Alignment Analysis within the VFCS
A Pragmatic Method for Supporting Data Distribution

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

One of the major tasks in programming distributed memory multiprocessors with state of the art parallel languages is the specification of efficient data distribution schemes. Unfortunately, there are no optimal strategies for generating such data distributions and thus automatic support is very difficult to provide; several heuristics have been proposed to provide some support to the user. In this paper we outline the Alignment Analysis Tool which has been implemented within the Vienna Fortran Compiler VFCS. The tool is able to provide help for finding a good distribution scheme in a pragmatic and general purpose way. It automatically generates alignment proposals for the arrays accessed in a procedure and thus simplifies the data distribution problem. Based upon the fundamental work by Li & Chen and Manish Gupta, our main contributions to the alignment problem are the extension of the set of alignment preferences detected as well as the simple but powerful heuristic for weighing alignment preferences using an explicit compiler model. We furthermore developed a method that considers both aspects of the problem - the inter- and the intradimensional alignment problem - simultaneously. We can thus provide a powerful general purpose tool able to manage even large programs in acceptable time.

1 Introduction

Distributed memory multiprocessors (DMP) are now widely used for scientific applications because of their scalability and performance at relatively low cost. A frequent method of programming DMPs is the Single Program Multiple Data (SPMD) paradigm, where multiple processors work on the same problem but on a distinct data space. However, programming these machines at the commonly-used message-passing level is a difficult and error prone task, because the programmer must take care of performance issues (via the data and work distribution) and the semantical correctness of the program by synchronizing the parallel tasks explicitly.

Therefore high level data parallel languages have been developed which provide a sufficient level of abstraction to alleviate the task of programming DMPs. These languages provide the programmer with conceptually a single thread of control and a range of explicit parallel constructs (such as parallel loops) and high level primitives for the specification of data and work distribution. Based on these user specifications the compiler is able to insert the necessary communication routines automatically. Examples for such languages are High Performance Fortran

(HPF) [15], and its predecessors Fortran D [20], and Vienna Fortran [6, 4]. High Performance C [27] has provided similar features for C.

These languages require the user to specify the data and sometimes the work distribution. However, the choice of distribution scheme will have a large influence on the execution time, since it will greatly affect the load balance as well as the amount and pattern of data exchange between processors. Finding a satisfying distribution scheme is a nontrivial task and support for this step is necessary. However, even greatly simplified data partitioning problems and subproblems like finding redistribution schemes are NP-complete [22]. There are trade-offs between the goals of spreading workload evenly across processors and minimizing communication. Therefore schemes for providing automatic support for data and work distribution (the latter one can be coupled to the data distribution using the owner computes paradigm) rely heavily on heuristics and the overall task is divided into subproblems (e.g. the alignment problem or the redistribution problem). In many cases the solutions of these subproblems provide the user with substantial support in this difficult task.

In this paper, we present a tool dealing with the problem of reducing communication cost due to array cross-references by detecting which dimensions of arrays are generally used together; this problem is generally called the **Alignment Problem**. The solution of this problem may be used to provide the user with information on how to couple dimensions of arrays in order to minimize communication, or it may be used to produce alignment directives (as provided for example in HPF[15]) automatically.

We will show how this problem can be handled in a heuristic manner at low computation cost and how to integrate such an analysis tool in the framework of an existing semi-automatic parallelizer, the VFCS. Our work is based upon the pioneering work by Li and Chen [23], using their CAG-framework, and that of Manish Gupta [18]. We extend and modify their work in order to achieve more accurate results and to enlarge the applicability of the tool.

We present a new heuristic for weighing alignment preferences with respect to their assumed importance for the overall runtime performance, and show how inter- and intradimensional array usage patterns can be observed. We have tried to keep the heuristics as simple as possible to avoid time consuming computations, while still obtaining a sufficient level of reliability. In this version, only intraprocedural analysis will be performed.

The alignment analysis described in this paper has been fully implemented within the VFCS. The implementation supports the user who is creating a Vienna Fortran or HPF program by providing recommendations for the alignment of arrays in the individual subroutines of the code.

The remainder of this paper is organized as follows: In Section 2 we give a short introduction to the VFCS compiler, emphasizing those features used for alignment analysis.
Section 3 is the main part of this work and presents the alignment analysis strategy. It is divided into three subsections: the first introduces the alignment problem, showing how to construct the Component Alignment Graph (CAG) and how to weigh its edges, the second subsection presents a heuristic for solving the alignment problem and in the final subsection, some details about the integration in the VFCS are given.
An example application in Section 4 illustrates some of the main aspects of this work.
After a short discussion of related work, we conclude with a brief summary and some future research directions.

2 The VFCS

The Vienna Fortran Compilation System (VFCS) is an interactive compiler for programs written in Vienna Fortran[1] or a subset of HPF. It can furthermore be used to parallelize sequential applications written in Fortran 77 and provides sophisticated analysis and transformation features to support and optimize the parallelization process. VFCS translates the input program to equivalent message passing programs for different platforms.

VFCS puts a great deal of effort into optimizing in particular the communication within the generated message passing program. An important part of the VFCS is the analysis component which derives data dependence information and performs interprocedural data flow analysis. Another important feature is the performance subsystem, which provides performance measurement and estimation facilities [13, 14]. A full overview of the current state of the VFCS can be found in the User's Guide [5].

