Interprocedural Array Alignment Analysis*

Erwin Laure[1] **Barbara Chapman**[2]

[1]Institute for Software Technology and Parallel Systems
[2]VCPC, European Centre for Parallel Computing at Vienna
University of Vienna, Liechtensteinstrasse 22, A-1090 Vienna, Austria
[1]erwin@par.univie.ac.at [2]Barbara.Chapman@vcpc.univie.ac.at

Abstract. The specification of efficient data distribution schemes is one of the major tasks in programming DMMPs with data parallel languages. Although there are no optimal strategies for generating such data distributions, several heuristics have been developed to provide some support to the user. We presented an overview of an automatic alignment analysis tool elsewhere, which is able to automatically generate alignment proposals for the arrays accessed in a procedure and thus simplifies the data distribution problem. In this paper we extend our previous work to interprocedural analysis taking into account dynamic realignment. This feature is essential for applying alignment analysis to real programs.

1 Introduction

When migrating applications to distributed memory architectures, data locality is crucial for performance. Under the data parallel programming paradigm, the user must select a distribution of the program's data to the target machine which ensures good data locality and balances the work load. High level languages such as HPF[5] may use this information directly to generate code. But it is hard to determine an appropriate data distribution. One of the major approaches to this task consists of detecting suitable alignments between dimensions of arrays in the user code in order to reduce communication cost due to array cross-references.

The specification of appropriate alignments also alleviates the task of specifying data distributions in the sense that the aligned array inherits its distribution from the source array.

In [12] we presented a tool performing intraprocedural alignment analysis automatically within the framework of the VFCS compiler. This tool is able to solve both the inter- and intradimensional problem using a common problem model and efficient heuristic algorithms.

In this paper we extend our previous work in order to handle interprocedural problems allowing dynamic solutions. The capability to manage interprocedural problems is one of the key requirements to make the tool applicable for real programs. Unfortunately, the problem of dynamic alignment was proved to be NP-complete by Kremer [10]. We therefore propose to make use of an heuristic algorithm that works on an internal problem representation (the *Alignment Call Graph*) and uses greedy techniques that make the algorithm applicable to real world programs. Moreover, our algorithm minimizes the costs that are associated with realignment by carefully choosing which arrays have to be realigned. A simple example is presented in Section 5 that illustrates the main benefits of our heuristics.

* The work described in this paper was supported by the Austrian Science Fund FWF (SFB F011 "AURORA") and the European Union (ESPRIT project 23502 "FITS").

Related Work The pioneering work in the field of automatic alignment and, subsequently, automatic data distribution was done by Li and Chen [14] and Manish Gupta [8], whose results strongly influenced our analysis. Instead of using heuristics to solve the alignment problem Kremer [9] proposed to use 0/1 integer programming techniques. The approach of Garcia et al. [6, 7] introduces a new framework combining both alignment and data distribution analysis. Both approaches provide dynamic solutions but are restricted to intraprocedural analysis. Other researchers such as [4, 15, 3] deal with the alignment problem by solving matrix equations based upon the array access patterns. This approach seems to be more restricted than ours and its execution can moreover be very time consuming, thus preventing the analysis of larger applications. Chatterjee et al. [2] have developed algorithms in the field of dynamic programming. Ning et al. proposed an algorithm for dynamic alignment within the context of HPC [15]. Although their work is restricted to dynamic intraprocedural solutions, it has inspired our Procedure Clustering Algorithm. In some of the research mentioned above dynamic solutions are considered, however, they are limited to intraprocedural analysis. Ayguade et al. presented in [1] a research tool which is able to generate interprocedural solutions. However, exhaustive search techniques are applied thus preventing the analysis of bigger programs.

The remainder of this paper is organized as follows: The basic concepts used for alignment analysis are introduced in Section 2. Section 3 gives a brief review of our previous work on intraprocedural analysis. The problem of interprocedural analysis is handled in Section 4. Finally we present an application example showing the main features of our interprocedural algorithm and we end with some remarks on this work.

