Interprocedural Array Alignment Analysis

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

[1]Institute for Software Technology and Parallel Systems
University of Vienna - Austria
erwin@par.univie.ac.at

[2]VCPC, European Centre for Parallel Computing at Vienna
University of Vienna - Austria
Barbara.Chapman@vcpc.univie.ac.at

Abstract

The specification of efficient data distribution schemes is one of the major tasks in programming DMMPs with state of the art 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. Alignment analysis, for instance, is able to help find a good distribution scheme and is furthermore a useful prerequisite for automatic data distribution tools. We presented an overview of an automatic alignment analysis tool within the framework of VFCS 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. We present a heuristic algorithm for interprocedural dynamic alignment analysis as well as our cost models based upon sequential profiling data. Furthermore we show that the locality concept introduced elsewhere for intradimensional alignment preferences proved to be applicable and helpful also in this context.

1 Introduction

The development of high level data parallel languages has significantly alleviated the task of programming state of the art distributed memory multi processor machines (DMMP) under the single program multiple data (SPMD) paradigm. However, these languages require the user to specify the data and sometimes the work distribution; yet both of these have a profound influence on the execution time, since they greatly affect the load balance as well as the amount and pattern of data exchange between processors.

Alignment analysis is a major source of help for selecting data distributions, either automatically or by the user. It deals with the problem of reducing communication cost due to array cross-references by detecting which dimensions of arrays are generally used together. When two array dimensions are aligned, the aligned array inherits its distribution from the source array via alignment, thus alleviating the task of specifying data distribution. Alignment primitives are included in state of the art data parallel languages like HPF [10] or Vienna Fortran 90 [3].

In [17, 18] we presented a tool performing intraprocedural alignment analysis automatically within the framework of the VFCS compiler [4]. 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. We introduce the *Alignment Call Graph* for modeling the interprocedural problem and a heuristic solution algorithm, the *Procedure Clustering Algorithm*, performing a kind of "inlining" at the level of the intraprocedural model. The problem model as well as the solution heuristics are designed with respect to a future extension towards automatic data distribution.

At the moment our algorithm is applicable to structured Fortran programs and does not deal with details such as work arrays, reshaping, and equivalence. Although we do not consider dynamic intraprocedural solutions so far (we allow realignment only at procedure boundaries), our models and algorithms can be extended in this direction.

Related Work The field of automatic alignment and, subsequently, automatic data distribution became very popular during recent years.

Li and Chen at Yale University played a fundamental role in introducing the concept of alignment analysis [19] in the framework of the functional language *Crystal* [6]. They introduced the CAG framework and developed a heuristic algorithm for solving the alignment problem. A very simple scheme is used for weighing alignment preferences - they are weighted with 1 or ε depending on their overall importance.

Manish Gupta extended the work of Li and Chen and uses the CAG framework inside a tool for automatic data distribution [13]. The tool automatically generates a data distribution scheme performing four distinct phases: the *alignment phase*, which uses the CAG but with a more sophisticated manner of weighing the edges (possible communication cost is estimated using strong and time consuming heuristics); the *block vs. cyclic phase* decides whether to apply a block or cyclic distribution strategy; the *block size phase* tries to find the optimal block size for cyclic strategies; and finally the *numbers of processors phase* decides how many processors are to be attached to each grid dimension. To estimate a goodness measure for each phase Gupta uses a predefined standard distribution function for comparison.

Instead of using heuristics to solve the alignment problem Kremer [14, 16] proposed to use 0/1 integer programming techniques. Their approach provides dynamic intraprocedural solutions of the alignment problem.

The approach of Garcia et al. [11] introduces a new framework combining both alignment and data distribution analysis. Recently their work was extended to dynamic intraprocedural analysis taking control flow information into account [12], however their tool is currently restricted to intraprocedural analysis and one dimensional solutions. We plan to investigate their proposal for a future extension of our work towards automatic data distribution.

Other researchers such as [8, 20, 7] 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.

Ning et al. [20] proposed an algorithm for dynamic alignment within the context of HPC [21]. Although their work is restricted to dynamic intraprocedural solutions, it has inspired our Procedure Clustering Algorithm.

Chatterjee et al. [5] have developed algorithms in the field of dynamic programming.

Although in some of the research mentioned above dynamic solutions are considered, they are limited to intraprocedural analysis. Ayguade et al. presented in [2] a research tool able to generate interprocedural solutions. However, these are obtained by combining the intraprocedural solutions, rather than their models, and the interprocedural solution is generated by applying exhaustive search techniques.

In this paper we show that efficient interprocedural analysis is possible, taking into account all the features of the intraprocedural model by combining the problem's intraprocedural models.

The remainder of this paper is organized as follows: Section 2 describes the global structure of the Alignment Analysis Tool within the framework of the VFCS. The basic concepts used for alignment analysis are introduced in Section 3. Section 4 gives a brief review of our previous work on intraprocedural analysis including our weight models for interdimensional constraints and our models for intradimensional constraints. The problem of interprocedural analysis is handled in Section 5. Finally we present an application example showing the main features of our interprocedural algorithm and we end with some remarks on this work.

