Automatic Vectorization of Communications for Data-Parallel Programs

Cécile Germain[1] and Franck Delaplace[2]

[1] LRI-CNRS Université Paris-Sud
[2] Université d'Evry

Abstract. Optimizing communication is a key issue in compiling data-parallel languages for distributed memory architectures. We examine here the cyclic distribution, and we derive symbolic expressions for communication sets under the only assumption that the initial parallel loop is defined by affine expressions of the indices. This technique relys on unimodular changes of basis. Analysis of the properties of communications leads to a tiling of the local memory addresses that provides maximal message vectorization.

1 Introduction

Static analysis of data-parallel programs, for the generation of distributed code, has been proposed by many authors, for instance [5] [6] [8] [10] [15]. Static analysis aims to improve performance over run-time resolution [3] which includes a lot of pure overhead in form of guards and tests. Many static compilation schemes have been considered; they differ in important points such as interleaving computation and communication as in [6], or having identical management of local an non-local data such as in [8]. However, they all use three basic sets: *Compute(s)* is the part of the index set which is local to processor s; *Send(s)* (resp *Received(s)*) is the part of a distributed array that has to be sent (resp received) by processor s when owner computes rule is applied. The central problem of static analysis is to define these sets at compile-time, and in an efficient form.

Two major costs have to be considered for the code generation scheme: the computing cost, and the communication cost. The computing cost is all the overhead required to compute local indices, and, when a communication occurs, to compute the parameters of the communication, the destination processors and the local addresses. As pointed out by [6], naive resolution leads to a symbolic form involving integer divides for each forwarded data, which may be as inefficient as run-time resolution. The communication cost depends on the volume and number of communications. For a data-parallel program, the volume, i.e. the number of data to send to a remote processor, cannot be modified, because it is fixed by the placement function (e.g. ALIGN and DISTRIBUTE directives). At the code generation level, optimization is only directed towards the number of communications, by aggregating all data that are to be sent to the same processor. Although this may seem a very specialized problem, the overwhelming part of startup in message cost makes this optimization a major component of performance, as shown in [15].

To be amenable to static analysis, the references must be affine functions of the parallel loop indices,a reference being an access or alignement function, and the loop bounds must be defined by affine inequalities. These assumptions are the weakest possible. Under these assumptions, deriving efficient closed forms of the previous sets for the most general block-cyclic distribution is an open problem. [8] gives a general compiling scheme under the weakest assumptions, but provides closed forms only when indices are independent: for instance, $T[j,i]$, but not $T[2i+j,i-j]$. [5] uses a finite state machine approach, allowing optimal memory utilization, but restricts references to array sections and uses integer divides. [10] solves the same problem with a virtualization method. Other special cases have been solved, for unit strides in [15], for one-dimensional arrays in [6] and [14].

In this paper, we derive closed forms providing an efficient code generation scheme, under the weakest assumptions, when the parallel arrays are cyclically distributed. Next part formally states the problem and discusses the relationship with the problem of scanning integer polyhedra. Part three analyzes the conditions for message vectorization and proposes an explicit closed form achieving maximal vectorization; part four details the SPMD code and its optimizations, and presents some examples.

2 General Compilation Scheme

2.1 Problem Statement

We consider nested parallel loops, with given alignement and the cyclic distribution, such as described in High Performance Fortran (HPF); we restrict our analysis to the static subset of HPF where arrays are aligned once, at compile-time, and all index functions are affine; moreover, the index set must be described by affine inequations. The generic loop nest is :

```
forall i in C
        X(Bi + b) = f(Y(A1 i + a1), Z(A2 i + a2),... )
end forall
```

where B, A_1 and A_2 are integer matrices, and b, a_1 and a_2 are integer vectors.

Some notations must be defined, associated with the cyclic distribution: let $p_1, p_2, \ldots, p_n$ be the extents of the PROCESSOR target of the distribution, p be the vector with coordinates p_i, P the diagonal matrix with coefficients p_i, and $\mathcal{P}$ the processor set, i.e. $\mathcal{P} = \prod_i [0, p_i - 1]$. Template element j is laid on processor s such that $s_i \equiv j_i \bmod p_i$ for all $i = 1 \ldots n$. In the following, the coordinates subscripts are elided, and scalar operations are extended to vector ones by coordinates. Hence, array element j defines a set of spatial coordinates s and a set of memory coordinates t by euclidean division: $j = Pt + s$, with $0 \leq s < p$. For any s in $\mathcal{P}$, $\mathbf{Z}^n_s$ is the set of integer vectors congruent with s modulo p.