2 Basic Concepts

Intraprocedural Analysis Our approach to intraprocedural alignment analysis is to analyze array references of statements inside the loops of the examined program unit (procedure).

Let L denote a loop and S an assignment statement within L. Let furthermore a and b denote references to arbitrarily dimensioned arrays A and B in S.

Information useful for the alignment problem can be gathered by analyzing each pair of these subscript expressions.

For modeling the alignment problem we use the *Component Alignment Graph (CAG)* framework first introduced by Li and Chen [14]. The CAG is an undirected weighted graph. A node represents a dimension of an array which is referenced in the examined unit. If the alignment analysis detects a preference for aligning two array dimensions, the corresponding pair of nodes in the CAG are connected by an undirected weighted edge, called *alignment edge (e)*. An alignment edge is attributed with information about both inter- and intradimensional preferences (cf. Section 3). The solution of the alignment problem is represented via the *Alignment Sets ($\mathcal{S}$)*, also introduced by Li and Chen [14]. $\mathcal{S} = \mathcal{S}_1, ..., \mathcal{S}_d (d \geq 1)$, where d is the number of dimensions of an array with the highest dimensionality in the analyzed unit. All nodes of the CAG are matched to $\mathcal{S}$ such that all nodes belonging to $\mathcal{S}_i \in \mathcal{S}$ express a preference for alignment (cf. Section 3).

In the remainder we use the following notation: a node of the CAG is denoted by the capitalized name of the array, subscripted with the dimension, and

alignment edges are denoted by e(node,node); $e(A_1, B_2)$ therefore represents the edge between the first dimension of array A and the second one of array B.

Interprocedural Analysis Let P denote a program. P is a set of $n \geq 1$ procedures denoted by p corresponding to Fortran program units. The *call graph* G of P is a directed graph $G = (N, E)$, where there is a one-to-one correspondence between N and P, and $(p, q) \in E$ iff procedure p contains a call whose execution may result in the direct activation of procedure q. Procedure p is called the *caller* and q the *callee*. The set of all calls occurring in the program text of p resulting in a direct activation of q is denoted by *calls(p,q)*. We assume that G is acyclic.

Let *formal(q)* denote the set of formal parameters of q and *global(q)* the set of global variables accessible in q. If during a call an array A is bound to an formal array $B \in formal(q)$ we denote this binding by $b\langle A, B\rangle$.

Let G be the call graph of P. The **Alignment Call Graph (ACG)** is an exact copy of G holding additional information for alignment analysis.

Each procedure $p \in ACG$ is linked to its CAG and Alignment Sets computed during the intraprocedural phase. Moreover, each edge of the ACG is weighted with the frequency measure of how often the call occurs. Note, that for sake of exact analysis we have to distinguish between the *absolute call frequency (acf)*, which is the overall frequency measured during the program execution, and the *relative call frequency (rfc)*, which is the average call frequency during one instance of the caller.

All further actions during the interprocedural analysis use the ACG only, and do not affect the program's call graph G. Thus, when applying procedure clustering (cf. Section 4.2) a node in the ACG may refer to a set of nodes in G.

3 Intraprocedural Analysis

The Intraprocedural Analysis is the kernel of our tool. For each program unit the CAG is constructed and the (intraprocedural) alignment problem is solved using a heuristic algorithm. The solution is represented in the form of Alignment Sets. In this section we give a brief overview of the problem formulations and the solution algorithms. A comprehensive discussion can be found in [12].

The Intraprocedural Alignment Problem The intraprocedural alignment problem is to identify *alignment preferences* between two array dimensions by analyzing their reference patterns. For each pair of array dimensions which have an alignment preference an *alignment edge* is constructed in the CAG (cf. Section 2). This edge is annotated with both, inter- and intradimensional information:

The interdimensional information is a *weight* which reflects the communication effort arising when the two array dimensions are not aligned. Since we do not know anything about the actual distributions and processor numbers at analysis time, we have to apply strong worst case assumptions reflecting the compiler's communication optimization and some machine specific properties. We use Equation (1) for computing the weight where the necessary data is supplied by a sequential profiler:

$$\omega(e) = (tt * ca + st) * cf \tag{1}$$

$\omega(e)$ symbolizes the weight of edge e, tt the transfer time, ca the amount of communication, st the startup time and finally cf the communication frequency.