The alignment analysis tool takes as input the internal representation of the source program, as built in the VFCS frontend. It makes some assumptions about the manner in which the data parallel program using alignment directives it suggests will subsequently be compiled. This underlying compiler model is in fact a simplified version of the VFCS. The *Weight Finder* [13], an advanced profiler for Fortran 77 programs provides the tool with sequential performance data needed to estimate the potential communication effort. *Dependence analysis* in VFCS is used to model the effects of the compiler's communication optimization [17].

In our approach, we model the loop extraction of communications (cf. 3.1.3) only, but the extension to model communication fusion also would be quite simple.

3 The Alignment Analysis

In this section we provide an overview of the Alignment Analysis Tool, which is built on top of the VFCS and uses the features described above in particular. Note that we define the alignment analysis as the detection of relationships between two dimensions of different arrays by analyzing their subscript expressions. Therefore we often refer simply to *subscript expression* when we mean a subscript expression of a specific dimension of an array reference.

In Subsection 3.1 we provide a description of the underlying problem, which is solved in a heuristic manner in Subsection 3.2. Subsection 3.3 finally focuses on some specific issues related to its integration in the VFCS.

3.1 Problem Description

The main aim of alignment is to reduce the cost of communication due to cross-references among different distributed arrays. This goal is achieved through the mapping of the relevant dimensions of arrays to common sets and applying the same distribution function on all members of such a set. In other words, all array dimensions belonging to the same set are coupled for the subsequent distribution step. This problem is generally known as the *interdimensional alignment problem* while the problem of finding the relative position of two arrays is called the *intradimensional alignment problem*. We show how both problems can be handled simultaneously and refer therefore to the unified problem as the *alignment problem*.

[1]Vienna Fortran [6] is based upon Fortran 77 with a few extensions for expressing explicit parallelism and for specifying data distributions.

3.1.1 Basic Concepts

Our approach of alignment analysis is based on the analysis of array references in various statements inside every loop in the examined program unit. As mentioned before, only intraprocedural analysis is performed.

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 needs not to 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 introduced by Li and Chen [23], and rely on the work of Manish Gupta [18]. Nevertheless we make substantial modifications to this fundamental approach to enlarge the number of alignment relationships detected and to model their relative importance in a more adequate manner.

3.1.2 Constructing the CAG

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 [18] 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 dimension of a subscript expression contains one and only one DO-loop variable and all the other components are integer constants) of subscript expressions.

In order to increase the number of alignment preferences which can be detected, we enlarge the concept of the single index type by allowing loop invariant components in the subscript expressions:

Definition 1 *Let $x_d(a)$ be of the form $\alpha * i + \gamma$. If α and γ are loop invariant, $\alpha \neq 0$, and i is a loop variable, then this subscript expression has the property* **single index**. □

Further we define two subscript expressions to **conform** when they are not of type single index, but identical except of a possible shift, which of course expresses also an alignment preference.

Definition 2 *Let $x_d(a)$ be of the form $\sum_{j=1}^{n} \alpha_j * i_j + \gamma$; and $x_t(b)$ be of the form $\sum_{j=1}^{n} \beta_j * i_j + \delta$. If $\alpha_j = \beta_j \; \forall j, 1 \leq j \leq n$ and $\gamma - \delta = ic$ (ic is any loop invariant component), then these subscript expressions are said to* **conform**. □

With these definitions we are now able to define an improved alignment preference compared to those of [23] and [18]. In general, alignment preferences can be found between a 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 in reduction operations, and helps to improve the overall results.

```
L1:  DO 10 I = 1, 100
L2:    DO 20 J = 1, 100
S1:      A(I,J) = B(J,I)+C(I+1,J+2)
S2:      C(I,J) = A(J,I)
  20   CONTINUE
  10 CONTINUE
```

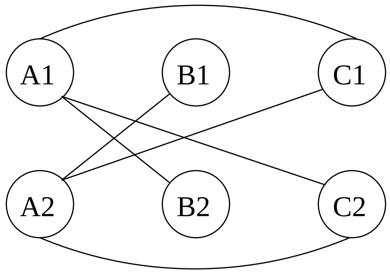

Figure 1: Construction of a CAG

Definition 3 *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**. □

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

With the help of Definition 3 we are now able to construct the **Component Alignment Graph (CAG)** as introduced by [23] by making a node for each dimension of each array and connecting all nodes expressing an alignment preference with an undirected edge, called **alignment edge** (e). Each edge is annotated with some information about the kind of alignment preference (this will be stated in more detail in the following section), so the CAG can be interpreted as an undirected, weighted graph.

Figure 1 illustrates the CAG which has been built from the code excerpt also given there.

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.

3.1.3 Attributing the Alignment Edges

As mentioned above, the edges of a CAG are annotated with some information about the underlying alignment preference. We can distinguish between interdimensional (i.e. how important is the alignment of two array dimensions for the overall performance) and intradimensional information (i.e. the relative position of array references). We can model both kinds of information within the CAG framework, thus needing only one analysis step. Below the main strategy for annotating an alignment edge with both inter- and intradimensional alignment information is outlined.

Interdimensional Information Due to the fact that the alignment analysis should provide some information helpful for avoiding communication, a weight indicating the presumed communication effort arising from a cross reference might be a useful information about interdimensional preferences. Li and Chen [23] modeled a very coarse estimate by attributing an edge with the value of 1 and ε respectively, depending on whether communication cost could be avoided or not. Gupta [18] extended their approach by estimating the possible magnitude of communication cost, but uses strong assumptions and time consuming heuristics to do so.

