested in are those belonging to the n-grid and verifying the conditions 1. For that reason and in order to insure the belonging of the solution to $\mathcal{N}^n$, a supplementary condition will be added later to the context.

Moreover, it is known that the image of the linear application characterized by $\mathbf{M}$ is a vector space of dimension r which is generated by r linearly independent columns of $\mathbf{M}$. This allows us to identify with precision the sending processors.

We have presented in the above the general theoretical solution to the problem of transforming `gets` into `sends`. Roughly, this solution is summarized as follows : *"The processors of the m-grid belonging to the image of the linear application characterized by* $\mathbf{M}$ *send their data to the processors of the linear space* $\mathbf{p}_{gen}$*"*.

In practice, a symbolic version of the Gauss algorithm is used to resolve the system of equations.

In order to go thoroughly into the above-mentioned theoretical solution, we shall study now three major cases: n is the dimension of the receiving grid, m is the dimension of the sending grid and r is the rank of the matrix $\mathbf{M}$.

3.1 First case : $n = r$

This coincides with the case where a total correspondence between the sending and the receiving processors is assured : each sending processor transmits its data to one and only one receiving processor .

In fact, since $n = r$ there are n lines of M which are linearly independent. Let $\mathbf{M'_{n,n}}$ be a sub-matrix of M constituted from such lines. For the sake of simplicity, let's suppose that it concerns the n first lines. The other $m - n$ lines may then be expressed in terms of the n first ones as follows :

$$\begin{cases} q_{n+1} = g_{n+1}(q_1, \ldots, q_n) \\ \vdots \\ q_m = g_m(q_1, \ldots, q_n) \end{cases} \tag{3}$$

it is now easy to transform **gets** into **sends** : an a priori active processor $(q_1, \ldots, q_m)$ which verify the conditions 3 must send its data to the processor:

$$\begin{pmatrix} p_1 \\ \vdots \\ p_n \end{pmatrix} = {M'^{-1}}_{n,n} \begin{pmatrix} q_1 \\ \vdots \\ q_n \end{pmatrix} - {M'^{-1}}_{n,n} K',$$

K' is composed by the n first components of K.
Obviously, only those processors belonging to the m-grid and verifying the above conditions have to transmit their data. These conditions are thus added to the context to which we add also the condition : ${M'^{-1}}_{n,n} \begin{pmatrix} q_1 \\ \vdots \\ q_n \end{pmatrix} - {M'^{-1}}_{n,n} K' \in \mathcal{N}^n$ in order to consider only solutions with integer components.

It should be noticed that the set of sending processors may be reduced to a grid of dimension r just as is the image of the linear application. We say that "the space of effectively active processors is 'smaller' than the space of the a priori active processors."

3.2 Second case : $m = r$

$m = r$ which implies that there are no redundant equations. We know that $n \geq m$. If $n = m$, this case coincides with the previous one treated above. If $n > m$, the system has $n - m$ free variables, which means that the kernel of the linear application is of dimension $n - m$ just as the linear space of solutions $\mathbf{p}_{gen}$. Indeed, in this case there is no total correspondence between the sending and the receiving processors. We thus are in a **spread** situation where a processor $(q_1, \ldots, q_m)$ has to diffuse its data among all the processors of $\mathbf{p}_{gen}$[3] (taking into account the bounds of the p_i).

[3] we recall that $\mathbf{p}_{gen}$ is the general solution to the parametric system of equations, it is thus expressed in terms of $q_1, \ldots, q_m$.

This diffusion as it is sketched above in its algebraic expression, is not tractable on most present architectures. However, some new promising approaches, like the one of A. Mérigot [14] and his team on associative nets, seem to offer some new possibilities for the exploiting of this kind of spreading.

Fortunately, there are a number of important particular cases that can be treated by most of the existing machines. Before tackling this issue, let us first briefly present how spreading functions.