For modeling intradimensional preferences we defined the concept of *locality* consisting of the *locality type* and the *locality relation*. The locality relation consists of the tuple (*multiplicator, constant offset*), where the multiplicator denotes strides within the access patterns and the constant offset a possible shift. The locality type gives concise information about the locality relation and can be *perfect, constant* or *invariant*.

The locality can give useful hints for choosing the appropriate alignment functions[2] and also helps us handling multiple occurrences of the same alignment preferences (this means that references to the same arrays are alignment preferring in distinct parts of the code). Furthermore, competing intradimensional alignment preferences can be detected.

Solving the Intraprocedural Problem As proposed by Li and Chen, the Alignment Problem can be solved by matching each node of the CAG to a number of sets (the number being equal to the highest dimensionality of all arrays handled), with the restriction that no two nodes of the same array may belong to the same set, while minimizing the weight of all edges between nodes belonging to different sets. These sets are called *Alignment Sets*. The goals of this procedure are obvious: when matching nodes (via Alignment Sets), they should be distributed in the same manner, certainly according to their intradimensional preferences, thus avoiding communication effort due to cross references. Only those edges between nodes in different sets symbolize unavoidable communication.

Unfortunately the matching problem as stated above has been proved to be NP-complete in [14], however Li and Chen presented an heuristic algorithm which reduced the problem to a bipartite-graph matching problem. We make use of an adapted version of this algorithm that takes also intradimensional preferences into account. A detailed description can be found in [12].

The Alignment Sets are arbitrarily numbered in order to distinguish them and can be represented as an ordered list. All nodes belonging to the same set should be aligned with each other, where the precise alignments will be based upon the intradimensional information with which each node is annotated.

The information contained in an Alignment Set can be directly used to generate alignment directives in HPF-like languages. When generating the alignment directives, we consider an array with highest dimensionality to be the *alignment source array* and align all other arrays with it. Subsequently, the user only has to specify the distributions for this single alignment source array.

The CAG and Alignment Sets are stored and linked to the respective node in the ACG for the interprocedural analysis.

4 Interprocedural Analysis

In this section we extend our model to interprocedural analysis in order to enable its application on full programs. Within this context the possibility of changes in the alignment schemes via realignment is worth to be considered.

Interprocedural alignment analysis which considers redistribution between procedure boundaries can be seen as one kind of *dynamic alignment*, where realignment is allowed at specific points in the program. In our approach we

[2] The main alignment functions under consideration are shift, stride and reflection.

currently do not handle intraprocedural phases and allow realignment only at procedure boundaries. This simplifies the problem and allows us to study the effectiveness of our approach. Nevertheless, our model can be expanded to handle intraprocedural phases as well and this will be considered in the future. Our approach assumes that the caller's alignment scheme is restored after execution of the return of control to it.

4.1 Problem Formulation

The intraprocedural analysis phase results in an ACG (cf. Section 2) where each unit is attributed with the CAG and the Alignment Sets, representing the problem model and its static solution, respectively. The goal of the interprocedural analysis is to combine as many units as possible into unit clusters - where within such a cluster the alignment scheme stays invariant - so that the alignment cost within a cluster and the realignment cost between clusters are minimized. The alignment cost and the realignment cost are defined as follows:

Alignment Cost The cost of an alignment solution can be defined as the accumulated weight of all alignment edges connecting nodes in different Alignment Sets plus the weight of all edges with a locality relation which is not observed. In the following we denote the cost of solution $\mathcal{S}$ by $ac(\mathcal{S})$. The full mathematical formulation can be found in [13].