In our approach we also try to obtain a comparable heuristic measure for assumed communication cost, but we will achieve this goal with simpler heuristics, thus reducing the time spent in computation while nevertheless gaining a sufficient level of reliability. Note that our aim is not to predict the parallel performance, but to compute a reliable measure for comparing alignment preferences. Since neither distribution schemes nor the number of processors are known at analysis time, we cannot provide a realistic measure of communication cost. Hence we use strong worst case assumptions regarding the possible communication effort throughout our work. A test run of the program is profiled using the Weight Finder, and we use the execution frequency of specific statements it detects, as well as values of loop bounds and true ratios.

A simple approach for estimating the presumed worst case communication cost would be to assume that at each single statement instance one data item has to be communicated. However, this approach is quite straight forward and even without any communication optimization such a situation will hardly occur at each statement. Another possibility would be to assume that we have to communicate one data item each time the value of the subscript expression changes. This assumption makes sense since it allows us to examine each dimension independently of the others and is thus closely related to the interdimensional alignment problem. But again, this approach does not take into account any dependence information nor communication optimization and can again lead to inexact results which might differ from the real communication cost in both directions. Nevertheless, we will compute this value, denoted as *alignment frequency*. To do so it is necessary to record the *variation level* of an alignment preference. Since an alignment preference is symbolized by an alignment edge, this edge can be attributed with the variation level and we refer to it by $vl(e)$. The alignment frequency is then equal to the execution frequency of a (virtual) statement nested at the level $vl(e)$. This measure will be used only to show the effects of our communication optimization model which we describe below.

A more exact estimation of the real worst case communication cost can be achieved by modeling the compiler optimizations, especially the vectorization of communication. Since we know, that we can move the communication outside the loops if no true dependence is violated [29], we can denote the nesting level to which the communication caused by an alignment edge may be extracted as the communication dependence (**cd**). Hence the cd is equal to the deepest nesting level of loop-carried dependences or the deepest nesting level of a statement involved in a loop-independent dependence relationship.

Now we can easily compute both the frequency with which a communication routine might be invoked, and the amount of data items each message will transmit in the worst case:

Definition 4 *Let $F(S_n)$ be the measured sequential execution frequency of a statement at nesting level n, $vl(e)$ the variation level of alignment edge e, $cd(e)$ the communication dependence of e and $ni(L_{vl(e)})$ the number of iterations of the loop at level $vl(e)$. With this information we can calculate the data amount sent by each communication instance (*communication amount *ca*) and the frequency a communication routine might be invoked (*communication frequency *cf*) as follows:*

$$ca = \begin{cases} 1 & \text{if } vl(e) \leq cd(e) \\ ni(L_{vl(e)}) & \text{if } vl(e) > cd(e) \end{cases} \qquad cf = F(S_{cd(e)})$$

□

Note that so far we do not include the true ratio in case of control dependent statements, since also the compiler's parallelization process uses worst case assumptions for each conditional branch. Anyway, the true ratios for each statement are provided by the Weight Finder and can be included in the below formula.

We also included a simple machine model in our communication model using the average startup time a machine needs to start a transmission and the average transfer time required to transmit a single data item.

Now we are able to compute the respective weight of an alignment edge using a simple, yet effective method that leads to Definition 5:

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:*

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

$\omega(e)$ denotes 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 of an alignment edge will be used in subsequent steps to evaluate the relative importance of an alignment preference for solving the alignment problem and matching the nodes to the respective sets (cf. 3.2).

Intradimensional Information Another issue when attributing the alignment edges is the observance of intradimensional alignment preferences. By intradimensional alignment preferences we mean the relative position of the two array references to each other, mainly shifts, strides, and reflections. A shift is represented by different constant offsets in the subscript expressions, a stride results from different index coefficients, and a reflection is a special case of a stride where one of the two index coefficients is negative.

Due to the fact that the alignment analysis is restricted to regular subscript expressions of type single-index (an exception are *conforming* subscript expressions which will be handled separately), intradimensional alignment preferences are expressed through the quotient of the two index coefficients and the difference of the two constant components, respectively.

Although this has also been observed by other researchers, the impact of intradimensional preferences on the interdimensional alignment problem is not yet considered in full detail. We handle this topic by introducing a new concept in order to get more accurate results. As the intradimensional preferences have an impact on the interdimensional preferences, we will model them together in the framework of the CAG.

For modeling intradimensional alignment preferences we introduce the concept of **locality**:

Definition 6 *Let e be an alignment edge associated with two subscript expressions $x_d(a)$ and $x_t(b)$. The* **locality** *between these subscript expressions is defined as a tuple (***locality type, locality relation***) and e is attributed with this tuple.*

The **locality relation** *consists of the tuple (***multiplicator, constant offset***) or (m,o) where the multiplicator is defined as the quotient of the two index coefficients (α/β) and the constant offset is defined as the difference of the two constant components ($\gamma - \delta$) respectively.*

The **locality type** *is defined as follows:*

$$\textit{locality type} = \begin{cases} \textit{perfect} & \textit{if } m = 1 \wedge o = 0 \\ \textit{constant} & \textit{if } (m \neq 1 \vee o \neq 0) \wedge m \textit{ and } o \textit{ are integer constants} \\ \textit{invariant} & \textit{otherwise} \end{cases}$$

□

Note that the locality is a directed, non symmetric relation[2], while an alignment edge represents an undirected symmetric relation between two array dimensions.
Due to the lack of symmetry in the locality, a kind of "direction" is put on the alignment edges which must be observed very carefully when handling the CAG.