2 The Alignment Tool within the Framework of the VFCS

In this section we give a brief overview of the Alignment Analysis Tool realized within the framework of the VFCS. A more comprehensive discussion about alignment analysis in general as well as the integration of the (intraprocedural) tool with the VFCS can be found in [17]. The overall structure of the tool is depicted in Figure 1 and can be divided into 3 parts:

The Frontend performs the initial parsing and compiler analysis, including data dependence analysis, and produces the flow & call graphs and the syntax tree. The internal structure is annotated with data dependence information as well as sequential profile information. The profile data is supplied by the Weight Finder [9], an advanced profiler for sequential Fortran programs, and includes loop parameters, true ratios and execution frequencies of statements. This data is needed to estimate the weight of alignment preferences. Source program transformations such as *scalar forward substitution* and *induction*

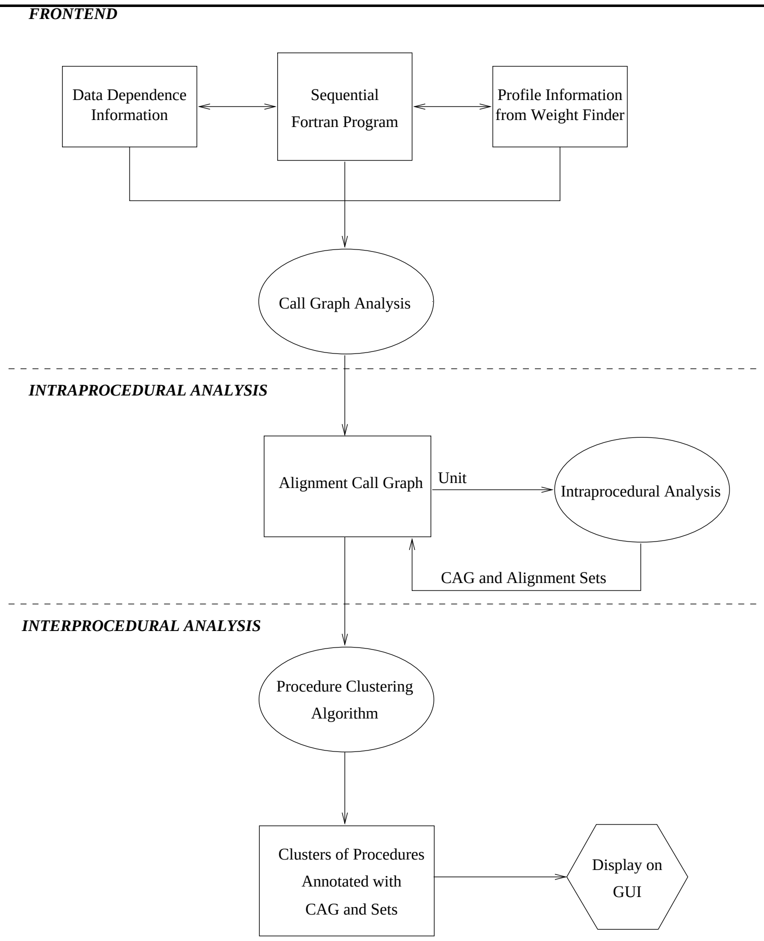

Fig. 1. Overall Structure of the Alignment Tool

variable substitution [22] are automatically applied in order to enlarge the number of alignment preferences detected.

The *Alignment Call Graph (ACG)* (cf. Section 3) is constructed. The ACG is a generalization of the program's call graph holding additional information which is needed for the interprocedural alignment analysis.

In the second phase, *Intraprocedural Analysis* (cf. Section 4), the alignment problem is analyzed and solved for each node of the ACG independently. This step consists of two parts: the *interdimensional analysis* focuses on the problem of determining which array dimensions are used together, and *intra-dimensional analysis* handles the problem of the relative position of the array accesses, expressing an interdimensional preference. We model both problems simultaneously within a common graph framework (the *Component Alignment Graph (CAG)*) and solve them using an heuristic algorithm. The solution is represented in the *Alignment Sets* and the ACG node is annotated with both the CAG and the Alignment Sets.

Finally, the third phase, *Interprocedural Analysis*, tries to cluster as many units of the ACG as possible: within a cluster the alignment scheme stays invariant, thus changes in the alignment scheme can only occur between different clusters. For each cluster, a single CAG and the corresponding Alignment Sets are computed and visualized via a graphical user interface. The *Procedure Clustering Algorithm* is described in Section 5.

3 Basic Concepts

3.1 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, S an assignment statement within L, and $L = L_1, L_2, ..., L_n (n \geq 1)$ be the sequence of all do-loops enclosing S, numbered successively from the outermost to the innermost loop. L_j is the loop at *level* j, and $(L_1, ..., L_n)$ the *(loop) nest* enclosing S. S is said to be a statement at level n. Note that the loop nest need not be perfect. $I = (i_1, ..., i_n)$ is the vector of all loop variables in L, where i_j is the loop variable of loop L_j.

Let furthermore a and b denote references to arbitrarily dimensioned arrays A and B in S. Their subscript expression in dimension d is referred to by $x_d(a)$, $x_d(b)$, respectively.