Realignment Cost For a call $c \in calls(p, q)$ the realignment cost can be computed by comparing the alignment solutions of program unit p with those of program unit q. If we find differing solutions for an array, we have to assume that the array must be redistributed. Note that we only compute the cost for one execution of a call, which means that the result must be multiplied with the call frequencies to get a comparable measure (cf. Section 4.2). We make worst case assumptions, i.e. that the whole array has to be communicated.

The major problem within this context is the specification of the arrays to be realigned. Recall that alignment is a symmetric relation between two arrays, thus in the case of differing alignment schemes we could realign either array.

Let $\mathcal{S}_p$ denote the Alignment Sets of p and $\mathcal{S}_q$ the Alignment Sets of q, respectively. We model the realignment problem in a manner similar to the interdimensional alignment problem: a bipartite graph is constructed whose columns are composed of $\mathcal{S}_p$ and $\mathcal{S}_q$, respectively. There is a node for each Alignment Set.

Let $A \in \mathcal{S}_p$ and $B \in \mathcal{S}_q$ be two arrays bound during the call $(b\langle A, B\rangle)$. Each pair of nodes of the bipartite graph whose associated Alignment Sets contain the same dimensions of A and B are connected with an undirected, weighted edge. The same holds for arrays within the intersection of *global(p)* and *global(q)*.

The edge is weighted with the cost of realigning array A (array realignment cost *arc*) which is according to Equation (1):

$$arc(A) = (tt * \text{arraysize}(A) + st) * 2 \tag{2}$$

Note that the realignment cost is multiplied by 2, since realignment occurs on the call and on the return.

The solution of the realignment problem can now be formulated as a matching problem for this bipartite graph that seeks the minimum accumulated *arc* for every call $c \in calls(p, q)$. The output are the total realignment costs between

```
procedure_clustering_algorithm(ACG);
{unmark_edges(ACG);
while (∃ unmarked edge) {
 /* take the two nodes (p,q) connected by the unmarked edge with the
  * highest absolute call frequency and their CAGs (CAG1, CAG2) as
  * well as their solutions (Sets1,Sets2) and merge the CAGs */
 CAGm  = merge_CAG(CAG1,CAG2);
 Setsm = heuristic_algorithm(CAGm);
 /* compute the cost */
 cost_merged = ac(Setsm);
 cost_unmerged=ac(Sets1)+∑_{c∈calls(p,q)}(rcf(c)*(ac(Sets2)+rc(Sets1,Sets2,c)));
 if (!(cost_merged > cost_unmerged))
    Update_ACG(ACG,node1,node2,CAGm,Setsm);
 else
   /* both nodes retain their CAGs and Solutions */
 mark_edge(); }}
```

Fig. 1. Procedure Clustering Algorithm

$\mathcal{S}_p$ and $\mathcal{S}_q$ for every call $c \in calls(p,q)$, denoted by $rc(\mathcal{S}_p, \mathcal{S}_q, c)$, as well as a list of all arrays that have to be realigned. Note, that this list identifies those arrays that lead to minimized realignment cost. Again, the full mathematical formulation can be found in [13].

The cost model is kept quite simple (e.g. network contentions that are likely to occur on redistribution are ignored), but it can easily be substituted with a more accurate one if required. Moreover, our analysis is intended to become a real automatic data distribution tool in future, which combines alignment and distribution analysis. An extension of our model towards this analysis can be achieved by substituting the cost model with a more appropriate one reflecting also the distributions.

4.2 The Procedure Clustering Algorithm

Since finding an optimal solution for dynamic alignment schemes was proved to be NP-complete by Kremer [10] we use a greedy algorithm to solve the procedure clustering problem heuristically. Our algorithm was inspired by the *Clustering Algorithm* developed by Ning et al. [15]. They developed their algorithm for the clustering of program phases defined as loop nests and did not handle interprocedural problems. We found that this algorithm can be successfully extended and modified for interprocedural problems.