For alignment edges associated with *conform* subscript expressions we can deduce from Definition 2 that their locality relation has a multiplicator of 1 and a constant or invariant offset.

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.

The example presented in Section 4 illustrates the concepts introduced so far.

3.1.4 Multiple Alignment Occurrences and Intradimensional Conflicts

The locality is used to detect and solve conflicts in the intradimensional alignment scheme. Such conflicts can occur when more than one pair of references to the same arrays express an interdimensional alignment preference.

Other researchers deal with this problem by simply accumulating the weights of all preferences thus increasing the probability for its consideration in the final solution. This procedure is however inaccurate and we show, that better results can be obtained using the concept of locality.

The following example illustrates the importance of locality when handling multiple occurrences of the same interdimensional preference:

Example
Suppose the following two statements are both nested in arbitrary loops and loop variable `I` goes from `1` to `N`. Both arrays express an interdimensional alignment preference in the first and second dimension, respectively. However in the second statement the `I` subscript of array A is used in a reflection function.

```
A(I,J) = B(I,J)
B(I,J) = A(N-I,J)
```

If we simply accumulate the weight of these two preferences we would model the situation incorrectly, since in this case a certain amount of communication remains no matter what distribution function will be applied. Fortunately we are able to detect such cases via the locality: the locality of the two preferences would be *(perfect,(1,0))* and *(constant,(-1,0))* respectively, thus making an intradimensional alignment conflict evident. ◇

We can define the case when the localities of two alignment preferences have no conflicts and refer to them as *similar localities*:

Definition 7 *Let* (m_1, o_1) *and* (m_2, o_2) *be two locality relations. They are said to be* **similar** *if* $(m_1 = m_2) \wedge (o_1 - o_2 = c)$*, where* c *is a small constant.* □

In the above definition we allow a difference in the constant offset in order to include stencil operations in it. 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

[2] These properties hold for constant and invariant localities. Perfect localities are obviously undirected and symmetric.

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. The one of the four 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. The locality relation therefore has the form $(m, o_{min}/o_{max})$.

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

The following example based upon the code fragment given in Figure 2 motivates our differentiation.

Example
The analysis of the code in Figure 2 results in alignment preferences between A_1 - B_2 and A_2 - B_1 in both statements. The intradimensional analysis of statement S1 results in a perfect locality for both preferences. The preferences in statement S2 belong to one of the above cases:

The preference between A_1 - B_2 due to the first occurrence of A on the rhs is annotated with a perfect locality, thus it belongs to case 3 and the weights of both preferences are accumulated. The second occurrence of A on the rhs also expresses an alignment preference with B_2 annotated with a perfect locality. This preference belongs to case 1, thus nothing needs to be done.
We can further detect an alignment preference between A_2 - B_1, which is however annotated with a constant locality due to the shift in the subscript of B. Hence this preference belongs to case 4 and a second edge is constructed between those nodes, thus representing an intradimensional conflict. When looking at the CAG depicted in Figure 2 we can see two edges between those nodes: edge "a" is generated from statement S1 whereas edge "b" results from statement S2. $\diamond$

3.2 Solving the Alignment Problem

As proposed by Li and Chen, the Alignment Problem can be solved by matching each node of a 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

```
L1:     DO 10 I=1,100
L2:        DO 10 J=1,100
S1:           A(I+15,J)=B(J,I+15)
10      CONTINUE
L3:     DO 20 I=1,100
L4:        DO 20 J=1,100
S2:           B(J+20,I)=A(I,J)+A(I,3)
20      CONTINUE
```

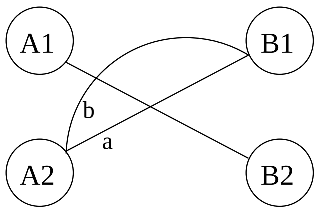

Figure 2: Example for multiple Alignment Preferences

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 [23], however Li and Chen presented an heuristic algorithm which reduced the problem to a bipartite-graph matching problem.

Other researchers tried to solve the problem using 0/1 integer programming algorithms [21, 16] neglecting the NP-completeness of the problem. This requires them to accept a huge amount of computation time.

The 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 dealt with 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 [2]. We are planning to investigate their proposal in the near future.

Furthermore the results computed by the original algorithm depended to some extent on the (random) order in which columns were chosen. This has been overcome by ordering the CAG with respect to the weight of the edges. We present an outline of the new heuristic algorithm in Figure 3.