Distributed code for the previous loop can be generated at compile-time if *Compute(s)*, *Send(s)* and *Receive(s)* can be described for each processor s in a convex and generic form. Convex form means that the set can be parametrized by a variable such that the parametrization is one-to-one, and the parameter set is described by an affine inequality, i.e. is a convex polyhedron in $\mathbf{Z}^n$. From a convex polyhedron, generating a loop nest is theoretically possible. The practical issues will be discussed in the following part. As the matrices defining the references (A_1, A_2 and B) will be used to generate such convex sets, we assume that these matrices are constant. Generic means that the distributed program is in SPMD style: code is identical on all processors, possibly parametrized by the processor address.

2.2 An Integer Equation

All our results comme from Lemma 1 which solves equation $Mx = \alpha + Pk$ with x and k as unknowns, where M is an integer matrix and α an integer vector. This lemma is a simple mathematical exercise of unimodular change of basis, but introduces a lot of notations, that will be used throughout this paper.

If D is a diagonal $r \times r$ matrix, let $[D, 0]$ be the $n \times n$ matrix $\begin{bmatrix} D & 0 \\ 0 & 0 \end{bmatrix}$.
Smith normal form theorem [13] states that, given an integer $n \times n$ matrix Q with rank r, there exists a unique $r \times r$ diagonal matrix D such that d_i divides d_{i+1}, all $d_i \neq 0$, and $Q = H[D, 0]K$ with H and K unimodular. Let $\pi = det(P) = \prod p_i$; $\pi P^{-1}M$ is an integer matrix, hence may be decomposed as HDK. From this, the equation to solve may be rewritten:

$$H[D, 0]Kx = \pi P^{-1}\alpha + \pi k \ . \qquad (1)$$

Let $\beta = \pi H^{-1}P^{-1}\alpha$ (h is integer because H is unimodular). We can now state our lemma

Lemma 1. *Equation $Mx = \alpha + Pk$ has solutions iff $gcd(d_i, \pi)$ divides β_i. If x_0, k_0 is a solution, the solutions are, for all λ in $\mathbf{Z}^n$*

$$x = x_0 + K^{-1}P'\lambda \ ,$$
$$k = k_0 + HD'\lambda \ .$$

Proof. Let $y = Kx$, $h = H^{-1}k$. From the definitions of y, h and β, (1) becomes

$$[D, 0]y = \beta + \pi h \ . \qquad (2)$$

If $d_1, \ldots, d_r$ are the diagonal coefficients of D, and $d_{r+1}, \ldots, d_n$ are defined to be 0, (2) has solutions iff β_i is a multiple of $\delta_i = gcd(d_i, \pi)$. In this case, the gcd algorithm gives a particular solution (y_0, h_0). Let $d'_i = d_i/\delta_i$ and $p'_i = \pi/\delta_i$, and D' and P' the corresponding diagonal matrices; the $n - r$ last components of y_0 are 0, the $n - r$ last diagonal coefficients of D' are 0 and of P' are 1. □

We note *ex-cond(M, P, α)* the condition for existence of solutions; when necessary, subscripts and variables will indicate the dependence on the initial equation of the vectors and matrices involved in Lemma 1.

2.3 Local Sets

Compute Set Generating SPMD code for the compute part of the loop needs to define the local iteration set and the local memory locations that are accessed during each iteration. With Owner Computes Rule, an index i is in *Compute(s)* if $Bi + b = Pt + s$. Lemma 1 applied to equation $Bi = Pt + (s - b)$ gives:

Proposition 2. *Let* $\mathcal{L} = \{\lambda \in \mathbf{Z}^n | CK^{-1}P'\lambda \leq c - Cx_0\}$.
If ex-cond (B, P, s-b), $Compute(s) = \{x_0 + K^{-1}P'\lambda | \lambda \in \mathcal{L}\}$
else $Compute(s) = \emptyset$

Compute(s) is parametrized by the λ in $\mathcal{L}$. As K^{-1} is unimodular, and P' has no null coefficient, the enumerating scheme is one-to-one. Finally, $\mathcal{L}$ is a convex polyhedron.