An outline of our *Procedure Clustering Algorithm (PCA)* can be found in Figure 1. The algorithm needs one pass over the ACG and tries to combine those procedures which are connected by a maximum weighted edge (absolute call frequency) first. It is important to analyze those procedures first, since possible realignment cost will be incurred at each instance of the call.

The next step is the merging of two CAGs, which is described below. The heuristic algorithm (cf. Section 3) is used to compute the solution of the merged CAG, and the cost of this solution is compared with the cost of the original solutions plus the realignment cost. Note that the alignment cost of the callee and the realignment cost are multiplied by the relative call frequency.

Depending on which result is better, the two nodes for the procedures either remain separate and retain their original CAGs and solutions or they are merged and the ACG is updated accordingly.

Merging two CAGs The central task of the PCA is the merging of two CAGs along a procedure call. This merging can be seen as a kind of inline expansion [16] on the level of the CAG. Due to Fortran memory management, several problems arise. While we can handle common blocks and parameter passing, some specific Fortran features such as equivalence and reshaping are not considered. Note that the locality relation plays an important role in this phase, since it is used to model changes in the intradimensional alignment during a procedure call. The locality of the callee nodes is updated with the locality of the equivalent caller nodes using an adapted locality propagation algorithm (the algorithm can be found in [11]).

Another important issue is the weight with which the edges of both the original CAGs and the merged CAG are annotated. Remember that the intraprocedural step uses absolute frequencies for the weight calculation. However, if we are looking at two procedures isolated from the program, we should not take into account calls from other procedures. Thus, we use relative weights ($r\omega$) for all CAG edges which indicate the weight for one instance of the procedure.

The merging algorithm first generates a copy of the caller's CAG, in which the callee's CAG is inserted. For each call the dummy arguments are substituted with the actual ones according to their bindings and their locality relations are propagated. The weights of the callee's edges are multiplied with the relative frequency of the call and all nodes and edges are inserted into the merged CAG using the concepts of our intraprocedural analysis, especially where multiple alignment occurrences and intradimensional conflicts are concerned. The formulation of the algorithm can be found in [13].

Note that both the caller and the callee in this context refer to a node in the ACG, hence they may represent more than one procedure.

Updating the ACG If the PCA decides to merge the alignment schemes of two procedures we have to update the ACG. The ACG node of the callee is merged into the node of the caller which now represents a cluster of procedures. All incoming ACG edges of the callee node are redirected to its new node as well as all the leaving ones; edges that are now within a node can be deleted. The relative frequencies of all the leaving edges have to be updated (multiplied) with the frequency of how often the callee is called directly by the caller. Finally, the original node of the callee is deleted from the ACG.

After the execution of the PCA the ACG holds one node for each cluster annotated with the CAG and the Alignment Sets, and a pointer to G for each procedure which is a member of the cluster. The alignment schemes stored in the ACG node can easily be propagated to the procedure's source code and realignment is only necessary along calls between procedures belonging to different clusters, where the optimal choice of the arrays to be realigned is specified.

Note that all merging actions only affect the ACG, not G. Furthermore our algorithm requires a procedure to have only one alignment scheme. Hence procedure cloning is not allowed.

In the worst case our algorithm needs to merge and analyze the CAGs and solutions of every pair of procedures connected by an edge in ACG. Thus, the complexity is directly proportional with the number of edges in the ACG. Fortunately, in the typical case it will not be necessary to analyze each edge on its own since we combine nodes in the ACG if we decide to cluster them. In

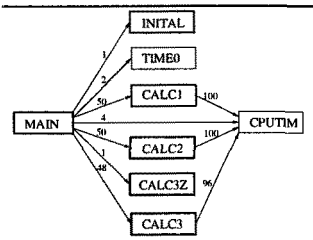


Fig. 2. ACG for shallow

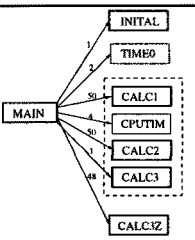


Fig. 3. Partially merged (1)

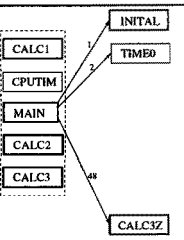


Fig. 4. Partially merged (2)

the following section this is motivated by an application example. Although it is simple we believe that this is typical over a large fraction of applications.