Impact of Intradimensional Preferences

In Section 3.1.4 we emphasized the importance of observing intradimensional conflicts and showed how to 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

```
Heuristic_Algorithm(CAG)
{
 Sort_Graph(CAG);

 Distribute_First_column(CAG,Alignment_sets);
 /* distribute the nodes of the first column from the sorted graph among the Alignment_sets */

 column_to_be_matched=Second_Column(CAG);

 while (column_to_be_matched!=NIL)
 {
        x=Form_Bipartite_Graph(column_to_be_matched,Alignment_sets);

        Mx=Optimal_Matching(x);
        /* the optimal matching of the bipartite graph is computed
           (for this polynomial time algorithms exist) */

        Distribute_Column (column_to_be_matched,Mx,Alignment_sets);
        /* distribute the column among the Alignment_sets according to the computed matching */

        column_to_be_matched=Next_Column(CAG);
        /* get the next column from the sorted CAG */
 }
}


Sort_Graph(CAG)
{
 /* sort the CAG according to the following rules:

    1) the first column is the one with the highest weight of all edges leaving the column.

    2) compute the accumulated weight of all edges between the nodes of the sorted part of the
       CAG and the columns not yet sorted. The column with the maximum of these weights is
       placed next.

    3) Repeat 2) until the whole graph is sorted.
 */
}


Form_Bipartite_Graph(column,sets);
{
 bigraph =
 /*
        Form a bipartite graph, whose nodes of the first column represent the Alignment_sets,
        the second column is the column, that should be matched. Create an edge, if the node
        of the second column is direct connected with any node of an Alignment_set. Attribute
        this edge with the highest accumulated weight of all edges between the node and all
        nodes of the set having the same relative locality.
        Set the locality of this edge to the appropriate type and locality relation.
 */
 return bigraph;
}
```

Figure 3: Heuristic Algorithm

to one node, called the *basis node*. The first node associated with a set becomes its basis node[3] 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 is given in Figure 4.

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 3.1.4. Thus all edges are merged if there are no conflicts, or several distinct edges connect the bipartite graph in the case of intradimensional conflicts.

The following example gives an intuitive understanding of the propagation algorithm. Remember that the locality relation is directed and we express its direction by specifying the sink node.

Example
Consider the following code excerpt with the given alignment preferences:

`A(I) = B(100-I)`	$e(A_1, B_1)$: locality = constant, locality relation for B_1=(-1,0)
`A(I) = C(100-I)`	$e(A_1, C_1)$: locality = constant, locality relation for C_1=(-1,0)
`B(I) = C(I)`	$e(B_1, C_1)$: locality = perfect

Furthermore we assume that A_1 is matched as the first node to Set 1, thus becoming the *basis node*. The next node we want to match to Set 1 is B_1, having a constant locality with A_1. It is therefore attributed with its locality relation (-1,0). Now C_1 is to be matched and we detect two alignment preferences ($e(A_1, C_1)$, $e(B_1, C_1)$), but with different localities. There is a reflection of C_1 in relation to A_1 and a perfect locality to B_1. Since B_1 is also a reflection in relation to A_1, we can propagate the locality information of B_1 to the edge $e(B_1, C_1)$ as shown in Figure 4. This results in a constant locality expressing a reflection for this edge, thus solving the intradimensional conflicts, as is evidently appropriate for the code. $\diamond$

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 function 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 distribution functions and the available processors, we are not able to estimate this.

To overcome this problem, we decided to rely still on worst case assumptions. We assume that only one of the intradimensional preferences can be observed, the others cannot. Hence we

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

Let `A` *be a node matched to an Alignment Set and* `loc(A)` *its associated locality. If a node* `B` *with an alignment edge* `e(A,B)` *should be matched to the same set, the locality information of* `A` *is propagated to* `e` *as follows:*

```
if (loc(e)==perfect)
     loc(e)=loc(A);
elsif ((loc(e)==invariant)||(loc(A)==invariant))
     loc(e)=invariant;
elsif ((loc(e)==constant)||(loc(A)==constant))
{    loc(e)=constant;
     constant offset(e)=multiplicator(e)*constant offset(A)
                        + constant offset(e);
     multiplicator(e)=multiplicator(e)*multiplicator(A);
}
```

Figure 4: Propagation of Locality Information

consider only the edge with the highest weight for the bipartite graph matching step and ignore all the others.

Let us denote by e_{bg} 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:

$$\omega(e_{bg}) = \max_{E_i \in E} \; (\sum_{e \in E_i} \omega(e))$$

The second step of this phase is solved in a straight forward manner. We can simply deduce the propagated locality an edge of the bipartite graph is annotated with to the node. Of course, it is important to record whether conflicts regarding a node have arisen 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 have 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 is 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 dimensionality to be the *alignment source array* and align all other arrays with it. Subsequently, the user only has to specify the distribution functions for this single alignment source array.

3.3 Integration in the VFCS

In this section we provide a short overview of the Alignment Analysis Tool which has been built as an integrated part of the VFCS. The implementation follows the definitions and procedures above closely, so we do not provide many details.

The overall structure of the tool is presented in Figure 5 and can be divided into 3 phases:

