Compiling the Block-Cyclic Distribution

Cécile Germain [a] and Franck Delaplace [b]

[a] *LRI CNRS-Université Paris-Sud*

[b] *Université d'Evry*

Abstract

This paper presents a framework for compiling the communications created by general affine references to block-cyclically distributed arrays in HPF style. Closed form for the basic sets are described. The scheme provides optimal vectorization in the cyclic(1) case, and experimental results show that it is very efficient in the cyclic(k) case.

1 Introduction

Static analysis of data-parallel programs, for the generation of SPMD code for distributed memory architectures, has been proposed by many authors, for instance [0,0,0,0,0]. Static analysis aims to improve performance over run-time resolution [0] which includes a lot of pure overhead in form of guards and tests. All use three basic sets: *Compute(s)* is the part of the index set which is local to processor s; *Send(s)* (resp *Receive(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. This paper extends a previous work [0] dealing with the cyclic distribution to the most general cyclic(k) distribution.

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 scan the basic sets. The communication cost depends on the volume and number of messages. At the code generation level, the volume, i.e. the number of data to send to a remote processor, is constrained by the mapping function (e.g. HPF ALIGN and DISTRIBUTE directives); thus optimization is only directed towards the number of messages, by aggregating all data that are to be sent to the same processor (*message vectorization*). Although this may seem a very specialized problem, the overwhelming part of startup in the communication cost makes this optimization a major component of performance, as shown in [0].

2 Problem Statement

We consider nested parallel loops, where all arrays indices are affine functions of the loop indices, and the index set is described by affine inequations.

FORALL i in $\mathcal{I}$ X[i] = f(Y[Ai + a])

where $i = (i^1, i^2, \ldots, i^q)$. Compiling the communication sets needs to describe them in a convex and generic form. Convex form means that the set can be parameterized by a vector variable such that the parameterization is one-to-one, and the parameter set is described by affine inequalities, i.e. is a convex polyhedron in $\mathcal{Z}^n$. From a convex polyhedron, generating a loop nest is theoretically possible, and in practice if its dimension is low. Generic means that the distributed program is in SPMD style: code is identical on all processors, possibly parameterized by the processor address.

Some notations must be defined, associated with the cyclic(b) distribution; let $p^1, p^2, \ldots, p^n$ be the extents of the PROCESSORS (in HPF style) 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. $\mathcal{P} = \prod_i [0, p^i - 1]$. Similar notations are used for the vector of block sizes b; scalar operations including inequalities are extended point-wise; the left hand side and the right hand side of the assignment may have different distributions, for which the l and r subscripts are used. Then, the layout of X is uniquely defined by:

$$i = B_l(P_l t + s) + d \;\; 0 \leq d < b_l \;\; 0 \leq s < p_l \;\; ,$$

$X(i)$ is on processor s, in the memory slice t, and inside this memory slice at offset d. An analogue formula holds for Y.

3 Tiling the Index Set

The method is presented here on the example of computing the *Send* set, but is equally applicable to *Compute* and *Receive*. The elements of the *Send* set on a processor s that are to be sent to a processor s' are simply defined by the fact that j is in *Send (s)* iff: $Y(j)$ is owned by s, $X(i)$ is owned by s', and there exists an i in $\mathcal{I}$ such that $j = Ai + a$. With the distribution functions, the previous conditions can be rewritten formally as:

$$\begin{aligned} A[B_l(P_l t' + s') + d'] + a &= B_r(P_r t + s) + d && (1)\\ 0 \leq s' &< p_l && (2)\\ 0 \leq d &< b_r && (3)\\ 0 \leq d' &< b_l && (4)\\ B_l(P_l t' + s') + d' &\in \mathcal{I} && (5) \end{aligned}$$

where s and a are parameters, and s', t', t, d and d' are the unknowns. Solving the first equation relies on a technical lemma based on Smith Normal Form of integer matrices [**?**].

Lemma 1 *Let M be an integer matrix and Q a diagonal integer matrix. The solutions in $\mathcal{Z}^n$ of equation $Mx = Qy + \alpha$ are*

$$\alpha = Rk, x = Uk + S\lambda, y = Vk + W\lambda.$$

where R, U, V, S and W are integer matrices depending only on M and Q. Moreover, R and S are non-singular (i.e. their determinant in non-null).

Two array elements are vectorizable if they are image by the affine application of a valid index set, are laid on the same processor and are to be sent to the same processor. We consider first the cyclic case ($B_l = B_r = \mathrm{id}$). In the cyclic case, the following definitions and proposition provide a tiling of the index set following the memory and processor axes respectively, that creates maximal message vectorization.

