Communication Pre-evaluation in HPF

Pierre Boulet[1] and Xavier Redon[2]

[1] LIP, École Normale Supérieure de Lyon, 46, allée d'Italie, F-69364 Lyon cedex 07, France

[2] LIFL, Univeristé des Sciences et Technologies de Lille, Bâtiment M3, Cité Scientifique, F-59655 Villeneuve d'Ascq cedex, France

Abstract. Parallel computers are difficult to program efficiently. We believe that a good way to help programmers write efficient programs is to provide them with tools that show them how their programs behave on a parallel computer. Data distribution is the major performance factor of data-parallel programs and so automatic data layout for High Performance Fortran programs has been studied by many researchers recently. The communication volume induced by a data distribution is a good estimator of the efficiency of this data distribution.
We present here a symbolic method to compute the communication volume generated by a given data distribution during the program writing phase (before compilation). We stay machine-independent to assure portability. Our goal is to help the programmer understand the data movements its program generates and thus find a good data distribution. Our method is based on parametric polyhedral computations. It can be applied to a large class of regular codes.

1 Introduction

Parallel computing has become the solution of choice for heavy scientific computing programs. Unfortunately parallel computers require considerable knowledge and programming skills to exploit their full potential. A major mean to reduce the programming complexity is to use high-level languages. High Performance Fortran (HPF) is such a language. It follows the data-parallel programming paradigm where the computations are directed by the data. Indeed, the computations occur on the processor that "owns" the data being written. So the data distribution is a very important efficiency factor when programming with a data-parallel language.

Our goal is to help the programmer find a good data-distribution. We want a tool that is executed when the program is being written. Work on such a tool has started at the LIFL with HPF-builder [6], a tool that interactively displays the arrays as they are distributed and aligned by the HPF directives. This tool also allows to change the data distribution graphically.

We propose here a new step in the development of a tool that helps the programmer understand how data move in its program. Given a data-distribution, we are able to compute the volume of the communications generated by a program. We use symbolic computation tools to stay free of the problem size. All

this is done at the language level, thus retaining portability and machine independence. This tool is intended as a mean to write a reasonably efficient program that can be tuned for a particular parallel machine with profiling systems later.

The sequel of this paper is organized as follows. In Sect. 2 we briefly review related works, then in Sect. 3 we describe the problem we consider and present its modelization in Sect. 4. We then detail the tools used to solve our problem along with an example in Sect. 5. And we finally conclude in Sect. 6.

2 Related Work

Many researchers [1, 9, 10, 13–15] have studied automatic data distribution. Estimating communication costs has been the key factor to determine the quality of a data distribution. Most of the previous works [12, 7, 11] have studied compile-time estimation of these communication costs. Indeed, they use the fact that most program parameters are known at that time and in many cases, these studies also use machine (and compiler) dependent data. Our work differs from previous work by the techniques used and the stage of program development we focus on: the program writing phase. We also use exact parameterized methods and we stay compiler and machine independent, we work at the language level.

Description of the Problem

3.1 General Remarks

The problem we study in this paper is the evaluation of the communication volume in a HPF program before compilation. The goal is to help the programmer understand the communications generated by his program and find a good data distribution (or a more efficient way to code his algorithm).

The communications we consider are at the `PROCESSORS` level: as in HPF, we use an abstract target machine. We stay at the language level, thus allowing to find a data distribution that is well suited to the problem, and thus retaining portability. Our aim is not to find the best data distribution for a given machine, but a good one for any machine (and compiler).

In the future, if some compiler optimization techniques such as overlap areas to vectorize communications are used by most compilers, we could adapt our evaluation techniques to these optimizations. In a first step, though, we remain at the language level to validate our approach. We believe indeed that a good data distribution at the language level should not be a bad one for any compiler, regardless of the compilation techniques used.