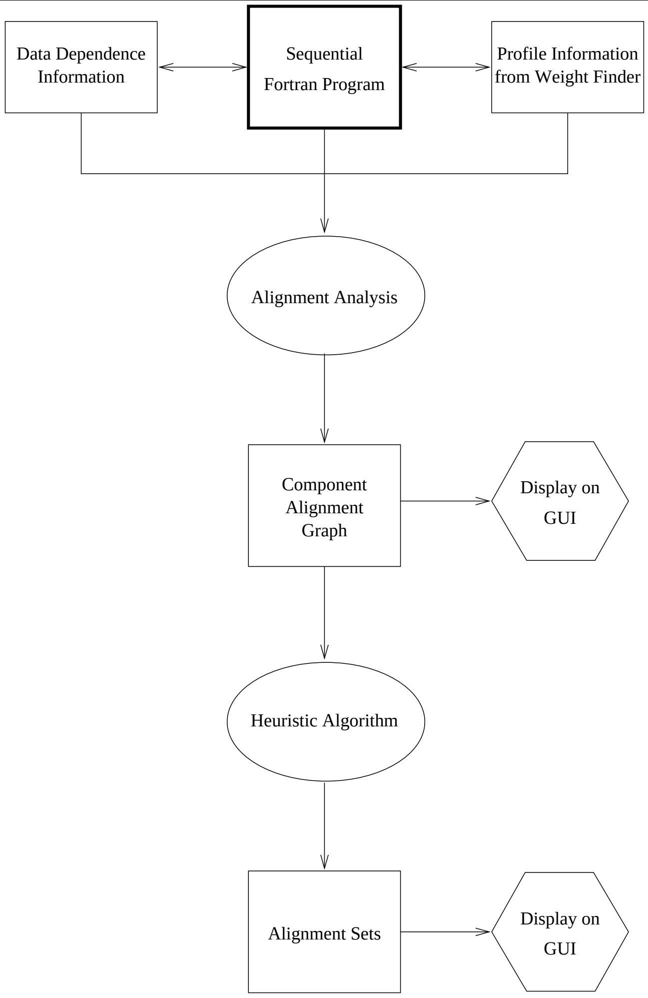

Figure 5: Overall Structure of the Alignment Tool

3.3.1 Phase 1 - Frontend

The first phase aims to produce a suitable form of the program for the further steps. In this phase, the VFCS frontend creates the internal representation and performs initial analysis as well as data dependence analysis. The Weight Finder will be run in order to provide information about frequencies, loop parameters, and true ratios needed for the calculation of the weights.

In this step some very important program transformations are applied, namely the *scalar forward substitution* and the *induction variable substitution* [28], which are able to enlarge the number of detectable alignment preferences significantly. In [3], empirical studies are presented which support this assumption.

The user then specifies the region for analysis. This is currently a program unit or a specific

```
L1:   DO 10 I = 1, 100                          <-- FR=1; LP=[1.00:100.00]
L2:      DO 20 J = 1, 100                     <-- FR=100; LP=[1.00;100.00]
S1:          A(I,J) = B(J,I)+C(I+1,J+2)                      <-- FR=10000
S2:          C(I,J) = A(J,I)                                 <-- FR=10000
  20     CONTINUE                                              <-- FR=100
  10  CONTINUE                                                   <-- FR=1
```

Figure 6: Example Loop

part of a unit. This selection mechanism was implemented to enhance the potential further use of this information in a variety of settings. It could be especially useful for analyzing phases with possible changes in the distribution scheme.

3.3.2 Phase 2 - Alignment Analysis

The second phase performs the alignment analysis proper. The selected part of the code is analyzed by a single run over the syntax tree. All array references in nested assignment statements are analyzed and each pair of them is tested for alignment preferences (cf. Definition 3) in all dimensions. If two references prove to have an alignment preference, all relevant information for the edge attributes is computed using the Weight Finder and dependence analysis information.

The result of this phase is a fully annotated CAG, which is made available to the user via a graphical user interface. This CAG is also the basis for the subsequent third phase in which the alignment problem is solved. It is clearly possible, that in extreme cases no CAG is generated at all, thus reducing the execution of phase three to a trivial operation.

3.3.3 Phase 3 - The Heuristic Algorithm

Assuming that Phase 2 has built a CAG, the third phase will solve the alignment problem using the heuristic algorithm as presented in Section 3.2. An outline of this heuristic algorithm was given in Figure 3.

The task of the heuristic algorithm is the solution of the intra- and interdimensional alignment problem by reducing the latter one to a bipartite graph matching problem. This is implemented using a branch & bound strategy [19].

As the result of this third phase, all nodes of the CAG are matched to the Alignment Sets (thus representing the solution of the interdimensional problem), which are made available to the user via the graphical interface. The nodes are annotated with their relative locality information, thus representing the solution of the intradimensional problem.

4 Application Example

In this section we give an example to show how our tool works. Profile data generated by the Weight Finder is printed to the right of each statement in our example loop, shown in Figure 6 (`FR` gives the associated frequency and `LP` the value of loop bounds).

In this example both statements are nested and all references within them contain alignment preference information, however, with some intra- and interdimensional conflicts. While the intradimensional conflicts can be neglected (this is a stencil computation with a maximum overlap of 2), the references to array A and C reflect an interdimensional conflict which is unsolvable. Hence we must accept a certain amount of communication, but the alignment analysis will help us to minimize this.

The results of the second phase (the alignment analysis) are plotted in Figure 7 (the CAG) and Table 1, in which all necessary information about the alignment edges is contained: for each edge we print the alignment frequency, all the locality information (note that in the case of a constant locality the direction of the locality relation is given by printing the sink node), the dependence information (0 means that no dependence is detected and the eventual communication can be extracted from loops), and finally the weight (the startup time was fixed with 100 and the transfer time with 0.1 - this reflects the real order of magnitude for the Meiko CS-2). Note that the dependence information is not an edge attribute. It is printed in order to demonstrate the computation of the weight. (In fact the dependence information cannot be annotated to an edge because an edge may symbolize multiple statements.)

