Yale University
Department of Computer Science

Index Domain Alignment: Minimizing Cost of Cross-Referencing Between Distributed Arrays

Jingke Li and Marina Chen

YALEU/DCS/TR-725
November 1989

This work has been supported in part by the Office of Naval Research under Contract N00014-86-K-0310, N00014-86-K-0564 and N00014-89-J-1906.

Index Domain Alignment: Minimizing Cost of Cross-Referencing Between Distributed Arrays

(Extended Abstract)

Jingke Li Marina Chen

Department of Computer Science
Yale University
P.O. Box 2158, Yale Station
New Haven, CT 06520
li-jingke@yale.edu chen-marina@yale.edu

September 1989

Abstract

Programming distributed-memory machines requires that data associated with a given computation be partitioned and distributed to the local storage of each individual processor. How this distribution is done affects the amount of data movement required to carry information between different parts of the machine, which in turn affects program performance. This work addresses the issue of data movement between processors due to cross-references between multiple distributed arrays. The problem of *index domain alignment* is formulated as finding a set of suitable alignment functions that map the index domains of the arrays into a common index domain so as to minimize the cost of data movement due to cross-references between the arrays. The cost function and the machine model used are abstractions of the current generation of distributed-memory machines such as the Warp systolic machine [1], the Intel iPSC/2 [2], and the NCUBE hypercube multiprocessors [13]. The problem as formulated is shown to be NP-complete. A heuristic algorithm is devised and shown to be both efficient and providing excellent results.

1 Introduction

Programming distributed-memory machines requires *partitioning* of the program data structures. Slightly more subtle is the problem of *aligning* multiple data structures, in which the relative locations of these data are determined. In all of the CSP[7]-based languages with explicit "send" and "receive" commands, such information is provided by specifying directly the mapping of data structures to each processor. In higher-level languages that automate the generation of communications (e.g. AL[15], DINO[14], and Parascope[3]), partition and alignment are explicitly specified by the programmer using directives.

In this paper, we focus on the problem of automatically generating array alignment information based on array reference patterns in the source program. This method is used in the Crystal compiler[4, 12] and can be applied to the parallel implementations of other programming languages on distributed memory machines as well (e.g. those for parallelizing Fortran source programs).

We view a distributed data structure as a function from some index domain to some value domain (e.g. floating-point numbers). The data structures under consideration here are restricted to multi-dimensional arrays. In other words, index domains that are Cartesian products of interval domains, where an interval domain is a set of consecutive integers.

Alignment addresses the issue of reducing data movement between processors due to cross-references among different distributed arrays. The problem of alignment is formulated as finding a set of suitable alignment functions that map the index domains of these arrays into a common index domain.

The rest of this paper is organized as follows: Section 2 introduces some notation and provides a programming example illustrating domain alignment. In Section 3, we motivate the need for domain alignment and discuss the benefits and limitations of domain alignment as a compilation technique. Next, in Section 4, a model of a distributed memory machine and the cost of array references in the model are defined. In Section 5, the class of alignment functions under our consideration is presented. Section 6 provides the model of the component alignment problem. Discussions of the complexity and a heuristic algorithm for solving the problem are presented in Section 7. A few concluding remarks are given in Section 8. Finally, a proof of NP-completeness of the alignment problem is included in the appendix.

2 Notation

The Crystal language contains special data types for representing distributed data structures. First, an *index domain* specifies the shape of a data structure. Functions over index domains, called *data fields*, represent distributed data structures, unifying the notion of arrays and functions. Since they are functions, a set of data fields can be defined recursively.

A primitive index domain is an *interval domain*, denoted by $[m..n]$, where m and n are integers and $m \leq n$. An interval domain represents a set of consecutive integers. More complex index domains can be constructed from primitive domains by using domain constructors, which include *product*, *disjoint union*, etc. In this paper, we restrict index domains to be Cartesian products of interval domains. A data field defined over such an index domain represents a parallel computation defined over a multi-dimensional array. We use $\mathsf{dom}(a)$ to denote the index domain of a data field a. Each of the interval domains that constitutes the Cartesian product domain is referred to as a *component* of the product domain. The p-th component of domain $\mathsf{dom}(a)$ is denoted by $\mathsf{dom}(a,p)$.