In order to achieve a **spread**[4] operation, the processors within a given grid are grouped into classes which are defined by considering the processors grid and a given axis. Precisely, two processors belong to the same class along axis i, if and only if their coordinates, except for the i^{th} component, are strictly identical. Restated, a diffusion along axis i of an information owned by a processor $(p_1, \ldots, p_i, \ldots, p_n)$ can be done only along the line of processors parallel to axis i and passing by the point $(p_1, \ldots, p_i, \ldots, p_n)$. We note this diffusion as :
$(p_1, \ldots, p_i, \ldots, p_n) \longrightarrow (p_1, \ldots, \bullet, \ldots, p_n)$.
There is another diffusion primitive called `multi-spread` where it is possible to arrange several classes into one.
Let $i_1, \ldots, i_k$ be k axes of the grid. Two processors belong to the same class along with axes $i_1, \ldots, i_k$, if and only if their coordinates, except for the $i_1^{th}, \ldots, i_k^{th}$ components, are strictly identical. We note this diffusion schema by :
$(p_1, \ldots, p_{I_1}, \ldots, p_{I_k}, \ldots, p_n) \longrightarrow (p_1, \ldots, \bullet, \bullet, \bullet, \bullet, \bullet, \ldots, p_n)$
(supposing for the sake of simplicity that the k axes are contiguous).
In other terms, the diffusion can be achieved only along with those hyperplans that are parallel to a vector sub-space generated by a subset of canonical vectors.
Finally, it should also be stated that during a **spread** operation, only active processors might receive the data.

According to the above, the particular diffusion cases that are tractable are those where the schemas of the **spread** or `multi-spread` primitive above-mentioned are brought out; i.e something like : $(p_1, \ldots, p_n) \longrightarrow (p_1, \ldots, \bullet, \bullet, \ldots, p_n)$
Consequently, within the linear system of equations, *a certain number of variables -say $p_{i_1}, \ldots, p_{i_k}$- may take arbitrary values and in addition they do not participate in the expressions of the remaining variables.* This means that the k variables are simply absent from the original system. It is easy to notice that under these conditions, k is none other than $n - r$.
Algebraically spoken, this coincides with a kernel P_{hom} generated by a subset of canonical vectors.

The original system can then be reduced to m equations with m unknowns (since $m = r$). The matrix is invertible, and the system has a unique solution $\begin{pmatrix} p_1 \\ \vdots \\ p_m \end{pmatrix} = \begin{pmatrix} h_1(q_1, \ldots, q_m) \\ \vdots \\ h_m(q_1, \ldots, q_m) \end{pmatrix}$. This situation may be illustrated by the communication schema : (supposing that the k variables correspond with the last k vari-

[4] the two terms spread and diffuse are used interchangeably.

ables) $(q_1, \ldots, q_m) \longrightarrow (h_1(q_1, \ldots, q_m), \ldots, h_m(q_1, \ldots, q_m), \overbrace{\bullet, \bullet, \bullet}^{k})$. In order to implement this communication schema, we use a **send** followed by a **multi-spread**:

$$\begin{pmatrix} q_1 \\ \vdots \\ q_m \end{pmatrix} \xrightarrow{\texttt{send}} \begin{pmatrix} p_1 = h_1(q_1, \ldots, q_m) \\ \vdots \\ p_m = h_m(q_1, \ldots, q_m) \\ \left.\begin{matrix} p_{m+1} = 0 \\ 0 \\ p_n = 0 \end{matrix}\right\} k \end{pmatrix} \xrightarrow{\texttt{multi-spread}} \begin{pmatrix} p_1 \\ \vdots \\ p_m \\ \left.\begin{matrix} \bullet \\ \bullet \\ \bullet \end{matrix}\right\} k \end{pmatrix}$$

We will see farther that this last communication schema can still further be optimized when h_i are **shift** functions.

3.3 Third case : $r < m$ and $r < n$

This case is in a way a combination of the two previous ones. $r < m$ means that $m-r$ equations are redundant (which is the same as saying that $m-r$ lines of **M** are linearly dependent of the others). We then have $m - r$ additional conditions that will join the context calculation just as in the first case.
r is strictly lower than n. This is a diffusion situation that has been treated in the second case.

Let us mention the particular case where $r = 0$, the matrix **M** is then null. In this situation, all processors ask for the same data owned by the processor of coordinates **k**. This cases coincides with a global static diffusion , i.e a **multi-spread** along with all axes. This is usually called **broadcasting**.