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

For modeling the alignment problem we use the *Component Alignment Graph (CAG)* framework first introduced by Li and Chen [19]. 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 4).

The solution of the alignment problem is represented via the *Alignment Sets* $(\mathcal{S})$, also introduced by Li and Chen [19]. $\mathcal{S} = \mathcal{S}_1, ..., \mathcal{S}_d (d \geq 1)$, 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 4.4).

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, (A_1 represents therefore the first dimension of array A), 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.

3.2 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. pv_q is the *parameter vector* of q. A call $c \in calls(p, q)$ can be represented by $c = q(\boldsymbol{y})$, where $\boldsymbol{y} = (y_1, ..., y_m)(m \geq 0)$ is the argument list passed to q. Each element $f \in formal(q)$ is bound to an element of y via the function $bind_q(pv_q(i)) := \boldsymbol{y}(i)$ for all $i, 1 \leq i \leq |\boldsymbol{y}|$. If an array $A \in \boldsymbol{y}$ is bound to an formal array $B \in pv_q$ we denote this binding by $b\langle A, B\rangle$.

Definition 1. 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.

Definition 2. Let $F(c)$ be the measured sequential execution frequency of a call $c \in calls(p, q)$. The *absolute call frequency (acf)* for the edge $(p, q) \in ACG$ is

$$acf(p, q) = \sum_{c \in calls(p,q)} F(c) \tag{1}$$

□

Definition 3. Let $F(p)$ by the measured sequential execution frequency of procedure p. The *relative call frequency (rcf)* of $c \in calls(p,q)$ is

$$rcf(c) = F(c)/F(p) \tag{2}$$

$\square$

Thus the *rcf* indicates how often a call statement is executed during one instance of the caller.

An edge $(p,q) \in ACG$ is attributed with the $acf(p,q)$ and with the $rcf(c)$ for each $c \in calls(p,q)$.

Note that all further actions during the interprocedural analysis use the ACG only, and do not affect the program's call graph G.

4 Intraprocedural Alignment Analysis

The Intraprocedural Analysis is the kernel of our analysis 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.

The problem to be solved in the intraprocedural phase can be split up into two components: the inter- and the intradimensional alignment problem. In this section we give a brief overview of the problem formulations and the solution algorithms. A comprehensive discussion can be found in [17].

4.1 The Interdimensional Alignment Problem

The main task during alignment analysis is the appropriate definition of *alignment preferences*, i.e. the preference to align two array dimensions in subsequent steps and to provide an adequate measurement which expresses the relative importance of an alignment preference to the overall performance.

As mentioned above information about alignment preferences can be gathered by analyzing array access patterns within loops. For our analysis approach we use and extend the concepts provided by Gupta [13] which are the *variation level* (i.e. the level of the innermost loop changing the value of a subscript expression in the specified dimension) and the *single index* property (this means, that a subscript expression contains one and only one DO-loop index and all the other components are integer constants).

We extend Gupta's model in order to increase the number of detectable alignment preferences. The single index property is modified by allowing invariant components within the subscript expressions. Furthermore the *conformance* of two subscript expressions (i.e. where they are may be exactly the same, or may differ by a constant or invariant offset) is introduced. Of course, conformable subscript expressions also have a preference to be aligned.

With help of these concepts we are now able to define an improved alignment preference compared to those of [19] and [13]. In general, alignment preferences can be found between an array reference on the left-hand-side (lhs) and one on the right-hand-side (rhs) of an assignment statement. However, we introduce also alignment preferences between two array references on the rhs of an assignment statement, if there is no preference with the lhs. This situation often occurs in practice, especially in reduction operations, and helps to improve the overall results.

Definition 4. Let a, b, and c be array references in $S \in L$.
If $x_d(a)$ and $x_t(b)$ either *conform* or are *single index* with the same *variation level* and S is either

- of the form $a = \mathcal{F}(b)$ or
- of the form $c = \mathcal{F}(a,b)$ and neither a nor b are alignment preferring with c,

then dimension d of A and dimension t of B are called *alignment preferring*. $\square$

This definition covers all cases where useful information about alignment preferences can be statically detected.

Weighing the Edges One of the crucial issues in the construction of a CAG is the question of the weight with which its edges are annotated. Since these weights reflect the overall importance of an alignment preference, an appropriate choice is essential for the overall quality of the analysis.

The weight should reflect 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. The necessary data for our computations is supplied by the Weight Finder.

Definition 5. Let e be an arbitrary alignment edge. This edge is annotated with a weight representing a heuristic estimation of the possible communication due to this edge, calculated by:

$$\omega(e) = (tt * ca + st) * cf \tag{3}$$

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

This weight associated with an alignment edge will be used in subsequent steps to evaluate the relative importance of an alignment preference and thus for solving the alignment problem and matching the nodes to the respective sets (cf. 4.4).

4.2 The Intradimensional Alignment Problem

The intradimensional alignment problem is the problem of finding the relative position of array references which are alignment preferring. This information can give useful information about the alignment functions which should be applied when aligning and distributing these two dimensions. The main alignment functions under consideration are shift, stride and reflection. Furthermore, intradimensional preferences have a strong impact on the solution of the interdimensional problem since conflicts in intradimensional preferences might lead to a certain amount of unavoidable communication effort when distributing the arrays. Therefore we model them together within the framework of the CAG.