For any index $i = x_0 + K^{-1}P'\lambda$, the local address for array element $Bi + b$ is $t = k_0 + HD'\lambda$, providing the location to write.

The following very simple example is often quoted:

forall (i = 0:n) T(i, i) = 0.0

Suppose T is cyclically distributed onto a (4, 8) PROCESSOR array. B being the reference matrix (2×2 matrix because T has a 2-D index space), we have:

$$B = \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} \text{ and } \pi P^{-1} B = HDK = \begin{pmatrix} 2 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 16 & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} .$$

From this, *ex-cond(B, P, s-b)* is 4 divides $s_2 - s_1$. In this case, let $g = (2s_1 - 2s_2)/8$ for short; the compiler has to find a solution of

$$\begin{cases} y_1 = s_2 + 8h_1 \\ 0 = g + h_2 \end{cases}$$

As $1 - 8 * 0 = 1$, the particular solution for y_1 is s_2, for h_1 is 0; $y_2 = 0$ by our algorithm, hence $h_2 = -g$. The compiler can deal with all these symbolic manipulations.

Inequality $0 \leq i \leq n$ is rewritten:

$$\begin{pmatrix} -1 & 0 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} \leq \begin{pmatrix} 0 \\ n \end{pmatrix} ,$$

If T_s is the local piece of array T, the code is

```
if ((s2 - s1)%4 == 0) then
  g = IntDiv(s1 - s2, 4)
  do l = IntDiv(-s2 + 7, 8), IntDiv(n - s2, 8)
       T_s(-g + 2 * l, l) = 0.0
endif
```

This example points out that the initial solutions (x_0 and k_0) are also symbolic: going back to (2), after simplification, we have to find y_0 and h_0 such that $d'_i y_i = \beta_i + p'_i h_i$, with d'_i and p'_i relatively prime. As we assumed these coefficients to be numerical constants, the gcd algorithm allows to find at compile-time u_i and v_i such that $d'_i u_i = 1 + p'_i v_i$. Hence for run-time β_i, the symbolic form of an initial solution is $(\beta'_i u_i, \beta'_i v_i)$, with $\beta_i = \beta'_i \delta_i$. This must be considered, because β_i are run-time quantities, coming from the processor number and possible variables in the references.

Send and Receive Sets A sending set or a receiving set is related to source and destination references, for instance A_1+a_1 and $B+b$ in our generic example. As sending and receiving sets are symmetrically defined, identical methods can be used to compute both of them, and we present only the method for the sending set. For clarity, subscripts are elided in A_1 and a_1. Formally, we define the sending set on processor s as

$$Send(s) = \{(s', j), s' \in \mathcal{P}, j \in \mathbf{Z}_s^n | \exists i \in \mathcal{C} \exists t' \in \mathbf{Z}^n : Bi + b = Pt' + s'; j = Ai + a\}$$

s' is the remote processor address (processor number) and t' is the remote memory address in processor s'. Let $j = Pt + s$; t is the memory address in processor s. In order to minimize the number of actual communications, this set should be enumerated first along the s' coordinates, and next the t ones. This is the so-called called *message vectorization* [15].

Hence we have to solve in s', t', t, i the system with parameter s:

$$\begin{cases} Ai + a = Pt + s \\ Bi + b = Pt' + s' \end{cases}$$

Valid solution must verify

$$\begin{cases} Ci \leq c \\ s' \in \mathcal{P} \end{cases}$$

We defer the solution to the next section, and we first discuss the relationship of code generation with an extensively studied topic, scanning polyhedra [2] [9] [11] [12]. Clearly, the code generation problem may be restated as a polyhedron scanning problem. For instance, *Compute(s)* may be rewritten as the polyhedron in $\mathbf{Z}^n \times \mathbf{Z}^n$:

$$\{(t, i) | Bi = Pt + s; Ci \leq c\} \ .$$

In these sets, some variables completely determine other ones (e.g. i defines t in *Compute(s)*). As the final code uses only some variables (t for *Compute(s)*), we need to enumerate the projection of a convex polyhedron, which is not always convex. Many libraries are available for this kind of loop generation (Omega Calculator [12], LIC [11]). However, if these tools scan very efficently the polyhedra created by a block distribution, they generate very poor code or are oveflowed in the cyclic case. For instance, *Compute(s)* of the previous example is defined by

$$\{(t_1, t_2, i) | i = 4t_1 + s_1; i = 8t_2 + s_2; 0 \leq i \leq n\} \ ,$$

and the best-effort loops generated by the Omega Calculator are:

```
do t1 = IntDiv(-s1 + 3, 4): IntDiv(-s1 + n, 4)
      if ((s1 + 4*t1 - s2) % 8 == 0) then
        t2 = IntDiv(s1 + 4*t1 - s2 + 7, 8)
      endif
```

The test is executed at each loop iteration, while our solution has only one test. The problem is worst for communication, because of the higher dimensionality of the polyhedron. Hence the compiler has to provide some *a priori* tiling of the processors and memory spaces.

3 Vectorization

As vectorization is a major source for communication performance [15], analyzing the conditions where vectorization may occur is the first task. Let the lhs side of the parallel affectation be T[i], i.e. matrix B of the previous part equals Id and $b = 0$. The general problem comes down to this case when B is unimodular; extending our framework to the general case case is straightforward, but leads to clumsy formulas. In this case:

$$Send(s) = \{(s', j), s' \in \mathcal{P}, j \in \mathbf{Z}_s^n | \exists i \in \mathcal{C}, \exists t' \in \mathbf{Z}^n : i = Pt' + s', j = Ai + a\} \ .$$

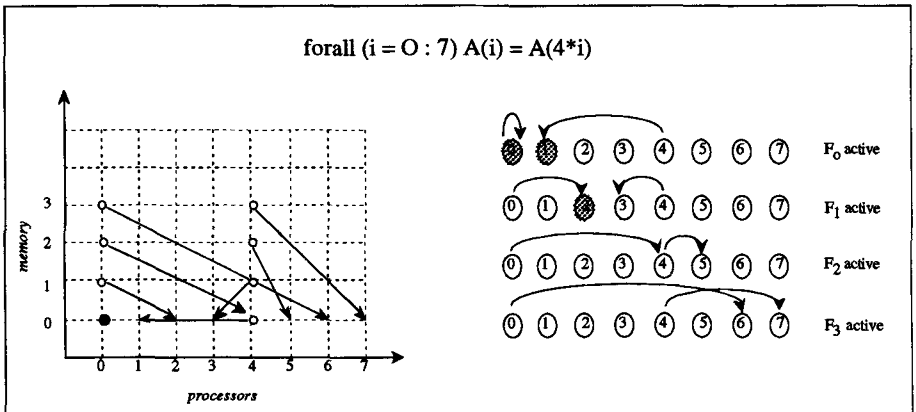
3.1 Tiling the Index Set

A set of array elements on a processor is candidate to be aggregated in a unique message if all elements have the same destination processor. Such a set will be called *vectorizable* in the following.

Two data i_1 and i_2 have vectorizable images if they are on the same processor, that is $i_1 = Pt'_1 + s'$ and $i_2 = Pt'_2 + s'$, and if their images are on the same processor, that is $Ai_1 \equiv Ai_2 \bmod p$. Next definition formalizes this idea:

Definition 3. A subset $\mathcal{T}$ of $\mathbf{Z}^n$ is a *remanence set* for A if

$$\forall t_1, t_2 \in \mathcal{T}, AP(t_1 - t_2) \equiv 0 \bmod p \ .$$



a) global diagram of communications

b) decomposition of communications along free sets

Fig. 1. A remanent but not free communication

Hence data candidate to be aggregated must be defined from a remanent set. This is not a sufficient condition, as exmplified in fig. 1: A and P being

1-dimensional, the remanence property is always true; however, only local addresses congruent modulo 4 can be aggregated inside the same message. This comes from the fact that different data inside the same processor are required by different processors for the same memory slice. Next definition formalizes this idea:

Definition 4. A subset $\mathcal{S}$ of P is a *free set* for A if

$$\forall s_1 \neq s_2 \in \mathcal{S}, A(s_1 - s_2) \not\equiv 0 \bmod p \ .$$

Proposition 5 elucidates the relationship between remanence sets, free sets and vectorization.