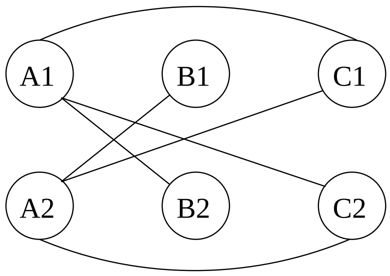

Figure 7: CAG

EDGE	*FREQ.*	*LOCALITY TYPE*	*LOCALITY RELATION*	*DEP.*	*WEIGHT*
A_1 - B_2	100	perfect	(1,0)	0	110
A_1 - C_1	100	constant	(1,1) for C_1	0	110
A_1 - C_2	10000	perfect	(1,0)	2	1001000
A_2 - B_1	10000	perfect	(1,0)	0	110
A_2 - C_1	100	perfect	(1,0)	2	1001000
A_2 - C_2	10000	constant	(1,2) for C_2	0	110

Table 1: Analysis Results

The importance of modeling communication optimization becomes evident when looking at the two edges $e(A_1, C_2$ and $e(A_2, C_2)$, which both have an alignment frequency of 10000 but the difference of their weights is on the order of 10^4.

This CAG is finally the basis for the heuristic algorithm which tries to find the solution for the interdimensional alignment problem. Though there are many conflicts in the CAG, the solution is obvious because of the varying weights of the edges. Figure 8 plots the two sets (since the maximum dimensionality of all arrays examined is two). Note that the nodes are annotated with locality information, relative to the first node (In fact all nodes are annotated with a perfect locality because the two edges with constant locality are not considered to be observed for the final solution.)

We measured our example loop with several different alignment and distribution settings on the MEIKO CS-2 using the *pghpf* Compiler from The Portland Group [25]. The results

Figure 8: Alignment Sets

can be found in Figure 9. The Figure shows that our suggested solution outperformed all other alignment settings. These results confirm our hypothesis that our alignment suggestions will be among the best ones independently of the distribution function (despite the fact that not all aspects of parallelization are modeled). The faster execution times with the `(*,BLOCK)` distribution can be explained by the fact that parallelization of the (distributed) `J`-loop is possible here, but not in the other case.

5 Related Work

The research field of automatic alignment and automatic data distribution has become very popular during the last few years. Hence there is a lot of work related to this topic and we restrict ourselves to some of the major contributions to the field in this section, especially those which have a strong impact on our work.

Li and Chen at Yale University played a fundamental role in introducing the concept of alignment analysis [23] in the framework of the functional language *Crystal* [9]. 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 [18]. 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.

Robert et al. at LIP-Lyon first introduced the *communication graph* [10] which models write and read accesses on arrays and tries to locate them on the same processor using matrix equations. In subsequent work they introduced the *access graph* [12] which is actually a refinement of their previous work.

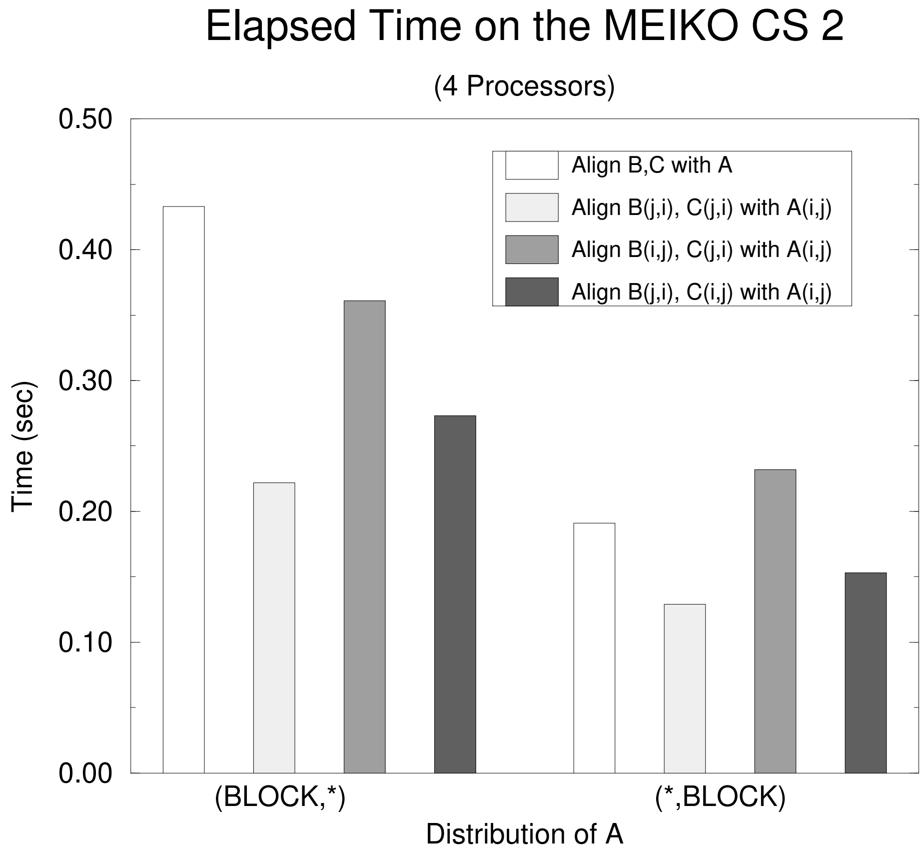

Figure 9: Runtime Results with the PGI Compiler

Garcia et al. [16] introduced the *Communication-Parallelism Graph* which is able to model the alignment problem and the problem of work distribution in one step using parallelism-hyper-edges for the latter problem. The solution method uses 0/1 integer programming techniques and is only able to distribute one dimension.