The Locality To model intradimensional preferences we defined the concept of *locality* consisting of the *locality type* and the *locality relation*. An edge of the CAG is annotated with the locality, thus representing its intradimensional preferences. 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*. Perfect locality indicates that there are no shifts or strides within the access pattern, whereas a constant or invariant type reflects the presence of a shift or stride with a known (constant) or unknown (invariant) size.

The locality can give useful hints for choosing the appropriate alignment and distribution functions 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.

4.3 Interactions Between the Inter- and Intradimensional Problem

The locality is used to detect conflicts in the intradimensional alignment scheme. Such conflicts can occur when more than one pair of array references express an interdimensional alignment preference. Their careful observance is very important for the quality of the obtained results as we showed in [18].

We defined the case when the localities of two alignment preferences have no conflicts and refer to them as *similar localities*. In this definition we allowed a difference in the constant offset in order to include stencil operations. Nevertheless, the difference should not be more than ± 5.

Another aspect for the intradimensional model is the question whether the values referenced multiple times might have changed or not, thus if (potentially) a renewed communication is necessary or not. Since an edge of the CAG is not connected to a specific statement, we must assume - without the use of more sophisticated analysis - the worst case when detecting a second occurrence of this preference. Only when an array is identically referenced multiple times in the same statement, we can immediately infer that no renewed communication will be necessary.

Suppose $\mathcal{A}$ and $\mathcal{B}$ are two occurrences of an alignment preference between the same pair of array dimensions. One of the four following cases will hold and they are handled as described:

Case 1 $\mathcal{A}$ and $\mathcal{B}$ belong to the same statement and have a similar locality:
$\mathcal{B}$ is redundant information and can be ignored because any potential communication is modeled by $\mathcal{A}$.

Case 2 $\mathcal{A}$ and $\mathcal{B}$ belong to the same statement and have a different locality:
In this case an intradimensional conflict arises. We model this by generating a new edge in the CAG for preference $\mathcal{B}$ and annotate it as usual.

Case 3 $\mathcal{A}$ and $\mathcal{B}$ belong to different statements and have a similar locality:
This case implies a strengthening of the alignment preference, because the possible communication effort may arise twice if the two dimensions are not aligned. We model this by accumulating the weights of both preferences. Since we allowed similar localities to have a small difference in the constant offset, we store the maximum and minimum offset of all accumulated preferences.

Case 4 $\mathcal{A}$ and $\mathcal{B}$ belong to different statements and have a different locality:
This case is equivalent to the second one.

4.4 Solving the Intraprocedural Alignment 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 [19], 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 as shown below.

Other researchers tried to solve the problem using 0/1 integer programming algorithms [14, 11] neglecting the NP-completeness of the problem. This requires them to accept a huge amount of computation time.

Our heuristic algorithm begins by creating initial Alignment Sets: each Alignment Set contains a single element which is a dimension of an array in the code. The array selected must have the highest accumulated weight of alignment preferences in that program unit. At each step the algorithm constructs a bipartite graph from the current Alignment Sets and an array which is to be matched to them. There is a node for each dimension of this array. Once matched, the nodes are merged with the corresponding Alignment Set. A weighted edge connects a set with an array node if there is an edge between this node and any node belonging to the set in the CAG. When building the bipartite graph, the intradimensional problem is handled as shown below.

In the original algorithm, indirect alignment preferences are given a weight (in most cases too high) independent of other matchings; this has been improved by a new algorithm for building a bipartite graph which only considers direct alignment preferences (the transitivity of the alignment preference is observed by ordering the graph). Ayguade et al. proposed to use the min-cut of all paths connecting the nodes, thus achieving more accurate results [1]. We are planning to investigate their proposal in the near future.

Furthermore the results computed by the original algorithm depend to some extent on the (random) order in which columns were chosen. This problem can be eliminated by ordering the CAG with respect to the weight of the edges.[1]

Impact of Intradimensional Preferences

In Section 4.2 we emphasized the importance of observing intradimensional conflicts and showed how to

[1] An outline of the heuristic algorithm can be found in [17].

model them. The same is valid for the interdimensional solution phase. However, conflicting intradimensional preferences do not only occur between the same two nodes, but also during the matching step in the interdimensional solution.

In order to detect these additional intradimensional conflicts, we have to modify the locality in such a way that it expresses a relationship between a node and all nodes belonging to a set rather than a relationship between two single nodes. We therefore relate all the localities to one node, called the *basis node*. The first node associated with a set becomes its basis node[2] and is attributed with a perfect locality. When an array dimension is included in an Alignment Set, the locality relationship with the basis node is stored in order to retain details of intradimensional alignment information. We call this the *node locality*. When other nodes are successively matched with a set, the edge of the bipartite graph is annotated with alignment information summarizing the information of all CAG edges between the candidate node and any node associated with the set.

The node locality is propagated to the edge in order to establish a relationship between the candidate node and the basis node. The algorithm for propagating locality information in order to relate all of them to the basis node can be found in [17].