3.2 Modelization

We consider that the only communication generation statements are storage statements, we do not take into account I/O statements. For each of these storage

statements, the surrounding loops define an iteration domain that "shapes" the communication pattern.

Given the mathematical tools we use in the following, we have to restrict ourselves to loop bounds that define a polyhedron, that is loop bounds defined as extrema of affine functions of the surrounding loop indices and parameters. For the same reason, we restrict the array access functions to affine functions. This class of loops contains most of the linear algebra routines.

To simplify the following discussion, we make the hypotheses below without loss of generality:

- Scalar values are described as arrays with one element.
- Storage statements read only one array reference that may not be aligned with the result value. Indeed, any more complex statement can be decomposed in a sequence of such statements with temporary storage [3, 2].
- All the arrays of the considered storage statement are aligned with respect to the same template T. This restriction makes sense since the distributions of computation related arrays should be related in some way. Actually most HPF programs contain only one template with all the arrays aligned onto. The domain of this template T is noted $\mathcal{D}_T$

Given the previous restrictions, a storage statement $\mathcal{S}$ is represented by:

$$W_{\mathcal{S}}(\phi_{W_{\mathcal{S}}}(I)) = f_{\mathcal{S}}(R_{\mathcal{S}}(\phi_{R_{\mathcal{S}}}(I))) \tag{1}$$

where the iteration vector I has value in the iteration domain $\mathcal{D}_{\mathcal{S}}$ defined by the surrounding loops. $W_{\mathcal{S}}$ and $R_{\mathcal{S}}$ are arrays and $\phi_{W_{\mathcal{S}}}$ and $\phi_{R_{\mathcal{S}}}$ are affine access functions.

We call *operation* an instance $\mathcal{S}(I)$ of a statement $\mathcal{S}$ for a given value I of the iteration vector.

The number of communications generated by a given statement is the number of array elements that need to be communicated for some operation generated by this statement. There is a communication when the array elements being read are not on the same virtual processor than the one the written element is. As array elements can be replicated, we will focus on the template elements.

4 Algebraic Method to Evaluate HPF Communications

As said in the previous section, we evaluate the communication cost in a HPF program by counting the number of communications between template elements.

4.1 Formula for Communication Cost Evaluation

There is a communication between two template elements if they are not distributed onto the same virtual processor and if the computation of a value to be stored in the first template element uses a value stored in the second one.

To be more precise, let us consider a storage statement $\mathcal{S}$ (as defined in the previous section) and a template T. We also need to define some functions:

- Function $\mathcal{O}_T^{\mathcal{S}}$ gives, for an element J of $\mathcal{D}_T$, the set of storage operations from $\mathcal{S}$ which compute values to be stored in $T(J)$;
- Function τ gives, for an operation o from $\mathcal{S}$, the set of template subscripts from where o may read its data.
- Last, function π_T is the distribution function which maps the template T on the virtual processors.

The number of template communications generated by statement $\mathcal{S}$ is equal to the number of elements in the union of the sets $\mathcal{C}_J(\mathcal{S})$:

$$\mathcal{C}(\mathcal{S}) = \bigcup_{J \in \mathcal{D}_T} \mathcal{C}_J(\mathcal{S}) = \bigcup_{J \in \mathcal{D}_T} \{(J, o) \mid o \in \mathcal{O}_T^{\mathcal{S}}(J),\ \pi_T(J) \notin \pi_T(\tau(o))\} \ . \quad (2)$$

Note that $\mathcal{C}(\mathcal{S})$ is a set of couples and not of mere operations. Indeed we may have to count several times the same operation, so we added the template subscript to distinguish different occurrences of the same operation. Computing the set $\mathcal{C}(\mathcal{S})$ implies the application of a change of basis on the program loop nests. Indeed, in the original program, the loops are used to enumerate the operations in the program execution order and the set $\mathcal{C}(\mathcal{S})$ enumerates them following the template iteration space.