5 Application Examples

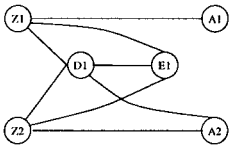
In this section we illustrate our interprocedural algorithm by considering the *shallow* benchmark from the xHPF benchmark set. This code is a straightforward weather prediction benchmark program that uses finite differencing.

The ACG of *shallow* can be seen in Figure 2. In this diagram, the boxes representing those subroutines in which alignment preferences can be found have been emboldened. Note that the time measurement procedures do not provide any information about alignment, as can be expected. The edges of the call graph are annotated with the absolute call frequency (acf).

First, the PCA tries to merge procedure `CPUTIM` with `CALC1`, `CALC2`, and `CALC3`. This merging is trivial, since `CPUTIM` has no CAG at all and the other procedures are not mutually connected. We can therefore combine these four procedures into one cluster (cf. Figure 3). Note that the sinks of some edges from procedure `MAIN` are now within the cluster. Thus, in the next step, which is the merging of `MAIN` and `CALC1`, all the other calls from `MAIN` to any procedure belonging to the same cluster as `CALC1` are considered simultaneously.

At this step, we find that there are no conflicts in the alignment schemes of all procedures involved and can therefore decide to merge `MAIN` with the cluster also. The result can be seen in Figure 4. It remains to merge the procedures `CALC3Z`, `TIME0`, and `INITAL`, which can indeed also be inserted into the cluster in this program. Overall, the main loop of the PCA is executed 7 times in this example, although there are 10 edges in the program's call graph. This shows that in real programs not every edge of the call graph has to be analyzed isolated which speeds up our algorithm significantly. All the results of our analysis as well as code excerpts of shallow can be found in [13].

Another example is the Tred2 routine from the Eispack library. It reduces a symmetric matrix to a tridiagonal matrix using and accumulating orthogonal similarity transformations. In Figure 5 we show the inter-procedural analysis results of tred2 and performance measurements on the Meiko CS2 with different alignment and distribution settings. The size of the matrix was 160^2. It is worth noting that our proposed alignment performs best in all but one of the test cases. Moreover, with our alignment and an appropriate data distribution the optimal performance can be achieved.



Set 1
Z_1-Perfect Locality
A_2-Perfect Locality
Set 2
Z_2-Perfect Locality
A_2-Perfect Locality
D_1-Perfect Locality
E_1-Constant Locality(1;0,1)

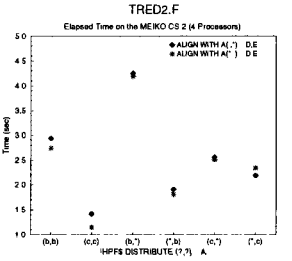


Fig. 5. The Tred2 Example

6 Conclusions

In our previous work, we have focused on advanced models for alignment analysis which combine inter- and intradimensional alignment preferences. This combination turned out to have an important impact on the solution's accuracy. However, the analysis was so far restricted to intraprocedural problems.

In this paper, we have extended our analysis to interprocedural problems. A strategy for dealing with array alignment across procedure boundaries is naturally a key requirement for any system which is able to handle real applications.

We have shown how to combine the intraprocedural models for the alignment problem taking into account both inter- and intradimensional preferences. A greedy algorithm was proposed to solve the problem of clustering several procedures, where the alignment scheme within each cluster will remain invariant. The main step of this *Procedure Clustering Algorithm (PCA)* was the merging of the alignment models (CAG) of two procedures. Cost models are introduced to measure the cost of a candidate alignment scheme as well as realignment cost at procedure boundaries. In formulating our methods, we considered the needs of future extensions to perform automatic data distribution. These extensions will use essentially the same methods, with additional cost models for available parallelism and array redistribution.