If we propagate all locality information while constructing the bipartite graph, we are able to compare pairs of alignment preferences and deal with them as described in the distinguishing cases of Section 4.3. Thus all edges are merged if there are no conflicts, or several distinct edges connect the bipartite graph in the case of intradimensional conflicts.

Solving the Intradimensional Alignment Problem The solution of the intradimensional alignment problem consists of two steps: first potential conflicts (expressed by the fact that there is more than one edge between a set and a node in the bipartite graph) have to be solved; then the nodes matched to an Alignment Set have to be annotated with their locality information relative to the basis node (the *node locality*).

The detection of conflicts in the intradimensional alignment preferences indicates that a certain amount of communication effort will remain, no matter what distribution is chosen. To model this situation correctly we would have to accumulate the weights of all competing edges, reduced by the unavoidable communication effort. Unfortunately without any a priori knowledge of distributions and the available processors, we are not able to estimate this.

To overcome this problem, we rely still on worst case assumptions. We assume that only one of the intradimensional preferences can be observed, the others cannot. Hence we consider only the edge with the highest weight for the bipartite graph matching step and ignore all the others.

Let us denote by e_{be} an edge of the bipartite graph between the Alignment Set $\mathcal{S}_i$ and the node B_j. $E = \{e(A_k, B_j) | A_k \in \mathcal{S}_i\}$ is the set of all alignment edges between $\mathcal{S}_i$ and B_j. E can be split up into n disjoint subsets $E_1, ..., E_n$ such that all edges $e \in E_i$ have a *similar* locality. Then the weight of the bipartite graph's edge can be computed by:

$$\omega(e_{be}) = \max_{E_i \in E} \left(\sum_{e \in E_i} \omega(e) \right) \tag{4}$$

The second step of this phase is solved in a straight forward manner. The node is attributed with the propagated locality from the edge of the bipartite graph. Of course, it is important to record whether conflicts with respect to a node arise during analysis. Such an information can indicate that the user should take care when distributing this array. In order to do so, we introduce a new locality type *competing*; a node is attributed with this locality type if intradimensional alignment conflicts occurred.

The output of the heuristic algorithm are the Alignment Sets, containing all nodes of the CAG so that the weights of the edges between nodes belonging to different sets are minimized. Those edges represent unavoidable communication and reflect conflicts in the interdimensional alignment. 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

[2] Note that the basis nodes of all sets may not necessarily belong to the same array.

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.

5 Interprocedural Alignment 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.

Although realignment can potentially be appropriate anywhere in a program, a common approach for the problem of realignment (and redistribution) is to split a program into disjoint *phases* and allow realignment only between phases. However, a number of definitions for phases exist and an efficient and useful specification of these is still a topic of ongoing research. A good overview of related work can be found in Kremer's thesis [16].

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 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.

Our implementation puts several major restrictions on the Fortran programming style: we cannot handle equivalence and reshaping of arrays at call sites; common blocks are handled only if the component arrays match in shape and the size of their data type. These are severe in view of common programming practice, but any attempt to deal with them will increase the complexity of this problem enormously.

5.1 Problem Formulation

The intraprocedural analysis phase results in an ACG (cf. Section 3) 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.

Definition 6. Let $\mathcal{C}$ be any CAG with nodes A_i, $B_j \in \mathcal{C}$ and $\mathcal{S}$ the Alignment Sets of $\mathcal{C}$. Let furthermore $set(A_i)$ be a function returning the number of the alignment set to which A_i is matched, and $loc(e)$ and $loc(A_i)$ be a function returning the locality of edge e and node A_i, respectively. Let $\mathcal{DS}$ denote the set of all edges between nodes belonging to different sets:

$$\mathcal{DS} = \{e(A_i, B_j) \mid set(A_i) \neq set(B_j)\} \tag{5}$$

and $\mathcal{DL}$ the set of all edges with a locality relation that is not observed:

$$\mathcal{DL} = \{e(A_i, B_j) \mid set(A_i) = set(B_j) \wedge loc(e) \text{ not similar } loc(A_i)\} \tag{6}$$

so we can define the cost of the alignment scheme $\mathcal{S}$ (alignment cost, ac) as:

$$ac(\mathcal{S}) = \sum_{e \in \mathcal{DS}} \omega(e) + \sum_{e \in \mathcal{DL}} \omega(e) \tag{7}$$

□

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 5.2). 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 Set 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 (3):

$$arc(A) = (tt * \text{ arraysize}(A) + st) * 2 \tag{8}$$

Note that the realignment cost is multiplied by 2, since realignment occurs on the call and on the return. Furthermore the edge is annotated with a pointer to the array A in order to allow the identification.

The solution of the realignment problem can now be formulated as a matching problem for this bipartite graph. All arrays which are connected to an edge between nodes in the final graph have to be realigned. Hence we can define the total realignment cost (rc):

Definition 7. Let $\mathcal{S}_p$ and $\mathcal{S}_q$ be the Alignment Sets of program unit p and program unit q, p calls q. Let furthermore $\mathcal{R}_c$ denote the set of all arrays which have to be realigned on a call $c \in calls(p,q)$.