4.2 Formal Representation of HPF Alignments

To perform this change of basis the only informations needed are the HPF `ALIGN` directives. Basically HPF alignments are a sub-set of linear alignments but a replication symbol $*$ has been added. Because of the replication symbol, a HPF alignment cannot be defined by a linear transformation from the array space to the template space.

A convenient way to represent a HPF alignment α is to use two linear transformations γ and δ. Transformation δ defines the replication part of the alignment. Let I be a subscript of an array `A` aligned on a template `T` using α. The set of the subscripts of `T` on which the data `A`(I) is stored is:

$$\{J \mid J \in \mathcal{D}_T,\ \gamma(I) = \delta(J)\} = \delta^{-1}(\gamma(I)) \ .$$

Let us consider the alignment:

```
!HPF$ ALIGN A(i) WITH T(i,*)
```

The first transformation from the array space to an intermediate template space is obtained by removing the replication symbols in the directive: $\gamma : i \mapsto i$. The second transformation is the projection from the template space to the intermediate template space: $\delta : \begin{pmatrix} i \\ j \end{pmatrix} \mapsto i$.

With this representation of a HPF alignment one can explicit the function τ of Sect. (4.1). Let us consider a statement $\mathcal{S}$ as defined in (1) and an alignment τ_{R_S} for array R_S on the template T defined by $(\gamma_{R_S}, \delta_{R_S})$. In this context, the following holds:

$$\forall I \in \mathcal{D}_T,\ \tau(\mathcal{S}(I)) = \tau_{R_S}(\phi_{R_S}(I)) = \delta_{R_S}^{-1}(\gamma_{R_S}(\phi_{R_S}(I))) \ .$$

4.3 Enumeration of Operations in the Template Iteration Space

The main problem of enumerating operations in the template space is that there may be several operations corresponding to a template element $T(J)$. I.e. more than one operation may produce a value to be stored in $T(J)$. In fact, the solution lies again in the formal representation of HPF alignments. Let us denote by τ_{W_S} the alignment for array W_S. According to Sect. 4.2, the alignment can be written as $\tau_{W_S} = \gamma_{W_S} \circ \delta_{W_S}^{-1}$. Therefore, the set $\mathcal{O}_T^S(J)$ introduced in the beginning of this section is defined by:

$$\mathcal{O}_T^S(J) = \{\mathcal{S}(I) \mid I \in \mathcal{D}_S,\ \phi_{W_S}(I) \in \gamma_{W_S}^{-1}(\delta_{W_S}(J))\} \ .$$

Let us consider the code fragment below:

```
!HPF$ ALIGN M(i,j) WITH T(i,*)
DO i=1,n
  DO j=1,m
    M(i,j) = ...                (s1)
  END DO
END DO
```

The aligment of M on T is defined by: $\left(\gamma \ : \ \begin{pmatrix} i \\ j \end{pmatrix} \mapsto i,\ \delta \ : \ \begin{pmatrix} i \\ j \end{pmatrix} \mapsto i\right)$. Since the subscript function for M is the identity, the set $\mathcal{O}_T^{s1}$ is such that:

$$\mathcal{O}_T^{s1}\begin{pmatrix} i' \\ j' \end{pmatrix} = \left\{ s1\begin{pmatrix} i \\ j \end{pmatrix} \mid \gamma\begin{pmatrix} i \\ j \end{pmatrix} = \delta\begin{pmatrix} i' \\ j' \end{pmatrix} \right\} = \left\{ s1\begin{pmatrix} i' \\ j \end{pmatrix} \mid j \in [1,m] \right\} \ .$$

This means that a template element T(i,j) owns the full row of rank i of array A.