Syntactically, index domains and data fields are specified as in the following example

$$\begin{aligned}
&\mathsf{dom}\ D_1 = [0..n],\\
&\mathsf{dom}\ D_2 = D_1 \times D_1,\\
&\mathsf{dfield}\ a(i) : D_1 = i + 2,\\
&\mathsf{dfield}\ b(i,j) : D_2 = \left\{ \begin{array}{l} i = 0 \rightarrow 0, \\ \mathsf{else} \rightarrow b(i-1,j) + a(j) \end{array} \right\},
\end{aligned}$$

where domain D_1 is an interval domain and D_2 is constructed from D_1 by the product constructor. Data field a is defined over domain D_1 while data field b is over domain D_2.

The general form for a conditional expression is

$$\left\{ \begin{array}{ccc} p_1 & \rightarrow & \tau_1 \\ \vdots & & \\ p_n & \rightarrow & \tau_n \end{array} \right\}$$

where the p_i's are boolean expressions called *guards*, and the τ_i's are arbitrary expressions. The guards p_i are mutually exclusive, and the value of the conditional expression is τ_k if p_k is the guard whose value is true.

The reduction operator "\" takes as arguments a binary associative function $\oplus : T \times T \rightarrow T$ over some data type T and an array $a = [a_1, \ldots, a_n]$ of elements of type T, and is defined as

$$\backslash\oplus\, a = a_1 \oplus \cdots \oplus a_n.$$

For example,

$$\backslash + [1,2,3,4,5,6] = 21.$$

Reduction can also be applied to sets of elements, provided $\oplus$ is commutative.

In the following, the Greek letters denote arbitrary expressions. Square brackets [ ] are used to select a subexpression. For example, when we want to emphasize the term $a(i,j)$ in the expression $a(i,j) + b(j)$, we use $\tau[a(i,j)]$ to denote the expression. Quasi-quotes $\ulcorner\urcorner$ are used to emphasize that we are interested in the syntactic form of the quoted expression.

Definition Given a data field definition of a containing a data field reference to b as follows:

$$\mathsf{dfield}\ a(i_1, \ldots, i_m) : D = \phi[\gamma \rightarrow b(\tau_1, \ldots, \tau_n)],$$

the symbolic form

$$\ulcorner a(i_1, \ldots, i_m) \leftarrow b(\tau_1, \ldots, \tau_n) : \gamma \urcorner$$

is called a *reference pattern.*

A reference pattern represents a collection of dependencies in an aggregate form. Multiple reference patterns can be derived from a data field definition if there are more than one instances of data field references occurring on the right-hand side of the definition. A reference pattern is either a *self-reference* pattern to the same data field (e.g. a calls a) or a *cross-reference* pattern between different data fields (e.g. a calls b).

Throughout this paper, we use Gaussian elimination with partial pivoting as an example to illustrate the alignment algorithm. A Crystal program for solving this problem is included in the appendix.

In the Crystal compiler, a program is first decomposed into *phases*; each phase comprises a set of data fields which are closely coupled to each other in terms of data dependence. A phase is treated as a unit in domain alignment. For the above example, two phases are constructed: one consists of data fields a, *ipivot*, *apivot*, and *fac*; the other consists of x and *psum*. The formal definition of phase can be found in [12].

3 Issues in Domain Alignment

Below are the definitions of two data fields a and b.

$$\begin{aligned} &\mathsf{dom}\ D = [1..n] \times [1..n], \\ &\mathsf{dfield}\ a(i,j) : D = b(j,i), \\ &\mathsf{dfield}\ b(i,j) : D = \tau, \end{aligned}$$

The reference pattern derived from the definition of a is

$$\ulcorner a(i,j) \leftarrow b(j,i) \urcorner$$

Even though the two arrays are of exactly the same shape, the distribution of their elements to processors need not necessarily be done in the same way. Due to the way b is referenced by a, it is beneficial to store $a(i,j)$ and $b(j,i)$ on the same processor to eliminate the need of communication. This simple example indicates that the compiler can choose the relative location of arrays by analyzing array reference patterns. In all of the examples that follow, data fields a and b are defined over index domain D as defined above.

The following example illustrates conflicting reference patterns:

$$\ulcorner a(i,j) \leftarrow b(i,j-2) : \gamma_1 \urcorner$$
$$\ulcorner b(i,j) \leftarrow a(j,i) : \gamma_2 \urcorner$$

Whether $a(i,j)$ is aligned to $b(j,i)$ or $b(i,j-2)$, the communications due to one of these two reference patterns cannot be reduced. Thus, the alignment problem must be formulated as an optimization problem where reference patterns may carry different weights.