The total realignment cost between $\mathcal{S}_p$ and $\mathcal{S}_q$ for a call $c \in calls(p,q)$ ($rc(\mathcal{S}_p, \mathcal{S}_q, c)$) can be defined as:

$$rc(\mathcal{S}_p, \mathcal{S}_q, c) = \sum_{A \in \mathcal{R}_c} arc(A) \tag{9}$$

$\square$

The bipartite graph matching algorithm seeks the minimum of rc.

The above model yields the optimal solution for the choice of arrays to be realigned. Another benefit of this method is that it uses the same cost model as the intraprocedural alignment model, thus both measures can easily be compared. 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.

5.2 The Procedure Clustering Algorithm

Finding an optimal solution for dynamic alignment and data distribution schemes was proved to be NP-complete by Kremer [15]. We have selected a greedy algorithm to solve the procedure clustering problem heuristically. Our algorithm was inspired by the *Clustering Algorithm* developed by Ning et al. [20]. 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 2.

The algorithm needs one pass over the ACG and tries to combine those procedures which are connected by a maximal 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 4.4) 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 (cf. Section 5.2).

```
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)
   */

  /* 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. 2. Procedure Clustering Algorithm

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 [22] 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 [17]).

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:

$$r\omega(e) = \omega(e)/F(p) \tag{10}$$

We apply therefore Equation (10) after Equation (3) in the intraprocedural analysis step, in order to be able to generate interprocedural solutions.

When merging two CAGs the weights of the callee are multiplied with the relative frequencies of the call, thus leading to the absolute weight if it is merged with all calling procedures.

The merging algorithm uses the concepts provided by our intraprocedural analysis and is depicted in Figure 3.

```
merge_CAG(CAG1,CAG2)
{
 /* copy the callers CAG */
 merge_CAG = copy_CAG(CAG1)