In conclusion, the set $\mathcal{C}(\mathcal{S})$ may be computed using the subscript functions in $\mathcal{S}$ for W_S and R_S, the alignments of W_S and R_S on the template T and the distribution of T on the virtual processors:

$$\mathcal{C}(\mathcal{S}) = \bigcup_{J \in \mathcal{D}_T} \{(J, \mathcal{S}(I)) \mid I \in \Phi_{W_S}^{-1}(J),\ \pi_T(J) \notin \pi_T(\Phi_{R_S}(I))\} \qquad (3)$$

with the functions Φ_{W_S} and Φ_{R_S} defined as follows:

$$\Phi_{W_S} = \tau_{W_S} \circ \phi_{W_S} = \delta_{W_S}^{-1} \circ \gamma_{W_S} \circ \phi_{W_S} \ ,$$
$$\Phi_{R_S} = \tau_{R_S} \circ \phi_{R_S} = \delta_{R_S}^{-1} \circ \gamma_{R_S} \circ \phi_{R_S} \ .$$

4.4 Formal Representation of HPF Distributions

A HPF distribution directive for a template T can be represented using a projection ρ_T and an integer vector κ_T of size the dimension of the virtual processors grid P. Projection ρ_T selects the dimensions of T to be distributed on P. The dimension of T projected on the i^{th} dimension of P is distributed according to the pattern CYCLIC$((\kappa_T)_i)$. We denote by $A^{\min}$ (respectively by $A^{\max}$) the

vector of lower bounds (respectively the vector of upper bounds) for array A. In this context the distribution π_T can be explicited:

$$\pi_T(J) = P^{\min} + (\rho_T(J - T^{\min}) \div \kappa_T) \% (P^{\max} - P^{\min} + \mathbb{1}) \ . \qquad (4)$$

In the previous expression, the operator $\div$ (respectively the operator %) represents an element-wise integer division (respectively modulo) on vectors. One may remark that a BLOCK distribution can be achieved on the i^{th} dimension of P using a relevant $(\kappa_T)_i$ value:

$$(\kappa_T)_i = \left\lceil \frac{T_i^{\max} - T_i^{\min} + 1}{P_i^{\max} - P_i^{\min} + 1} \right\rceil \ .$$

5 Tools for the Evaluation of HPF Communications

In the previous section we have reduced the problem of evaluating HPF communications of a statement $\mathcal{S}$ to counting the elements of a set $\mathcal{C}(\mathcal{S})$. This section presents the tools used to automatically build $\mathcal{C}(\mathcal{S})$ and to count its elements. We illustrate the use of these tools on the following HPF program:

```
Program MatInit
!HPF$ PROCESSORS P(8,8)                          do i=1,n
!HPF$ TEMPLATE T(n,m)                              do j=1,m
!HPF$ DISTRIBUTE T(CYCLIC,CYCLIC) ONTO P             A(i,j)=B(i) (S1)
  real A(n,m), B(n)                                end do
!HPF$ ALIGN A(i,j) WITH T(i,*)                     end do
!HPF$ ALIGN B(i) WITH T(i,1)                      end
```

We have integrated the different tools into an interactive program called CIPOL. CIPOL provides a lisp-like textual interface to the tools, and pretty-prints their results.

5.1 Manipulation of Polyhedra

Our main tool is the polyhedral library developed by Wilde [16]. Most of the operations on polyhedra (union, intersection, image, etc.) are implemented in this library which allows to define a polyhedron by a set of constraints or a set of rays. For our application, we only need the first definition scheme.

The generic sets $\Phi_{W_S}^{-1}(J)$ and $\Phi_{R_S}(I)$ described in the previous section are computed using the *image* and *pre-image* functions of the polyhedral library.

The generic sets for our example are:

$$\Phi_A^{-1}(j_1, j_2, n, m) = \begin{cases} 1 \leq j_1 \leq n \wedge i_1 = j_1 \\ 1 \leq i_2 \leq m \end{cases} , \Phi_B(i_1, i_2, n, m) = \begin{cases} 1 \leq i_1 \leq n \wedge j_1' = i_1 \\ j_2' = 1 \end{cases} .$$

The first set means that the i^{th} row of array A is duplicated on each element of the i^{th} row of template T. The second set shows that array B is aligned with the first column of template T.