Now let us look at a slightly more involved example where a reduction operator occurs in the definition of data field a.

$$\mathsf{dfield}\ a(i,j) : D = \backslash + \{b(k,i) \mid 0 \leq k < n\},$$

its reference pattern is

$$\ulcorner a(i,j) \leftarrow b(k,i) : 0 \leq k < n \urcorner$$

We consider first the simple scenario of mapping one element per processor. For each $(i,j) \in \mathsf{dom}(a)$, a set of elements of b is referenced. If we store $a(i,j)$, $b(i,j)$ at processor (i,j), then for each i we must do a reduction across the ith column of elements of b and distribute the result along the ith row. Alternatively, if $b(j,i)$ is stored in processor (i,j), only a reduction operation over a row is needed, provided the processors are connected by networks such as hypercubes, butterflies, etc., because the broadcast can be achieved at the same time as a side-effect of reduction [9, 6]. Thus alignment is related to the communication routines specific to the interconnection network of the target machine. We will discuss in the next section the abstract model of a target machine and the corresponding communication cost.

Next, for the same example, suppose each domain is partitioned into sub-domains, each of which is mapped to a processor: by aligning $a(i,j)$ with $b(j,i)$ for all $(i,j) \in D$ (i.e. transpose of b), and partitioning the domain along the first dimension (mapping a row into a processor), there will be no communication involved at all since the reduction operations now take place within a single processor. However, if $a(i,j)$ and $b(i,j)$ are mapped to the same processor, some communication must occur no matter how partitioning is done. We want to point out here that choosing the right partitioning strategy so as to "internalize" as many communications as possible is considered in a separate *domain partitioning* stage in the Crystal compiler dealing with minimizing the cost of self-references, and is beyond the scope of this paper. This example illustrates that alignment always helps in reducing cross-references from a to b, independent of domain partitioning.

The following example illustrates a reference pattern whose first component is a non-linear expression.

$$\ulcorner a(i,j) \leftarrow b(j^2 - j, i+j) : \gamma \urcorner$$

Though it might be possible to align a and b in such a way as to avoid communication, the cost of evaluating the extra conditional and non-linear expressions generated by the alignment process may exceed the cost of communication. Thus, there are some tradeoffs involved in doing alignment. We will discuss the class of alignment functions under our consideration.

Finally, any compilation technique is limited by what is known at compile time, and alignment is no exception. Below is an example where a reference pattern contains an indirect reference $a(i,j-1)$ whose value may not be known until the program is in execution. Hence, such reference patterns shall not be taken into account by the alignment algorithm.

$$\ulcorner a(i,j) \leftarrow b(a(i,j-1),\tau) : \gamma \urcorner$$

To summarize, alignment should use cost functions that reflect the communication costs on real machines and the symbolic forms of the alignment functions should be simple enough for a compiler to derive and to implement.

4 Reference Cost

In order to optimize domain alignment, we need to have a notion of reference cost. We classify reference patterns according to their "uniformity". The idea is to map individual index domains occurring in a program into a common index domain so that the reference patterns achieve maximum "uniformity".

Consider a reference pattern defined over a common n-dimensional index domain,

$$\ulcorner a(i_1,\ldots,i_p,\ldots,i_n) \leftarrow b(\tau_1,\ldots,\tau_p,\ldots,\tau_n) : \gamma \urcorner$$

We say the pattern is *uniform* in pth dimension if

$$\tau_p \cong \ulcorner i_p + c \urcorner$$

where c is a small constant independent of domain bounds, and $\cong$ denotes that the two expressions have the same canonical form. [1]

With this concept, we can classify reference patterns with respect to their *uniformity*, namely patterns that are uniform in every dimension; those that are non-uniform in one, two, three, etc. dimensions; and those that are non-uniform in every dimension.

In particular, we have the following special cases:

- **Local Memory Access:**

$$\ulcorner a(i_1,i_2,\ldots,i_n) \leftarrow b(\tau_1,\tau_2,\ldots,\tau_n) : \gamma \urcorner \text{ where } \tau_1 \cong i_1,\ldots,\tau_n \cong i_n.$$

- **Neighborhood Access:** patterns that are uniform in all the dimensions as in

$$\ulcorner a(i_1,i_2,\ldots,i_n) \leftarrow b(i_1+c_1,i_2+c_2,\ldots,i_n+c_n) : \gamma \urcorner \text{ where } c_1,\ldots,c_n \text{ are constants.}$$

- **Random Access:** patterns that are non-uniform in all the dimensions.