Although we do not handle dynamic intraprocedural analysis at the moment, our models can be successfully applied for this kind of problem, too. The main issues within this context include the definition of computational phases, within which the alignment scheme stays invariant, as well as more advanced flow analysis for the reaching distribution problem as needed for the estimation of realignment cost.

In our future work, we plan to extend our models with parallelism constraints in order to evolve towards automatic data distribution. We will study the proposals of Garcia et al. [6, 7] for including information on parallelism in the alignment model and will consider replacing the current heuristic algorithm for solving the intraprocedural alignment problem with integer programming techniques, as proposed by Kremer and Garcia et al. However, we have to carefully study the time complexity of these techniques within our framework.

7 Acknowledgments

The authors would like to thank Markus Egg from the VCPC for implementing the GUI within the tool ANALYST.

References

1. E. Ayguade, J. Garcia, M. Girones, M. Luz Grande, and J. Labarta. DDT: A Research Tool for Automatic Data Distribution in HPF. Technical Report UPC-CEPBA-1995-20, Polytechnic University of Catalunya, Barcelona, Spain, 1995.
2. S. Chatterjee, J.R. Gilbert, R. Schreiber, and S. Teng. Automatic Array Alignment in Data-Parallel Programs. In *Proceedings of the Twentieth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, Charleston, January 1993.
3. C. G. Diderich and M. Gengler. Solving the constant-degree parallelism alignment problem. In *Proc. of EuroPar '96*, August 1996.
4. M. Dion and Y. Robert. Mapping Affine Loop Nests: New Results. Technical Report Nr. 94-30, Laboratoire de l'Informatique du Parallèlisme, Ecole Normale Supèrieure de Lyon, November 1994.
5. High Performance Fortran Forum. High Performance Fortran Language Specification Version 2.0, January 1997.
6. J. Garcia, E. Ayguadè, and J. Labarta. A Novel Approach Towards Automatic Data Distribution. In *Workshop on Automatic Data Layout and Performance Prediction*, Houston, April 1995. CRPC, Rice University.
7. J. Garcia, E.Ayguadè, and J. Labarta. Dynamic Data Distribution with Control Flow Analysis. In *Proceedings Supercomputing 96*, Pittsburgh, PA, November 1996.
8. M. Gupta. *Automatic Data Partitioning on Distributed Memory Mulitcomputers.* PhD thesis, Coordinated Science Lab, University of Illinois at Urbana-Champaign, 1992.
9. K. Kennedy and U. Kremer. Automatic Data Layout for High Performance Fortran. In *Proceedings of Supercomputing 95*, San Diego,CA, December 1995.
10. U. Kremer. NP-completeness of Dynamic Remapping. In *Fourth International Workshop on Compilers for Parallel Computers.* Delft University of Technology, Dezember 1993.
11. E. Laure and B. Chapman. Alignment Analysis within the VFCS - A pragmatic Method for Supporting Data Distribution. TR 96-2, Institute for Software Technology and Parallel Systems, University of Vienna, October 1996.
12. E. Laure and B. Chapman. Combining Inter- and Intradimensional Alignment Analysis to Support Data Distribution. In *Proceedings HPCN Europe 1997*, Lecture Notes in Computer Science 1225, pages 830–839. Springer, April 1997.
13. E. Laure and B. Chapman. Interprocedural Array Alignment Analysis. TR 97-7, Institute for Software Technology and Parallel Systems, University of Vienna, 1997.
14. J. Li and M. Chen. Index Domain Alignment: Minimizing Cost of Cross-Referencing Between Distributed Arrays. Technical Report YALEU/DCS/TR-72, Yale University, November 1989.
15. Qi Ning, V. Van Dongen, and G.R. Gao. Automatic Data and Computation Decomposition for Distributed Memory Machines. In *28th Hawaii International Conference on System Science Wailea*, Maui, Hawaii, January 1995.
16. H.P. Zima and B.M. Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press Frontier Series. ACM, Addison-Wesely, 1990.