Proposition 5. *If $\mathcal{T}$ is a remanent set and $\mathcal{S}$ is a free set, the image by A of $\mathcal{T} \times \mathcal{S}$ is a vectorizable set on each processor.*

Proof. Let j_1 and j_2 be on processor s, and fulfill the conditions of the previous proposition: $j_1 = A(Pt'_1 + s'_1)$ and $j_2 = A(Pt'_2 + s'_2)$, with t'_1 and t'_2 belonging to a remanence set and s'_1 and s'_2 to a free set. From the fact that j_1 and j_2 are on the same processor, $A[P(t'_1 - t'_2) + s'_1 - s'_2] \equiv 0 \bmod p$. As t'_1 and t'_2 belong to the same remanence set, $A(s'_1 - s'_2) \equiv 0 \bmod p$; thus $s'_1 = s'_2$, because they are in the same free set. □

From this proposition, tiling the array elements following maximal remanence and free sets creates maximal vectorization. Proposition 6 gives closed form of these sets. Let Smith normal form of the integer matrix $\pi P^{-1}AP$ be $H_1 D_1 K_1$, P'_1 be defined as P'_{AP}, and p'_1 be the vector of the diagonal coefficients of P'_1.

Proposition 6. *Let $\mathcal{T}(u) = \{u + K_1^{-1}P'_1 v, v \in \mathbf{Z}^n\}$. The set of $\mathcal{T}(u)$, for $0 \leq K_1 u < p'_1$, is a partition of $\mathbf{Z}^n$ is maximal remanent sets. Moreover, for all t in $\mathcal{T}(u)$,*

$$APt = APu + PH_1 D'_1 v \ .$$

Proof. Let t be an integer vector; t belongs to the set $\mathcal{T}(u)$ such that $K_1 u$ is the remainder in the integer divison of $K_1 t$ by p'_1. As K_1 is unimodular, and $K_1 u$ is uniquely determined, u exists and is uniquely defined, proving the partition of $\mathbf{Z}^n$ by the $\mathcal{T}(u)$. To prove that each $\mathcal{T}(u)$ is maximal, let t_1 and t_2 be in $\mathcal{T}(u_1)$ and $\mathcal{T}(u_2)$; if t_1 and t_2 form a remanent set, the following equality is true:

$$AP(u_1 - u_2 + K_1^{-1}P'_1(v_1 - v_2)) \equiv 0 \bmod p \ .$$

From lemma 1, this implys

$$u_1 - u_2 + K_1^{-1}P'_1(v_1 - v_2) = K_1^{-1}P'_1\lambda \ ,$$

that is $u_1 = u_2 + K_1^{-1}P'_1\mu$. From unicity of euclidean division, $K_1 u_1 = K_1 u_2$, and from unimodularity $u_1 = u_2$.

If t is in $\mathcal{T}(u)$, $t - u = K_1^{-1}P'_1 v$. From lemma 1, this implys $AP(t - u) = PH_1 D'_1 v$. This proves the last part of the proposition. □

A maximal free set is defined by

$$\mathcal{F}(\lambda) = \{s \in \mathcal{P} | P'\lambda \leq Ks < P'(\lambda + 1)\} \ .$$

However, enumerating all remanence sets and all free sets on each sending processor would create useless iterations. In the example of section 3.3, there are six free sets, but only four need to be enumerated on processor 0. Next section precises our enumeration scheme.

3.2 SPMD Code

The basic idea of our scheme is to enumerate maximal remanence sets, then free sets, to create vectorizable communications. Closed forms are possible because enumeration of the free sets depends on an external index which denotes the remanence set.