A memory access within a processor is often far faster than inter-processor communication. The difference in cost can be as big as 2 or 3 orders of magnitude.

To communicate with nearby processors within a constant distance, a message needs only to be routed through a small constant number of processors. Message collisions can be avoided. So neighborhood communications are the most efficient inter-processor communications.

Non-uniformity implies non-local communications, which are likely to cause message collisions. The more the non-uniform dimensions a reference pattern has, the higher the chance of message collisions.

Another reason for favoring more uniform references is because index domains will be partitioned into subdomains where each subdomain is assigned to a processor. Non-local communications can be eliminated by mapping the non-uniform dimensions into the memory of a single processor.

[1] A canonical form of an expression is a syntactic form in which variables appear in a predefined order and constants are partially evaluated. For example, $\ulcorner 2-i+j \urcorner$ and $\ulcorner j-i+3-1 \urcorner$ would have the same canonical form $\ulcorner -i+j+2 \urcorner$. Symbolic transformations and partial evaluations are carried out by a compiler to obtain canonical forms.

5 The Class of Alignment Functions under Consideration

An *alignment function* is a mapping from an index domain D to another index domain E. Correspondingly, a data field defined over D will be transformed to a new definition over E. This transformation of the definition can be done mechanically once the function is given [8]. The goal of selecting alignment functions is to transform those reference patterns in the original data field definition to ones that have the least cost in the new definition.

The uniformity notion defined earlier suggest that we relate the components of two index domains in such a way that maximum uniformity will be achieved. It also suggest that we reduce the constant offsets in each dimension of a reference pattern. Corresponding to these needs, we focus our attention to four simple types of alignment functions namely, *permutation*, *embedding*, *shift*, and *reflection*. All these alignment functions have a simple symbolic form and are easy to compute. This is important because otherwise the transformed program may have a very high computation overhead. In addition, finding optimal alignment can be expensive.

Without lost of generality, we assume that all the index domains are aligned to a common domain, and all the components of the common domain are large enough to accommodate any component of the individual index domains. For simplicity, we omit the boundary conditions in all the following definitions.

Definition For two n-dimensional index domains, D and E, an alignment function $g : D \longrightarrow E$ is said to be a *permutation* if

$$g(i_1, i_2, \ldots, i_n) = (i_{q_1}, i_{q_2}, \ldots, i_{q_n})$$

where $(q_1, q_2, \ldots, q_n)$ is a permutation of $(1, 2, \ldots, n)$.

Definition For an m-dimensional index domain D and an n-dimensional index domain E, where $m < n$, an alignment function $g : D \longrightarrow E$ is said to be an *embedding* if

$$g(i_1, i_2, \ldots, i_m) = (i_{q_1}, i_{q_2}, \ldots, i_{q_n})$$

where $(q_1, q_2, \ldots, q_n)$ is a permutation of $(1, 2, \ldots, n)$, and the expressions i_k, where $m < k \leq n$, are expressions that may contain $i_1, \ldots, i_m$.

For example, functions $g(i,j) = (i, 0, j)$ and $g(i,j) = (i, i+j, j)$ are both embeddings.

Definition Let D and E be two interval domains. An index domain function $g : D \longrightarrow E$ is said to be a *shift* if $g(i) = i - c$ where c is an integer.

Definition Let $D = [l..u]$ and $E = [(-u)..(-l)]$. An index domain function $g : D \longrightarrow E$ is said to be a *reflection* if $g(i) = -i$.

Permutation and embedding deal with transformation between different components of a domain (inter-component), while shift and reflection deal with transformation within a given component (intra-component). The inter-component alignment functions are useful in transforming reference patterns into more uniform ones. The intra-component alignment functions can be used to decrease the reference cost further by reducing the constant offsets.

Since the inter-component and intra-component alignment functions are independent of each other, the domain alignment problem can be solved in two separate steps: first by considering permutation and embedding, and then shift and reflection. Retiming [11] can be brought to bear on finding shifts. Our approach to finding reflections is ad hoc: we try to match special patterns. It turns out that generating appropriate permutations and embeddings is the central problem. Since it deals with only inter-component alignment, we call this problem the *component alignment* problem which is the focus of the following sections.