 foreach (c in calls(caller,callee)) do
 {
   substitute_formal_params(CAG2,argument_list);
   propagate_locality(CAG2,CAG1)
   multiply_edges_weight(CAG2,rcf(c))
   insert_nodes(merge_CAG,CAG2);
   insert_edges(merge_CAG,CAG2);
 }
 return merge_CAG;
}
```

Fig. 3. CAG Merging Algorithm

First a copy of the caller's CAG is generated in which the callee's CAG is inserted. For each call we have to substitute the dummy arguments with the actual ones according to their bindings and propagate their locality relations. 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 (cf. Section 4.2).

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 nodes of these procedures are merged into a common cluster and the edges are updated accordingly. The updating algorithm performs the following steps:

- A new node is generated in the ACG and attributed with the merged CAG and its solution.
- All edges entering one of the two merged procedures are redirected to the new node.
- The new node holds pointers to the call graph G for both procedures.
- All edges leaving the caller are copied to the new node.
- The relative frequencies of all edges leaving the callee are updated (multiplied by the relative frequency the callee is called by the caller) and the edges are copied to the new node.
- The nodes for the original two procedures are deleted from the ACG.

After the execution of the PCA step 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 all 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 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.

6 Application Example

In this section we illustrate our interprocedural algorithm by considering the *shallow* benchmark from the xHPF benchmark set[3]. This code is a straightforward weather prediction benchmark program that uses finite differencing.

The ACG of *shallow* can be seen in Figure 4. 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).

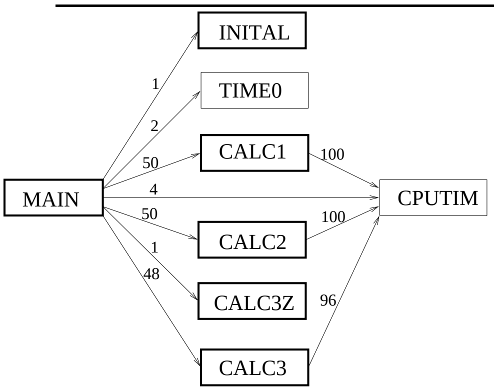

Fig. 4. ACG for shallow

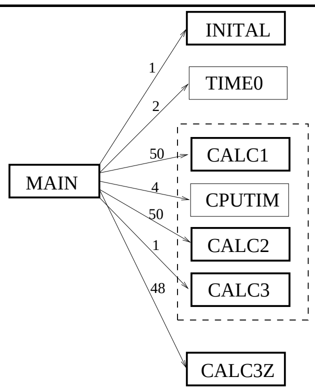

Fig. 5. Partially merged (1)

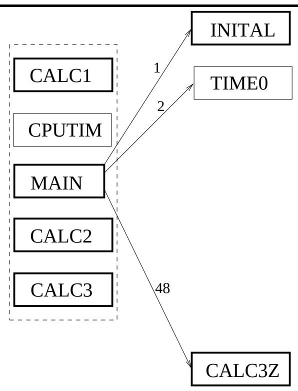

Fig. 6. Partially merged (2)

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 5). 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 6. 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. The resulting CAG consists of 29 nodes (14 two-dimensional arrays and 1 one-dimensional) and 72 edges. The solution for the whole program is depicted in Table 1.

[3] The xHPF benchmark set is available by anonymous ftp at ftp.infomall.org in directory tenants/apri/Bench

Note that for the one-dimensional array involved, the preferences for Alignment Set 1 are the same as for Alignment Set 2. Thus this solution might not necessarily be the best and the user receives a warning. In fact, the diagonal of all matrices is assigned to this array.

Note the different localities given in the solution. Although only stencils with an offset of 1 can be found in the code, constant localities occur in the range -2 to 3 in our solutions. This is due to the propagation of the localities as stated in Section 4.4. Without our intradimensional analysis, the user might well not be aware of these differing alignment possibilities. We plan to give the user more hints by also computing the offset value with the highest weight.

In Appendix A code excerpts of the most important loops of the subroutines `CALC1`, `CALC2`, and `CALC3` are given. These loops require 98 % of the overall sequential execution time. To illustrate the reason for the range in the constant offsets possibilities, we illustrate its computation for node Z_2 in detail referring to the code given in the appendix. We explain, how each element successively matched to $\mathcal{S}_2$ is assigned its locality tuple and give the associated line number from the code excerpt. Note that we only give one example for a preference resulting in an offset, although there might be multiple preferences.

The first node matched to $\mathcal{S}_2$ is P_2. It therefore becomes the *basis node* and is attributed with a perfect locality. The next two nodes ($POLD_2$ and $PNEW_2$) have no offsets in their access patterns (with respect to P_2) and are consequently also attributed with a perfect locality. CV_2 is attributed with the locality tuple (1;0,1) because of its access in Statement 2: `CV(I,J+1)=.5(P(I,J+1) + P(I,J)) ....` The next node to be matched is CU_2 which is again attributed with a perfect locality. The first negative value in the range of constant offsets can be found when matching node $UNEW_2$. The reason is the access patterns in Statement 5: `UNEW(I+1,J)=...CV(I,J+1)+CV(I,J)....` As can be seen from this access pattern, $UNEW_2$ can either be perfectly aligned with CV_2 or with a left shift of 1, resulting in a locality tuple (1;-1,0) for $e(UNEW_2,CV_2)$. Since CV_2 is attributed with (1;0,1) we have to accumulate the offsets resulting in the locality tuple (1;-1,1) for $UNEW_2$. In the same statement we can find the reason for the tuple (1;-1,2) of Z_2. From the access patterns `UNEW(I+1,J)=...Z(I+1,J+1)+Z(I+1,J)...` we deduce a locality tuple of (1;0,1) for $e(Z_2,UNEW_2)$. Again, we have to accumulate this tuple and the one of $UNEW_2$ resulting in the locality tuple (1;-1,2) for Z_2.

The other ranges result from similar access patterns, however, not all of them are contained in the given code excerpts.

The idea behind the range in the constant offset is: depending on the chosen alignment function for the previous arrays, the alignment function for an array may have an constant offset within the given range with respect to the *basis array*.

SET 1		**SET 2**	
Node	*Locality*	*Node*	*Locality*
P_1	perfect	P_2	perfect
$POLD_1$	perfect	$POLD_2$	perfect
$PNEW_1$	perfect	$PNEW_2$	perfect
CV_1	perfect	CV_2	constant (1;0,1)
CU_1	constant (1;0,1)	CU_2	perfect
$UNEW_1$	constant (1;0,1)	$UNEW_2$	constant (1;-1,1)
Z_1	constant (1;0,1)	Z_2	constant (1;-1,2)
$VNEW_1$	constant (1;-1,1)	$VNEW_2$	constant (1;-1,2)
H_1	constant (1;-1,1)	H_2	constant (1;-2,2)
V_1	constant (1;-1,1)	V_2	constant (1;-2,3)
$VOLD_1$	constant (1;-1,1)	$VOLD_2$	constant (1;-2,3)
U_1	constant (1;-1,2)	U_2	constant (1;-2,2)
$UOLD_1$	constant (1;-1,2)	$UOLD_2$	constant (1;-2,2)
IO_TEMP_1	constant (1;-1,2)		
PSI_1	constant (1;-1,2)	PSI_2	constant (1;-2,3)

Table 1. Alignment Sets for shallow

7 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. We showed how to perform this, including models for an exact weighing of alignment preferences and propagation methods for the intradimensional preferences (the *locality*). Cost models are furthermore introduced to measure the cost of a candidate alignment scheme as well as realignment cost at procedure boundaries. Much attention was paid to optimizing the necessary realignment schemes. 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. [11, 12] 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.

References

1. E. Ayguade, J. Garcia, M. Girones, J. Labarta, J. Torres, and M. Valero. Detecting and Using Affinity in an Automatic Data Distribution Tool. In *Languages & Compilers for Parallel Computing*, number 892 in Lecture Notes in Computer Science. Springer Verlag, August 1994.
2. 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, Computer Architecture Department, Polytechnic University of Catalunya, Barcelona, Spain, 1995.
3. S. Benkner. *Vienna Fortran 90 and its Compilation.* PhD thesis, Institute for Software Technology and Parallel Systems, University of Vienna, November 1994. Also available as Technical Report TR 94-8.
4. S. Benkner, S. Andel, R. Blasko, et al. *Vienna Fortran Compilation System Version 1.2, User's Guide.* Institute for Software Technology and Parallel Systems, University of Vienna, Vienna, Austria, February 1996.
5. 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.
6. M. Chen, Y. Choo, and J. Li. Theory and Pragmatics of Compiling Efficient Parallel Code. Technical Report YALEU/DCS/TR-760, Yale University, Dezember 1989.
7. C. G. Diderich and M. Gengler. Solving the constant-degree parallelism alignment problem. In *Proc. of EuroPar '96*, August 1996.
8. 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.
9. T. Fahringer. The Weight Finder, An Advanced Profiler for Fortran Programs. In *International Workshop of AP '93*, Saarbrücken, 1993.
10. High Performance Fortran Forum. High Performance Fortran Language Specification Version 1.1, November 1994.
11. 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. Center for Research on Parallel Computing, Rice University.

12. J. Garcia, E.Ayguadè, and J. Labarta. Dynamic Data Distribution with Control Flow Analysis. In *Proceedings Supercomputing 96*, Pittsburgh, PA, November 1996.
13. M. Gupta. *Automatic Data Partitioning on Distributed Memory Mulitcomputers.* PhD thesis, Coordinated Science Lab, University of Illinois at Urbana-Champaign, 1992. Also available as Technical Report UILU-ENG-92-2237.
14. K. Kennedy and U. Kremer. Automatic Data Layout for High Performance Fortran. In *Proceedings of Supercomputing 95*, San Diego,CA, December 1995.
15. U. Kremer. NP-completeness of Dynamic Remapping. In *Fourth International Workshop on Compilers for Parallel Computers.* Delft University of Technology, Dezember 1993.
16. U. Kremer. *Automatic Data Layout for Distributed Memory Machines.* PhD thesis, Computer Science, Rice University, October 1995. Also available as Technical Report CRPC-TR95559-S.
17. E. Laure and B. Chapman. Alignment Analysis within the VFCS - A pragmatic Method for Supporting Data Distribution. Technical Report TR 96-2, Institute for Software Technology and Parallel Systems, University of Vienna, Austria, October 1996.
18. 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.
19. 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.
20. 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.
21. V. Van Dongen, C Bonello, and C. Freehill. High Performance C Language Specification. Technical Report CRIM-EPPP-94/04-12, Center de Recherche Informatique de Montrèal, Montrèal, Québec, Canada, 1994.
22. H.P. Zima and B.M. Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press Frontier Series. ACM, Addison-Wesely, 1990.

A Code Excerpts