Kennedy and Kremer [21] presented a data distribution tool for the Fortran-D Project [20] which aims to provide dynamic distribution functions through the detection of "phases" inside a program. They also use 0/1 integer programming techniques.

Ning et al. [24] developed a data distribution tool for HPC [27] which also performs dynamic redistribution. They introduced a *Locality Algorithm* for solving the data distribution problem and a *Clustering Algorithm* for finding phases, the latter one using the concept of the *Loop Program Dependence Graph.*

Chatterjee et al. [8, 7, 26] developed a system for automatic data distribution using heuristic algorithms in the field of dynamic programming, solving both the intra- and interdimensional alignment problem, and for automatic data distribution subsequently, using the concept of the *Alignment-Distribution Graph* which is also able to model dynamic distributions.

6 Conclusion

In this paper we gave a short introduction to the alignment problem in general and its concrete realization within the context of the VFCS. We presented the major features for modeling and solving this problem and demonstrated their functionality on a simple example.

In the formulation of the problem a major concern was the applicability of the methods used, thus providing a general purpose, pragmatic tool which can be successfully used for commercial systems.

Our main contributions were the formulation of a simple yet effective method for weighing alignment preferences, using performance data provided by a sequential profiler, and the combination of intra- and interdimensional alignment preferences within one graph model. We introduced the locality concept to represent intradimensional preferences inside the CAG framework.

Future work will extend this approach for interprocedural solutions and will also provide automatically generated alignment and distribution directives. The integration of this tool in larger analysis systems such as a knowledge based Expert Adviser [1] or the PAP Recognizer [11], a tool based upon pattern matching facilities, is currently under examination.

References

[1] S. Andel, B.M. Chapman, J. Hulman, and H. Zima. An Expert Advisor for Parallel Programming Environments and Its Realization within the Framework of the Vienna Fortran Compilation System. Technical Report TR 95-4, Institute for Software Technology and Parallel Systems, University of Vienna, September 1995.

[2] 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.

[3] E. Ayguade, J. Labarta, J. Garcia, M. Girones, and M. Valero. Detecting Affinity for Automatic Data Distribution. In Mango M. Furnari, editor, *Massive Parallelism, Hardware, Software and Applications*, pages 32–46. World Scientific, London, 1994.

[4] 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.

[5] 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.

[6] B.M. Chapman, P. Mehrotra, and H.P. Zima. Programming in Vienna Fortran. *Scientific Programming*, Vol. 1(1):31–50, Fall 1992.

[7] S. Chatterjee, J.R. Gilbert, R. Schreiber, and T.J. Sheffler. Array Distribution in Data-Parallel Programs. Technical Report 94.09, RIACS, NASA Ames Research Center, 1994.

[8] 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.

[9] 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.

[10] A. Darte and Y. Robert. A Graph-Theoretic Approach to the Alignment Problem. Technical Report Nr. 93-20, Laboratoire de l'Informatique du Parallèlisme, Ecole Normale Supèrieure de Lyon, July 1993.

[11] B. Di Martino and B.M. Chapman. Program Comprehension Techniques to improve Automatic Parallelization. In *Workshop on Automatic Data Layout and Performance Prediction*, Houston, April 1995. Center for Research on Parallel Computing, Rice University.

[12] 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.

[13] T. Fahringer. The Weight Finder, An Advanced Profiler for Fortran Programs. In *International Workshop of AP '93*, Saarbrücken, 1993.

[14] T. Fahringer. Using the P^3T to Guide the Parallelization and Optimization Effort under the Vienna Fortran Compilation System. In *IEEE Proceedings of the Scalable High Performance Computing Conference*, Knoxville, TN, May 1994.

[15] High Performance Fortran Forum. High Performance Fortran Language Specification Version 1.1. Technical Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, Houston, November 1994.

[16] 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.

[17] M. Gerndt. *Automatic Parallelization for Distributed-Memory Multiprocessing Systems.* PhD thesis, Rheinische Friedrich-Wilhelms-Universität, Bonn, 1989.

[18] 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.

[19] F.S. Hillier and G.J. Lieberman. *Introduction to Operations Research.* McGraw-Hill, New York, 1990.

[20] S. Hiranandani, K. Kennedy, et al. An Overview of the Fortran D Programming System. In *Fourth Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, CA, August 1991.

[21] K. Kennedy and U. Kremer. Automatic Data Layout for High Performance Fortran. In *Proceedings of Supercomputing 95*, San Diego,CA, December 1995.

[22] U. Kremer. NP-completeness of Dynamic Remapping. In *Fourth International Workshop on Compilers for Parallel Computers.* Delft University of Technology, Dezember 1993.

[23] 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.

[24] 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.

[25] The Portland Group, Inc., Wilsonville, Oregon. *PGHPF User's Guide*, 1995.

[26] T.J. Sheffler, R. Schreiber, et al. Efficient Distribution Analysis via Graph Contraction. In *Workshop on Automatic Data Layout and Performance Prediction*, Houston, April 1995. Center for Research on Parallel Computing, Rice University.

[27] 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.

[28] H.P. Zima and B.M. Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press Frontier Series. ACM, Addison-Wesely, 1990.

[29] H.P. Zima and B.M. Chapman. Compiling for Distributed-Memory Systems. In *Proc. of the IEEE, Special Section on Languages and Compilers for Parallel Machines*, pages 264–287, February 1993.