6 Modeling the Component Alignment Problem

In the definition of reference pattern given in Section 2, associated with the reference pattern $\ulcorner a(i_1, \ldots, i_m) \leftarrow b(\tau_1, \ldots, \tau_n) : \gamma\urcorner$ are two domains: the m-dimensional domain of data field a and the n-dimensional domain of data field b.

Definition Given a cross-reference pattern

$$\ulcorner a(i_1, \ldots, i_p, \ldots, i_m) \leftarrow b(\tau_1, \ldots, \tau_q, \ldots, \tau_n) : \gamma\urcorner$$

two domain components, $\mathsf{dom}(a,p)$ and $\mathsf{dom}(b,q)$, are said to have *affinity* if

$$\tau_q \cong \ulcorner i_p + c\urcorner$$

where c is a small constant.

The affinity relation between two domain components reflects a preference for aligning them. Since aligning components according affinity relation will increase uniformity. For example, from the following reference pattern:

$$\ulcorner a(i,j) \leftarrow b(j,i) : \gamma\urcorner$$

two affinity relations can be derived, one between $\mathsf{dom}(a,1)$ and $\mathsf{dom}(b,2)$ and the other between $\mathsf{dom}(a,2)$ and $\mathsf{dom}(b,1)$. If the two domains are aligned according to these relations, $a(i,j)$ and $b(j,i)$ will be mapped to the same processor, and hence no communication is needed.

We want to point out that the definition of affinity depends on the definition of reference cost, which varies with the communication characteristics of the target machines and the degree to which one desires to model them. The affinity definition can always be refined to allow special patterns to be included.

6.1 Component Affinity Graph

Component alignment is modeled as a graph problem. An undirected, weighted graph called a *component affinity graph* (CAG) is constructed from the source program based on the reference patterns as described below.

The nodes of the graph represent the components of index domains to be aligned. They are grouped in *columns*: each column contains those nodes representing components from the same index domain.

Using the concept of affinity, edges in a CAG are constructed as follows: For each distinct reference pattern (excluding self-reference ones) in the program, an edge is generated between two nodes if the two corresponding domain components have affinity. An edge is denoted by a pair $\langle \mathsf{dom}(a,i), \mathsf{dom}(b,j)\rangle$ where $\mathsf{dom}(a,i)$ and $\mathsf{dom}(b,j)$ are the two corresponding domain components.

Note that self-reference patterns are ignored, because alignments occur only between index domains of different data fields. Also, different instances of the same reference pattern are considered only once since a datum can be referenced many times after it is communicated.

Using edges to represent affinity relations between domain components does not take into account conflicting reference patterns. We introduce edge weights for this purpose.

Definition Two or more edges in a CAG are said to be *competing* if they are generated by the same reference pattern and they are incident on the same node.

Example

$$\ulcorner a(i,j) \leftarrow b(i,i) : \gamma_1\urcorner$$
$$\ulcorner b(i,j) \leftarrow a(i,j) : \gamma_2\urcorner$$

From the first reference pattern, two competing edges, $\langle \mathsf{dom}(a, 1), \mathsf{dom}(b, 1)\rangle$ and $\langle \mathsf{dom}(a, 1), \mathsf{dom}(b, 2)\rangle$, are derived. From the second reference pattern, two non-competing edges, $\langle \mathsf{dom}(a, 1), \mathsf{dom}(b, 1)\rangle$ and $\langle \mathsf{dom}(a, 2), \mathsf{dom}(b, 2)\rangle$, are derived. A non-competing edge indicates a strong preference for aligning the two domain components. A competing edge indicating more than one equally good alignment in the absence of any non-competing edge.

We assign weights to the edges of a CAG to reflect the strength of preference: each non-competing edge is assigned weight 1, and each competing edge is assigned weight ϵ (a value much smaller than 1).

A CAG so defined may contain multiple edges between a pair of nodes since there might be multiple reference patterns between two data fields. The graph can be simplified by replacing each set of multiple edges with a single edge whose weight is the sum of their weights. Figure 1 illustrates the CAG of the first phase (i.e. the forward elimination phase) of the Gaussian elimination program in the appendix.