```
C     SUBROUTINE CALC1
C     ****************

      DO 100 J=1,N
         DO 100 I=1,M
1:          CU(I+1,J) = .5*(P(I+1,J)+P(I,J))*U(I+1,J)
2:          CV(I,J+1) = .5*(P(I,J+1)+P(I,J))*V(I,J+1)
3:          Z(I+1,J+1) = (FSDX*(V(I+1,J+1)-V(I,J+1))-FSDY*(U(I+1,J+1)
     1                   -U(I+1,J)))/(P(I,J)+P(I+1,J)+P(I+1,J+1)+P(I,J+1))
4:          H(I,J) = P(I,J)+.25*(U(I+1,J)*U(I+1,J)+U(I,J)*U(I,J)
     1                   +V(I,J+1)*V(I,J+1)+V(I,J)*V(I,J))
  100 CONTINUE



C     SUBROUTINE CALC2
C     ****************

      DO 200 J=1,N
         DO 200 I=1,M
5:          UNEW(I+1,J) = UOLD(I+1,J)+
     1                    TDTS8*(Z(I+1,J+1)+Z(I+1,J))*(CV(I+1,J+1)+CV(I,J+1)
     2                    +CV(I,J)+CV(I+1,J))-TDTSDX*(H(I+1,J)-H(I,J))
6:          VNEW(I,J+1) = VOLD(I,J+1)-TDTS8*(Z(I+1,J+1)+Z(I,J+1))
     1                    *(CU(I+1,J+1)+CU(I,J+1)+CU(I,J)+CU(I+1,J))
     2                   -TDTSDY*(H(I,J+1)-H(I,J))
7:          PNEW(I,J) = POLD(I,J)-TDTSDX*(CU(I+1,J)-CU(I,J))
     1                 -TDTSDY*(CV(I,J+1)-CV(I,J))
  200 CONTINUE



C     SUBROUTINE CALC3
C     ****************

      DO 300 J=1,N
         DO 300 I=1,M
8:          UOLD(I,J) = U(I,J)+ALPHA*(UNEW(I,J)-2.*U(I,J)+UOLD(I,J))
9:          VOLD(I,J) = V(I,J)+ALPHA*(VNEW(I,J)-2.*V(I,J)+VOLD(I,J))
10:         POLD(I,J) = P(I,J)+ALPHA*(PNEW(I,J)-2.*P(I,J)+POLD(I,J))
11:         U(I,J) = UNEW(I,J)
12:         V(I,J) = VNEW(I,J)
13:         P(I,J) = PNEW(I,J)
  300 CONTINUE
```