Definition 1 *A subset $\mathcal{V}$ of $\mathcal{I}$ is* vectorizable *if*

$$\forall i_1, i_2 \in \mathcal{V}, i_1 = Ai'_1 + a, i_2 = Ai'_2 + a, i_1 \equiv i_2 \bmod p_r \Rightarrow i'_1 \equiv i'_2 \bmod p_l \ .$$

A subset $\mathcal{T}$ of $\mathcal{Z}^n$ is remanent *for A if*

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

A subset $\mathcal{S}$ of $\mathcal{P}_l$ is free *for A if*

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

Proposition 1 *The set $\{A(P_l t + s); t \in \mathcal{T}; s \in \mathcal{S}\}$ defines a vectorizable set on each processor if and only if $\mathcal{T}$ is remanent and $\mathcal{S}$ free.*

Proof. All proofs are omitted for lack of space, and can be found in [0].

Maximal remanent sets are $\mathcal{Z}^n$ (memory space) lattices, while maximal free sets are polyhedral tiles of the processor set. Proposition 2 gives a closed form of the remanent sets. Let S_0 be the S matrix of lemma 1, for the equation $AP_l x = P_r y$.

Proposition 2 *Let $\mathcal{T}(u) = \{u + S_0 v, v \in \mathcal{Z}^n\}$. There exists an integer unimodular matrix K_0 and an integer vector q such that the set of $\mathcal{T}(u)$, for $0 \leq K_0 u < q$, is a partition of $\mathcal{Z}^n$ is maximal remanent sets.*

In the block-cyclic case, the block distribution introduces integer divides, which break the linearity of the communication map. The remanent and free sets can no more be defined through linear equations As we require all the sets being defined in convex form, the vectorizable sets will be splitted with respect to offsets. This technique adapts the virtual-block idea of [0] to the general affine framework.

4 SPMD Code

The basic idea of our scheme is to have five nested loops. The two outermost loops enumerate the valid offsets, the next two loops the maximal remanent sets and free sets for this particular combination of offsets ; at this level, the destination processor is identified, hence the innermost loop enumerates all the data that are to be sent to this processor. Each "loop" actually is a loop nest, at most as deeply nested as the initial parallel loop.

4.1 Address Computation

Equation (1) may be rewritten:

$$AB_l(P_lt' + s') = B_rP_rt - Ad' + d - a + B_rs, \tag{6}$$

and we want to solve this equation with s' (the destination processor) as a primary unknown, to allow vectorization. Eq. (6) is an instance of lemma 1 , with $M = AB_l$, $Q = B_rP_r$, and $\alpha = -Ad' + d - a + B_rs$. Let R_1, U_1, V_1, S_1 and W_1 be the integer matrices associated to these particular M and Q as in lemma. For a given d', eq. (6) has solutions if $-Ad' + d - a + B_rs = R_1k$. Brute application of the lemma 1 to eq. (6) does not give a satisfactory solution, because we will have $P_lt' + s' = U_1k + S_1\lambda$, and the destination processors (s') will be scanned as a function of the t', providing no vectorization. As in the cyclic case, we have to define the remanent sets. Here, these sets are defined by $AB_lP_lx \equiv 0 \bmod b_rp_r$. This is a new instance of lemma 1, with $M = AB_lP_l$, $Q = B_rP_r$, and $\alpha = 0$. From Lemma 1, the solutions are $x = S_2v$, where S_2 is the S matrix associated to these particular matrices M and Q. Let $t' = u + S_2v$; as in the cyclic case, the u are defined by $0 \leq K_2u < q_2$. Plugging the definition of t' into eq. (6), we have $AB_l(P_lu + s') = B_rP_r(t - W_2v) + R_1k$. Using once again lemma 1, we get:

$$\begin{aligned} s' &= -P_lu + U_1k + S_1\lambda, \\ t &= W_2v + V_1k + W_1\lambda. \end{aligned}$$

Equation (7) allows the computation of the destination processor. For a given s', the v parameter in eq.7 scans the set of data to send to s'.

4.2 *Code Generation*

The constraint that d and d' must be offsets (3)and (4) gives the scanning of the d' and k. The constraint that s' must be a valid processor number (2) gives the scanning of the λ. Finally, the iteration space (constraint (5)) gives the scanning of the v. More precisely, let

$$\mathcal{D}' = \{d' \in \mathcal{Z}^n | 0 \leq d' < b_l\}, \mathcal{K}_{d'} = \{k \in \mathcal{Z}^n | 0 \leq R_1 k + Ad' + a - B_r s < b_r\},$$

$$\mathcal{U} = \{u \in \mathcal{Z}^n | 0 \leq K_2 u < b_r p_r\}, \mathcal{L}_{u,k} = \{\lambda \in \mathcal{Z}^n | 0 \leq -P_l u + U_1 k + S_1 \lambda < b_l\},$$

$$\mathcal{V}_{d',u,k,\lambda} = \{v \in \mathcal{Z}^n | B_l(P_l S_2 v + U_1 k + S_1 \lambda) + d' \in \mathcal{I}\}.$$