Example 1:

Consider a 3-grid. Figure 1 illustrates the situation where a processor (p_1, p_2, p_3)

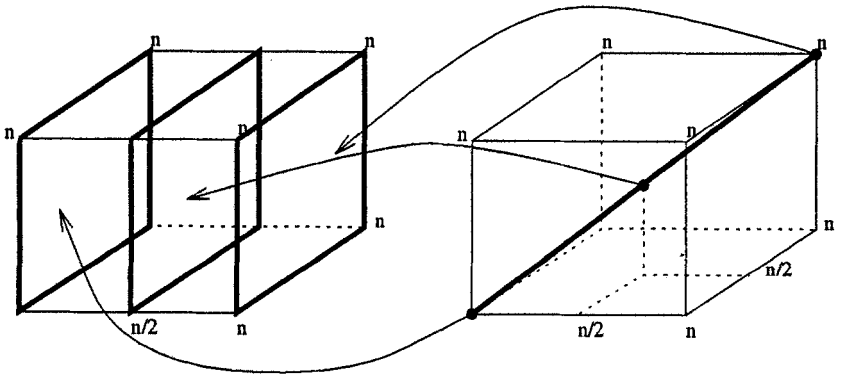

Fig. 1. Diffusion by diagonal

refers (using the **get** primitive) to a datum owned by the processor $(q_1, q_2, q_3) = (p_1, p_1, p_1)$.
The rank of the system is $r = 1$. The last two lines of the matrix are redundant (indeed identical) with the first. The kernel is of dimension 2 and generated by the canonical vectors : $(0,1,0)$ and $(0,0,1)$ which provide the spreading axes.
The diffusion is actually accomplished as follows : each processor (a, a, a) belonging to the cube's diagonal, sends its data to all processors belonging (at one and the same time) to the cube and the hyperplan of equation $p_1 = a$. Two supplementary conditions must thus be added to the context :

$$\begin{cases} q_2 = q_1 \\ q_3 = q_1 \end{cases}$$

The spreading schema may be illustrated as follows :

$$\begin{pmatrix} q_1 \\ q_2 \\ q_3 \end{pmatrix}_{/q_2=q_1 \wedge q_3=q_1} \longrightarrow \begin{pmatrix} q_1 \\ \bullet \\ \bullet \end{pmatrix}$$

Example 2:

Consider the following **get** operation : $(p_1, p_2, p_3) \longleftarrow (p_1 + p_2, p_1, 2p_1 + 2p_2, 2p_1 + p_2)$.
$n = 3$, $m = 4$, the parametric linear system of equations is given by :

$$\begin{cases} q_1 = p_1 + p_2 \\ q_2 = p_1 \\ q_3 = 2p_1 + 2p_2 \\ q_4 = 2p_1 + p_2 \end{cases}$$

The rank of the matrix is $r = 2$; it is then a diffusion issue. p_3 is absent from the system so the present case is thus tractable. Resolving the system gives :

$$\begin{cases} p_1 = q_2 \\ p_2 = q_1 - q_2 \\ p_3 = \bullet \end{cases}$$

One may notice that the last two equations (related to q_3 and q_4) are redundant with the first two. This leads us to add the following two conditions to the context :

$$\begin{cases} q_3 = 2q_1 \\ q_4 = q_1 + q_2 \end{cases}$$

Due to the fact that the receiving and the sending grids are not identical, this communication schema is implemented as illustrated below :

$$\begin{pmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \end{pmatrix}_{/q_3=2q_1 \wedge q_4=q_1+q_2} \xrightarrow{\textbf{send}} \begin{pmatrix} p_1 = q_1 \\ p_2 = q_1 - q_2 \\ p_3 = 0 \end{pmatrix} \xrightarrow{\textbf{spread}} \begin{pmatrix} p_1 \\ p_2 \\ \bullet \end{pmatrix}$$

Example 3:
Let's take up the **get** schema : $(p_1, p_2, p_3) \longleftarrow (p_1 + p_2, p_1 + p_2, p_1 + p_2)$. This is clearly a diffusion situation, but unfortunately it can not be treated.
Indeed, $n = m = 3$, $r = 1$, the kernel is of dimension 2 and is generated by the vectors $(-1, 1, 0)$ and $(0, 0, 1)$ which means that the spreading plane does not fit the algebraic conditions of tractability explained above. By the way, one can notice that only one variable (instead of two) is absent from the system.

4 Detecting shift communications

It is well known that regular communications, called **shift** or **NEWS** communications, are efficiently implemented on a large number of present parallel architectures. This is due to the fact that in such communications , all the receiving processors ask for data in such a regular pattern, that conflicts are completely avoided
We are typically in a **shift** communication situation when every active processor
$(p_1, \ldots, p_i, \ldots, p_n)$ within a given grid, asks for a datum from its neighboring processor $(p_1, \ldots, p_i - \alpha, \ldots, p_n)$.
The detection of this kind of communications is usually studied under the hypothesis that the sending and the receiving processors belong to the same grid. We will see that under certain conditions, this restriction may be removed.