The sending set may be expressed as

$$\begin{aligned} Send(s) = \{&(s', Pt + s), s' \in \mathcal{P}, t \in \mathbf{Z}^n | \\ &\exists t' \in \mathbf{Z}^n : Pt' + s' \in \mathcal{C}; A(Pt' + s') + a = Pt + s\} \ . \end{aligned}$$

Let t' be in $\mathcal{T}(u)$; we have to solve in s' and t:

$$A(P(u + K_1^{-1}P_1'v) + s') + a = Pt + s \ .$$

By proposition 6, this equation becomes

$$A(Pu + s') = P(t - H_1D_1'v) + s - a \ . \tag{3}$$

Consider (3) as an instance of $Ax + a \equiv s \bmod p$; solutions exist if *ex-cond(A, s-a)*. Note that *ex-cond* depends only on s and a. If this condition is satisfied, let $x_0 = x_0(A, P, s - a)$ and $k_0 = k_0(A, P, s - a)$. The solutions of (3) are $s' = x_0 - Pu + K^{-1}P'\lambda$ and $t = k_0 + H_1D_1'v + HD'\lambda$ with λ in $\mathbf{Z}^n$. A correct SPMD code will be achieved if all constraints on solutions can be expressed in convex form for the parameters (u, λ, v). From the definition of *Send(s)*, and of $\mathcal{T}(u)$, there are three constraints, which define three index sets:

(a) K_1u is a remainder in division by p_1'. Let $\mathcal{U} = \{u \in \mathbf{Z}^n | 0 \leq K_1u < p_1'\}$.
(b) $s' \in \mathcal{P}$. Let $\mathcal{L}_u = \{\lambda \in \mathbf{Z}^n | -x_0 + Pu \leq K^{-1}P'\lambda < -x_0 + P(1 + u)\}$.
(c) $Pt' + s' \in \mathcal{C}$. Let $\mathcal{V}_\lambda = \{v \in \mathbf{Z}^n | CPK_1^{-1}P_1'v \leq c - C(x_0 + K^{-1}P'\lambda)\}$.

All these sets are convex polyhedra. Finally, the SPMD code for the sending part is:

```
if ex-cond
  compute x0 and k0
  do u in U
      do λ in L_u
          do v in V_λ
              send (k0 + HD'λ + H1D1'v, -Pu + x0 + K^-1 P'λ)
```

The first parameter of the *send* it the local address of the data, and the second is the destination processor.

3.3 Example

The current HPF benchmark set is somehow limited. In fact, in all codes that we had, only the block distribution is used. Hence, we have to consider an artificial example.

forall (i= 0:n, j= 0:m) T(i, j) = T(2j, i + j)

on a 4×8 PROCESSOR set.

The reference matrix being A, we have

$$A = \begin{pmatrix} 0 & 2 \\ 1 & 1 \end{pmatrix}, \pi P^{-1} A P = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 16 & 0 \\ 0 & 128 \end{pmatrix} \begin{pmatrix} 1 & 2 \\ 0 & 1 \end{pmatrix}, \pi P^{-1} A = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 4 & 0 \\ 0 & 16 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}.$$

From this, the new index sets are defined by

$$\mathcal{U} = \{(u_1, u_2) | 0 \le u_1 + 2u_2 \le 1; 0 \le u_2 \le 0\} \ ,$$
$$\mathcal{L}_u = \{(\lambda_1, \lambda_2) | 0 \le 8\lambda_1 - 2\lambda_2 - 4u_1 + x_1 \le 3; 0 \le 2\lambda_2 - 8u_2 + x_2 \le 7\} \ ,$$
$$\mathcal{V}_\lambda = \{(v_1, v_2) | 0 \le 8v_1 - 8v_2 + 8\lambda_1 - 2\lambda_2 + x_1 \le n; 0 \le 8v_2 + 2\lambda_2 + x_2 \le m\} \ .$$

The SPMD code is then :

```
if (s1 % 2 == 0)
  x1 = s1; x2 = s1/2 ; k1 = k2 = 0;
  do u1 = 0 : 1
   do l2 = IntDiv(1 - x2, 2) : IntDiv(7 - x2, 2)
    do l1 = IntDiv(7 - x1 + 4*u1 + 2*l2, 8) : IntDiv(3 - x1 + 4*u1 +2*l2, 8)
     do v2 = IntDiv(7 - x2 -2*l2, 8) : IntDiv(m - x2 -2*l2, 8)
      do v1 = IntDiv(7 - x1 + 2*l2 - 8*l1 + 8*v2, 8) :
                           IntDiv(n - x1 + 2*l2 - 8*l1 + 8*v2, 8)
       send ( (l2 + 4*v2, l1 + v1), (x1 + 8*l1 - 2*l2 - 4*u1, x2 + 2*l2))
```

The loop bounds were obtained by submitting separately the $\mathcal{U}$, $\mathcal{L}_u$ and $\mathcal{V}_\lambda$ sets to the Linear Inequality Calculator, with constant propagation from each set to the following; here u2 is found equal to 0.