5.2 Using the PIP Software to Compute Inclusion

The evaluation of the non inclusion in definition (2) is not easy to implement since it implies that a generic condition must be verified for *each* value from a given set. It is simpler to verify an inclusion condition. In this case we just have to verify that a condition is verified for *one* value from a given set. Hence an inclusion condition can be modelized by an integer programming problem and can be resolved by a software such as PIP (see [8]). So, in place of counting the number of elements in $\mathcal{C}(\mathcal{S})$ we compute the number of elements in the set $\overline{\mathcal{C}(\mathcal{S})}$:

$$\overline{\mathcal{C}(\mathcal{S})} = \bigcup_{J \in \mathcal{D}_T} \{(J, \mathcal{S}(I)) \mid I \in \Phi_{W_S}^{-1}(J),\ \pi_T(J) \in \pi_T(\Phi_{R_S}(I))\} \ . \tag{5}$$

The final result is obtained using the following relation:

$$\mathrm{Card}(\mathcal{C}(\mathcal{S})) = \mathrm{Card}(\bigcup_{J \in \mathcal{D}_T} \{(J, \mathcal{S}(I)) \mid I \in \Phi_{W_S}^{-1}(J)\}) - \mathrm{Card}\left(\overline{\mathcal{C}(\mathcal{S})}\right) \ . \tag{6}$$

The PIP software is able to find the lexicographical minimum of a parametric set of integer vectors defined by a set of linear constraints $S(P)$. It may also take into account linear constraints on the parameters, this other set of constraints C is called the context. We denote by $\mathrm{lexmin}(C, S(P))$ the result computed by PIP. Since the initial set of integer vectors is parametric, PIP does not return an unique vector but a *quast* (Quasi-Affine Selection Tree). Indeed, the minimum depends on the values of the parameters, hence PIP splits the domain of the parameters in sub-domains on which the minimum can be expressed in a parametric way. If there is no solution for a sub-domain of the parameter space (because for these values of the parameters $S(P)$ is void), PIP denotes by $\perp$ the lack of solution.

Let us solve the following integer program:

$$\mathrm{lexmin}(C, S(I,J)),\ C = \begin{cases} J \in \mathcal{D}_T \\ I \in \Phi_{W_S}^{-1}(J) \end{cases},\ S(I,J) = \begin{cases} J' \in \Phi_{R_S}(I) \\ \pi_T(J) = \pi_T(J') \end{cases} . \tag{7}$$

Consider now the sub-domains of the parameter space for which PIP gives a solution other than $\perp$. It is easy to deduce, from what we said about PIP, that the number of elements in these sub-domains is equal to the number of elements in $\overline{\mathcal{C}(\mathcal{S})}$.

Remember that PIP only deals with linear constraints with respect to the parameters and the variables of the problem. Function π_T involves euclidian divisions but there is a well known method to linearize π_T that may be found in [5]. We just have to introduce three new integer vectors to replace the initial definition (4) of the distribution function by a definition which gives $\pi_T(J)$ as the solution of the following system (the $*$ operator is an element-wise vector multiplication):

$$\begin{cases} \rho_T(J - T^{\min}) = N * \kappa_T + R \wedge N = Q * (P^{\max} - P^{\min} + \mathbb{1}) + \pi_T(J) - P^{\min} \\ P^{\min} \leq \pi_T(J) \leq P^{\max} \wedge N \geq 0 \wedge Q \geq 0 \wedge 0 \leq R < \kappa_T \end{cases} .$$

One may note that N_i gives the block number for $T(J)$ with respect to the ith dimension. The previous system is effectively linear only if κ_T and $P^{\max} - P^{\min}$ are constant vectors. Computation of a parametric solution in other cases is left for future work but it is possible to obtain a result by asking the user to provide the values of some key parameters.