With these notations, the SPMD code for the sending part is shown in figure 1

```
do d' in D'
   do k in K_d'
      do u in U
         do λ in L_u,k
            s' = -P_l u + U_1 k + S_1 λ
            mess_exists = FALSE;
            do v in V_d',u,k,λ
               pack(mess, B_r(W_2 v + V_1 k + W_1 λ) + Ad' + a - B_r s + R_1 k);
               mess_exists = TRUE;
            endo
            if (mess_exists) send (mess, s');
endo endo endo endo
```

Fig. 1. Scanning the *Send* set

The receive code must match each call to a send from any processor by a call to a receive, and store the incoming data at the correct location. The scanning of the message set relies on the fact that for a given destination offset and free set, at most one message can be received. A formal proof a this result can be found in [0]. The memory slice is computed with the previous definition $t' = u + S_2 v$.

5 Experimental Results

Despite its theoretical advantage, the algorithm described above cannot be considered for solving the communication vectorization problem in compilers and run-time systems for data parallel languages, unless it allows for an efficient implementation. We now describe an implementation experiment, and a performance comparison with a commercial compiler, which shows ...

6 Related Work

A survey of HPF compilation issues and an extensive bibliography can be found in [0]. Efficient computation and vectorization of communication is an open problem for the general affine reference problem, with the general cyclic(b) distribution. The known methods fall in three classes. Iterative methods [0,0] record at run-time the behavior of communication over a period, and reuse the recorded schedule. This scheme does not require compile-ti;e knowledge of the structural parameters, but are restricted to array sections and square index sets. The second method relies on scanning integer polyhedra [0,0,0]. The underlying Fourier-Motzkin algorithm is very efficient for the block distribution, but may generate poor code or be overflowed in the cyclic and block-cyclic cases, because the input to the Fourier-Motzkin algorithm may have the size of all the memory and processor variables cumulated. To avoid this explosion, the order of variable elimination may be directed by the internal needs of the algorithm, at the possible expense of loosing vectorization. The last class is direct solvers, which aim to exhibit closed forms of the communications. This is exemplified in [0,0], but is limited in these papers to array sections and square index sets. The work presented here extends the scope of the direct solver method to the most general affine references and index sets.

7 Conclusion and Future Work

This paper describes a scheme for computing and vectorizing communication in the most general regular case, block-cyclic distributions and affine array references. Such general references are not common in hand written code, but are likely to appear when automatic parallelization tools are applied. A prototype implementation has been described. Preliminary studies show that interfacing this tool with the ADAPTOR compiler [0] is possible by extending the DALIB run-time system.

The paper has 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 determined by our initial lemma, in fact more easily than for the communication problem.

References

[1] T. Brandes and F. Zimmermann. ADAPTOR: A transformation Tool for HPF programs. In *Programming Environments for Massively Distributed Systems*, pages 91–96, 94.

[2] D. Callahan and K. Kennedy. Compiling programs for distributed memory architectures. *Jal. Supercomputing*, (2):151–169, Oct. 88.

[3] S. Chatterjee, J.R. Gilbert, F. Long, R. Shreiber, and S-H. Teng. Generating local addresses and communication sets for data-parallel programs. In *Symp on Principles and Practice of Programming Languages 93.* ACM, 93.

[4] C.Koelbel. Compile-time generation of regular communication patterns. In *Supercomputing 91*, pages 101–110, 91.

[5] F.Irigoin, C. Ancourt, F. Coelho, and R. Keryell. A linear algebra framework for static HPF code distribution. In *4th Int. Workshop on Compilers for Parallel Computers*, pages 117–132, 93.

[6] C. Germain and F. Delaplace. Compiling the block-cyclic distribution.

[7] C. Germain and F. Delaplace. Automatic vectorization of communications for data-parallel programs. In *Europar95, LNCS966*, pages 429–440. Springer Verlag, 1995.

[8] 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.

[9] K. Kennedy, N. Nedeljkovic, and A. Sethi. Communication generation for cyclic(k) distributions. In *3rd Work. on Languages compilers and run-time systems for parallel processing*, pages 185–196. Kluwer, 95.

[10] 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.

[11] W. Pugh. The omega test : a fast and practical integer programming algorithm for dependence analysis. *Comm. ACM*, (8):102–114, Aug. 92.

[12] C-W. Tseng. *An optimizing Fortran D compiler for MIMD Distributed– memory machines.* PhD thesis, Rice University, 93.