3.4 Analysis

As shown by the form of the general SPMD code, the destination processor does not depend on the innermost loop index v, and all parameters of the *send* primitive are affine functions of the loop indices. For n = m, there are at most 4 messages, proving that good vectorization is possible, even in this complicated case.

Loop bounds are in convex form; only the vector term (c in $Ci \le c$) depends on an external loop index. Each of the three loops $\mathcal{U}$, $\mathcal{L}_u$ and $\mathcal{V}_\lambda$ is at most as deeply nested as the initial loop; this is a key point: for instance, in the previous (contrivied) example, generating the loop bounds was immediate, but submitting the global system fails. The particular solutions are computed as in the case of *Compute(s)*.

Run-time integer divides appears only in computation of loop bounds; in many cases (see the previous example), there is no actual integer divide, because the divider is always a power of 2.

Another important property that the code is fully symbolic: all matrices are derived from the initial matrix A, the parallel loop bound matrix C, and the processor matrix P, allowing further optimizations of SPMD code based on loop transformations.

4 Optimizations

The most general case is, in fact, quite rare. Most practical programs will present some pecularities that may simplify the compilation process and the output code. Our output code regularly improves with the simplicity of the input code.

4.1 Array Sections

Parallel references using regular spacing are known as *array sections*. A generalization is to allow permutations of the indices, such as *T(i, j)* = *T(3*j, 3*i)*. In this case, and if the PROCESSOR extents ar all powers of 2, the loop bounds present no integer divides. To prove this, note that in our framework, generalized array sections create a A matrix with only one non-null coefficient on each row (the stride), and a C matrix with only one non-null coefficient on each row, this coefficient being equal to 1. It follows from the form of A that K and $K^{-1} = Id$. On the other hand, all the diagonal coefficients of P' and P'_1 divide π, which is a power of 2. From the form of the sets $\mathcal{U}$, $\mathcal{L}_u$ and $\mathcal{V}_\lambda$, it follows that the divisons will be only by powers of 2.

4.2 Remanent References

If all data required by each processor s' are sent by the same processor (depending on s'), reference A will be called *remanent*. In this case, there is only one maximal remanence set, $\mathbf{Z}^n$ itself; thus loop u would have to disappear, as shown below.

From the definition of remanence sets, A is remanent iff $P^{-1}AP$ is an integer matrix, say B. This condition can be easily checked by the compiler. For instance, A is remanent when it is diagonal, for all one-dimensional arrays, as matrix P reduces to a scalar, and for any PROCESSOR geometry where all p_i are equal.

Let Smith normal form of B be $H_2D_2K_2$. From unicity of Smith normal form, $\pi D_2 = D_1$, thus π divides d_i^1; as $p_i'^1 = \pi/gcd(\pi, d_i^1)$, $P'_1 = Id$; finally condition (a) results in $u = 0$, destroying the external loop. With some more manipulation, one can define an index set $\mathcal{W}_\lambda$such that the final loop becomes:

```
if ex-cond
  do λ in ℒ
       do w in 𝒲_λ
            send (k_0 + HD'λ + Bw, x_0 + K^{-1}P'λ)
```

4.3 Free References

If there is only one free set, reference A will be called free. In this case, a processor always sends its data to the same processor. As the solutions in x of equation $Ax = Pk$ are $x = K^{-1}P'\lambda$, a sufficient condition for A to be free is that $P^{-1}K^{-1}P'$ is an integer matrix, say Q. This condition can be easily checked by the compiler. When A is remanent and free, only one loop remains. This true for shifts, and when matrix A is diagonal with coefficients relatively prime with the p_i. One can choose x_0 such that $[P^{-1}x_0] = 0$, because the only requirement on x_0 is $Ax_0 \equiv s - a \bmod p$, and by the remanence property of A, the solutions in x of this type of equation are defined modulo p. $u = 0$ because A is remanent, $Q\lambda = u$ because A is free and the choice of x_0, ; as $detQ \neq 0$ (from its definition), $\lambda = 0$. Set $\mathcal{W}$ reduces to $\{w \in \mathbf{Z}^n | CPw \leq c - Cx_0\}$. As A defines a one-to-one mapping of $\mathcal{S}$ onto $\mathcal{S}$, *ex-cond* disappears. The final loop is:

```
do w in W
    send (k0 + Bw, x0)
```