Integer program (7) may so be rewritten with only linear constraints.

For our program example `MatInit`, the template is distributed on the processor grid using two `CYCLIC` patterns, hence the distribution is defined by:

$$\rho_T \begin{pmatrix} j_1 \\ j_2 \end{pmatrix} = \begin{pmatrix} j_1 \\ j_2 \end{pmatrix}, \quad \kappa_T = \begin{pmatrix} 1 \\ 1 \end{pmatrix} .$$

The integer problem to solve is $\text{lexmin}(C(n,m), S(i_1,i_2,j_1,j_2))$ with

$$C(n,m) = \{\, 1 \le j_1 \le n \wedge 1 \le j_2 \le m \wedge i_1 = j_1 \wedge 1 \le i_2 \le m \,,$$

$$S(i_1,i_2,j_1,j_2) = \begin{cases} j_1' = i_1 \wedge j_2' = 1 \wedge 1 \le p_1 \le 8 \wedge 1 \le p_2 \le 8 \\ j_1 - 1 = 8q_1 + p_1 - 1 \wedge j_2 - 1 = 8q_2 + p_2 - 1 \\ j_1' - 1 = 8q_1' + p_1 - 1 \wedge j_2' - 1 = 8q_2' + p_2 - 1 \\ q_1 \ge 0 \wedge q_2 \ge 0 \wedge q_1' \ge 0 \wedge q_2' \ge 0 \end{cases} .$$

One may note that, in this example, there is no variable representing the remainder in the division by κ_T (as R in (5.2)) since the block sizes are equal to 1. The result of PIP is that there exists a solution not equal to $\perp$ in the polyhedron defined by:

$$\overline{C}(n,m) = \{\, 1 \le j_1 \le n \wedge 1 \le j_2 \le m \wedge i_1 = j_1 \wedge 1 \le i_2 \le m \wedge j_2 - 1 = 8q \wedge q \ge 0$$

The new parameter q is used to express that $j_2 - 1$ must be a multiple of 8.

5.3 Counting the Elements of the Communication Set

The last stage of our method consists in counting the number of integer vectors in the sub-domains computed by PIP. These parametric sub-domains $D(I,J,P)$ are defined in function of a parameter J representing the subscript of a general template element $T(J)$, in function of a parameter I which represents the subscript of an operation storing a value on $T(J)$ and in function of a vector of program parameters P. We need to compute the number of integer vectors in $D(I,J,P)$ in function of the program parameters. Fortunately, Loechner and Wilde have extended the polyhedron library to include a function able to count the number of integer vectors in a parametric polyhedron (see [4]). Like PIP, this function splits the parameter space in sub-domains on which the result can be given by a parametric expression. Hence, to apply relation (6) one has to implement an addition on Quasts.

For the example `MatInit` the final result (in the context $n \ge 1$ and $m \ge 1$) is

$$\text{Count}(C(n,m)) - \text{Count}(\overline{C}(n,m)) = n.m \left(\frac{7m}{8} - \left[0, \frac{7}{8}, \frac{3}{4}, \frac{5}{8}, \frac{1}{2}, \frac{3}{8}, \frac{1}{4}, \frac{1}{8} \right]_m \right)$$

The brackets on the previous expression denote a periodic number: if we denote by v the vector $(0, \frac{7}{8}, \frac{3}{4}, \frac{5}{8}, \frac{1}{2}, \frac{3}{8}, \frac{1}{4}, \frac{1}{8})$, the value of the periodic number is $v_{m\%8}$.

When the parameter m is a multiple of 8 we have the expected result of $\frac{7n.m^2}{8}$ atomic communications at the template level.