Let us first study the case where $m = n$.
To detect a **shift** communication, one just has to scan the set of equations. Each time a relation of shape $q_i = p_i + \alpha$ is encountered, we deduce the presence of a **shift** communication along axis i in one direction or in another, depending of the sign of α.

Whenever **M** coincides with the identity matrix, the original **get** communication schema may be expressed by **shift** communications only. We shall see later that in certain cases it is possible to combine **shift** and **spread** communications.
The shift vector is given by **k**. When every component of **k** has a non zero value, a **shift** operation is accomplished along each axis of the grid.

Now, when $m \neq n$, the two grids are different and it is not directly possible to carry out **shift** communications.
We suppose that we have at our disposal a mechanism for axes mapping, which allows one to map out a set of axes of a given grid along with a set of axes of another grid. The mapping between axes is arbitrary, provided that each axis in the first grid has the same number of virtual processors as the corresponding mapped axis in the second grid[5].

[5] In other words, we need a facility that allows us to place a copy of a source field, taken from a given grid, into a destination field, in another grid. Specified axes and coordinates of the source grid are mapped to the specified axes and coordinates of the destination grid and so data is copied according to this mapping. Roughly, this can be accomplished by a relatively regular address computing, followed by a **send**.

Let us begin by processing the case of $m > n$. We have previously shown that under this circumstance $m - r$ equations are redundant, which is expressed by adding supplementary conditions to the context :

$$\begin{cases} q_{r+1} = g_{n+1}(q_1, \ldots, q_r) \\ \vdots \\ q_m = g_m(q_1, \ldots, q_r) \end{cases}$$

If among these $m - r$ conditions there are $m - n$ "projective conditions", i.e of shape $q_i = C^{st}$, then it is possible to accomplish a mapping between the $m - (m - n) = n$ remaining axes of the m-grid and the n-grid.
We are again in a similar situation to the one we have just treated above ($m = n$) where it is possible again to detect and to generate **shift** communications.

The remaining case is the one where $n > m$. There are thus $n - r$ free variables, this is actually a diffusion case. We have seen that the absence of $n - r$ variables from the system of equations allows the diffusion to be tractable. This condition is sufficient to carry out a mapping between the two grids[6].
In addition, if among the m equations there are r equations of shape: $q_i = p_i + \alpha_i$ (α_i is a possibly null integer), then the resulting communication schema of the diffusion can be optimized by replacing the **send** with a **shift** and then executing the **multi-spread**.

Example 4:

Consider the **get** operation : $(p_1, p_2) \longleftarrow (5, p_2 - 2)$. Without using **shift** communications, this schema would have been transformed as follows :

$$\begin{pmatrix} 5 \\ q_2 \end{pmatrix} \xrightarrow{\textbf{send}} \begin{pmatrix} p_1 = 0 \\ p_2 = q_2 + 2 \end{pmatrix} \xrightarrow{\textbf{spread}} \begin{pmatrix} \bullet \\ p_2 \end{pmatrix}$$

By noticing that the shape according with the second axis offers a **shift** situation, the schema may rather be transformed into :

$$\begin{pmatrix} 5 \\ q_2 \end{pmatrix} \xrightarrow{\textbf{shift}} \begin{pmatrix} p_1 = 5 \\ p_2 = q_2 + 2 \end{pmatrix} \xrightarrow{\textbf{spread}} \begin{pmatrix} \bullet \\ p_2 \end{pmatrix}$$

which is undeniably more efficient.
It may be also stated that during the **shift** operation, only processors belonging to line 5 have to execute the **shift** instruction along with the second axis. This is a part of the context specification.

Finally, let us mention another case that is not a priori a **shift** situation, but can be treated thanks to the mapping facility : if an equation of shape $q_i = p_j + \alpha$ is encountered, one can carry out a mapping between axis i and axis j and thus allowing for the accomplishing of a **shift** operation.
Obviously, the profitableness of this solution closely depends on the relative cost of the mapping operation in comparison with a simple **send**.

[6] One can notice that the absence of only $n - m$ variables is sufficient to achieve a mapping.

5 Experimental results

Our experiments were carried out on a 4K CM-200 system having 512 MB of memory and 128 Weiteks (64-bits floating point accelerators).
Our parallelizer generates code in C/Paris, since it is a sufficiently expressive language for our code generation and optimization needs.