On an 8×8 PROCESSOR, the parallel assignment
*T(i, j) = T(3*j, 3*i)*
becomes

```
do w1 = IntDiv(-x1 + 7, 8), IntDiv(n - x1, 8)
    do w2 = IntDiv(-x2 + 7, 8), IntDiv(w1 + x1 - x2, 8)
        send( (k1 + 3*w2, k2 + 3*w1), (x1, x2))
```

This exemplifies the fact that remanent and free communications can be perfectly vectorized.

5 Conclusion

Although many data-parallel languages do propose both block and cyclic distribution, most existing codes only use the block one. The motivation is that blocking provides locality. However, the cyclic distribution may be a key for sparse computations [1], which are a prominent component of numerical codes. The last part of this paper shows that, at least for many frequent cases, the cyclic distribution does not require a larger number of communications than the block one, although it increases the volume of each communication.

In this paper, we focused on the basic sets associated with SPMD code for communications. Another possible application is escaping from Owner Compute Rule, when remote computations are possible. The array elements involved in this local computation may be, once again, determined by our initial lemma. A different communication model is compiled communications, as proposed in [4] and [7]; in this model, the full communication scheme has to be known, to allocate network resources at compile-time. With some adaptation, the scheme presented here meets these requirements.

References

1. R. Asenjo and al. Sparse Block and Cyclick data distribution for matrix computation. In *High Performance Computing, Technology and Applications*, Elsevier, 95.
2. C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *3rd ACM Symp. on Principles and Practice of Parallel Programming*, pages 39–50, 91.
3. D. Callahan and K. Kennedy. Compiling programs for distributed memory architectures. *Jal. Supercomputing*, (2):151–169, Oct. 88.
4. F. Cappello and al. Balanced distributed memory parallel computers. In *22nd Int. Conf. on Parallel Processing*, pages I.72–I.76–, August 93.
5. S. Chatterjee and al. Generating local addresses and communication sets for data-parallel programs. In *Symp on Principles and Practice of Programming Languages 93*. ACM, 93.
6. C.Koelbel. Compile-time generation of regular communication patterns. In *Supercomputing 91*, pages 101–110, 91.
7. A. Feldmann, T.M. Stricker, and T.E. Warfel. Supporting sets of arbitrary connections on iWarp through communication context switches. In ACM, editor, *5th ACM Symp. on Algorithms and Architectures*, pages 203–212, 93.
8. F.Irigoin and al. A linear algebra framework for static HPF code distribution. In *4th Int. Work. on Compilers for Parallel Computers*, pages 117–132, 93.
9. M. Le Fur. Scanning parameterized polyhedron using Fourier-Motzkin elimination. Technical report, IRISA, Sept. 94. PI 858.
10. S.K.S. Gupta and al. On compiling array expressions for efficient execution on distributed-memory machines. In *1993 Int. Conf. on Parallel Processing*, pages II-301–II-305, 93.
11. D.E. Maydan, S.P. Amarasinghe, and M.S. Lam. Data dependence and data-flow analysis of arrays. In *5th Work. on Languages and Compilers for Parallel Computing*, pages 283–292, 92.
12. W. Pugh. The omega test : a fast and practical integer programming algorithm for dependence analysis. *Comm. ACM*, (8):102–114, Aug. 92.
13. A. Schrijver. *Theory of Linear and Integer Programming*. Wiley, 86.
14. J.M. Stichnoth, D. O'Hallaron and T.R. Gross. Generating Communications for Array Statements ; Design, Implementation and Evaluation. *Jal. Parallele and Distributed computing*, (21) 150–159, Apr. 94.
15. C-W. Tseng. *An optimizing Fortran D compiler for MIMD Distributed- memory machines*. PhD thesis, Rice University, 93.