For a detailled description of how the pre-evaluation can be done in an automatic way take a look at the report available at the URL
`ftp://ftp.lifl.fr/pub/reports/AS-publi/an98/as-182.ps.gz`

6 Conclusion

Data partitioning is a major performance factor in HPF programs. To help the programmer design a good data distribution strategy, we have studied the evaluation of the communication cost of a program during the writing of this program.

We have presented here a method to compute the communication volume of a HPF program. This method is based on the polyhedral model. So, we are able to handle loop nests with affine loop bounds and affine array access functions. Our method is parameterized and machine independent. Indeed all is done at the language level. An implementation is done using the polyhedral library and the PIP software.

Ongoing work includes extending this method to a larger class of programs and adding compiler optimizations in the model. The last point is quite important since the pure counting of elements exchanged is only one of the factors in the actual communication costs. We will have to recognize special communications patterns as broadcasts which can be implemented more efficiently than general communications.

We are also integrating this method in the HPF-builder tool [6].

References

1. Jennifer M. Anderson and Monica S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. *ACM Sigplan Notices*, 28(6):112–125, June 1993.
2. Vincent Bouchitte, Pierre Boulet, Alain Darte, and Yves Robert. Evaluating array expressions on massively parallel machines with communication/computation overlap. *International Journal of Supercomputer Applications and High Performance Computing*, 9(3):205–219, 1995.
3. S. Chatterjee, J. R. Gilbert, R. S. Schreiber, and S.-H. Tseng. Automatic array alignment in data-parallel programs. In ACM Press, editor, *Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 16–28, Charleston, South Carolina, January 1993.
4. Philippe Clauss, Vincent Loechner, and Doran Wilde. Deriving formulae to count solutions to parameterized linear systems using ehrhart polynomials: Applications to the analysis of nested-loop programs. Technical Report RR 97-05, ICPS, apr 1997.
5. Fabien Coelho. *Contributions to HPF Compilation.* PhD thesis, Ecole des mines de Paris, October 1996.
6. Jean-Luc Dekeyser and Christian Lefebvre. Hpf-builder: A visual environment to transform fortran 90 codes to hpf. *International Journal of Supercomputing Applications and High Performance Computing*, 11(2):95–102, 1997.
7. Thomas Fahringer. Compile-time estimation of communication costs for data parallel programs. *Journal of Parallel and Distributed Computing*, 39(1):46–65, November 1996.
8. Paul Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22:243–268, September 1988.

9. Paul Feautrier. Towards automatic distribution. *Parallel Processing Letters*, 4(3):233–244, 1994.
10. M. Gupta. *Automatic Data Partitioning on Distributed Memory Multicomputers.* PhD thesis, College of Engineering, University of Illinois at Urbana-Champaign, September 1992.
11. Manish Gupta and Prithviraj Banerjee. Compile-time estimation of communication costs of programs. *Journal of Programming Languages*, 2(3):191–225, September 1994.
12. Ken Kennedy and Ulrich Kremer. Automatic data layout for High Performance Fortran. In Sidney Karin, editor, *Proceedings of the 1995 ACM/IEEE Supercomputing Conference, December 3–8, 1995, San Diego Convention Center, San Diego, CA, USA*, New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1995. ACM Press and IEEE Computer Society Press.
13. Kathleen Knobe, Joan D. Lukas, and Guy L. Steele. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8:102–118, 1990.
14. Jingke Li and Marina Chen. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. In *Frontiers 90: The 3rd Symposium on the Frontiers of Massively Parallel Computation*, College Park, MD, October 1990.
15. S. Wholey. Automatic data mapping for distributed-memory parallel computers. In ACM, editor, *Conference proceedings / 1992 International Conference on Supercomputing, July 19–23, 1992, Washington, DC*, INTERNATIONAL CONFERENCE ON SUPERCOMPUTING 1992; 6th, pages 25–34, New York, NY 10036, USA, 1992. ACM Press.
16. Doran Wilde. A library for doing polyhedral operations. Master's thesis, Oregon State University, Corvallis, Oregon, dec 1993.