We will present below the experimental results obtained for the Cholesky factorization algorithm.

It is very interesting to study in detail the resulting parallel Cholesky program, but due to lack of space, we will just present the performances of the parallel program as well as those of the optimized code.

We have detected the following transformations :

- two **send** communication schemas,
- one **shift** communication schema,
- and five **spread** communication schemas.

All **get** operations have been successfully transformed according to our techniques.

The table below gives the execution time of the resulting parallel program and that of the optimized one for different matrix sizes, as well as the efficiency gains produced by the above-mentionned optimizations.

n	63	127	255	511
the non-optimized program (sec)	0.88	4.98	33.33	256.14
the optimized program (sec)	0.56	1.48	6.28	35.67
efficiency gain	1.57	3.36	5.30	7.18

These results confirm the importance of communications, seeing that the communica optimized program runs up to 7 times faster than the initial non-optimized one.

6 Conclusions

We have examined the use of compile-time analysis of communication schemas as an optimization technique for code produced by a parallelizing compiler.

We have applied this technique to improve the efficiency of parallel code in two ways. The first optimization is to transform **gets** into **sends** and thus to identify the diffusions.

The second optimization detects **shift** communications.
We have also discussed the combination of these two optimizations.

We have shown performance improvements resulting from applying these optimizations to an implementation of Cholesky factorization running on a CM-200. The efficiency gains observed are substancial.

We believe that the basic ideas behind the techniques we have employed, while particularly important for SIMD-style machines, and most easily implemented in parallelizing compilers, are also applicable to MIMD machines and

other compiler approaches. In particular in a data-parallel language compiler environment. Indeed, template and alignment directives provide the necessary information in order to analyse communications in a similar way. Templates provide the shapes and the dimensions of the grids in question, while the alignment information provides the placement functions. This topic and the trial of our techniques in a MIMD environment, will be considered in future work.

Aknowledgment

We are grateful to our friend Abe Wiebe for proofreading the manuscript.

References

1. Luc Bougé: The Data-Parallel Programming Model: a Semantic Perspective. Tech. Report No LIP-IMAG 92-45 (1992)
2. Jingle Li and Marina Chen: Compiling Communication-Efficient Programs for Massively Parallel Machines. IEEE Trans. on Parallel and Distributed Systems **2** (1991) 361–376
3. P. Quinton: Automatic Synthesis of Systolic Arrays from Uniform Recurrence Equations. IEEE, Int. Symp. on Computer Architecture, Ann Arbor (1984) 208–214
4. Paul Feautrier: Dataflow Analysis of Scalar and Array References. Int. J. of Parallel Programming, **20(1)** (1991) 23–53
5. Dror E. Maydan and Saman P. Amarasinghe and Monica S. Lam: Array Dataflow Analysis and its Use in Array Privatization. Proc. of ACM Conf. on Principles of Programming Languages (1993) 2–15
6. Paul Feautrier: Some Efficient Solutions to the Affine Scheduling Problem, I, One Dimensional Time. Int. J. of Parallel Programming **21(5)** (1992) 313–348
7. Paul Feautrier: Toward Automatic Partitioning of Arrays on Distributed Memory Computers. ACM Int. Conf. on Supercomputing, Tokyo (1993) 175–184
8. Paul Feautrier: Array Expansion. ACM Int. Conf. on Supercomputing, St Malo (1988) 429–441
9. M. R. Werth and P. Feautrier: A Systematic Approach of Program Transformation. Procs of the Int. Workshop on Compiler for Parallel Computers, Paris (1990).
10. M. R. Werth and P. Feautrier: On Parallel Program Generation for Massively Parallel Architectures. High Performance Computing II (1991) M. Durand and F. El Dabaghi editors, Elsevier Science Publisher
11. M. R. Werth and J. Zahorjan and P. Feautrier: Using Compile-Time Conditional Analysis to Improve the Performance of Compiler Parallelized Programs. Proc. Int. Conference on Massively Parallel Processing, Applications and Development. (1994) Elsevier Science Publisher
12. Gilbert Strang: Linear Algebra and its Applications. Harcourt Brace Jovanovitch, New York (1988)
13. Antal E. Fekete: Real Linear Algebra. Marcel Decker Inc., New York (1988)
14. Alain Mérigot: Associative Nets: A New Parallel Computing Model. Tech. Repport 92-02 IUF-Université Paris Sud (1992)