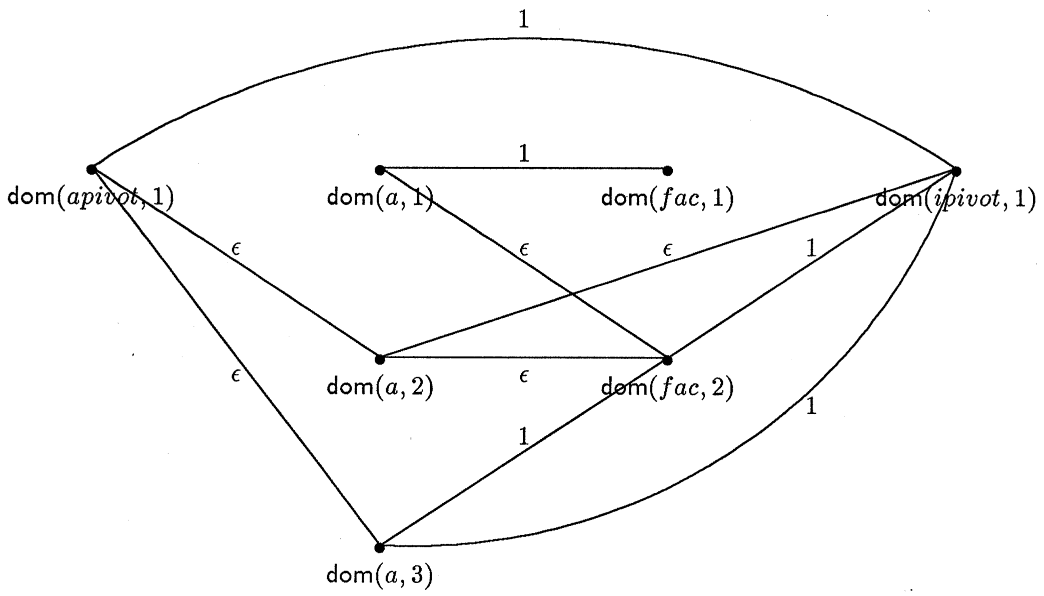

Figure 1: A component affinity graph.

6.2 Defining the Alignment Problem

Given a component affinity graph G as defined above, we can now define the component alignment problem as follows:

Let n be the maximum number of nodes in a column of G (i.e. n is the maximum dimensionality of all index domains to be aligned). Partition the node set of G into n disjoint subsets $V_1, V_2, \ldots, V_n$, with the restriction that no two nodes belonging to the same column are allowed to be in the same subset.

The underlying idea is that those nodes in the same subset correspond to the domain components to be aligned. Since our goal of alignment is to align those components that have affinity, we want to partition the component affinity graph G so as to minimize the total weight of edges that are between those nodes that are in different subsets.

Among the set of index domains $\mathcal{D}$ to be aligned, choose one which is of the highest dimensionality n to be the *target domain* E. A domain morphism g mapping from a domain $D \in \mathcal{D}$ (also of dimensionality n) into the target domain E can be constructed using the above graph partition:

$$g : D \to E,\ g(i_1, i_2, \ldots, i_n) = (i_{q_1}, i_{q_2}, \ldots, i_{q_n})$$

where the nodes representing component D_i and component E_{q_i} are in the same subset.

Similarly, an embedding morphism can be defined using the graph partition if D is of lower dimensionality than n. In this case, the component of the extra dimensions of E to which an elements of D maps need to be determined. In the following example:

$$\ulcorner a(i) \leftarrow b(i, i+1) : \gamma \urcorner$$

suppose the only component of $\mathsf{dom}(a)$ is aligned to the first component of $\mathsf{dom}(b)$, we now need to decide where $\mathsf{dom}(a)$ should reside with respect to the second dimension in $\mathsf{dom}(b)$. We check the reference patterns and find that the second component of $\mathsf{dom}(b)$ is referenced once with an expression $i+1$ in the definition of a. Clearly, if this reference pattern is the only one between a and b we want to align $\mathsf{dom}(a)$ with those elements $(i, i+1)$ in $\mathsf{dom}(b)$. In general, there can be more than one expression to be used if there are more reference patterns. Our approach is ad hoc in this case, using default constants such as the the lower bounds of the interval domains.

6.3 Alignment Results for Gaussian Elimination

Using the above definition, the optimal partition of the CAG of the Gaussian elimination program is illustrated in Figure 2.

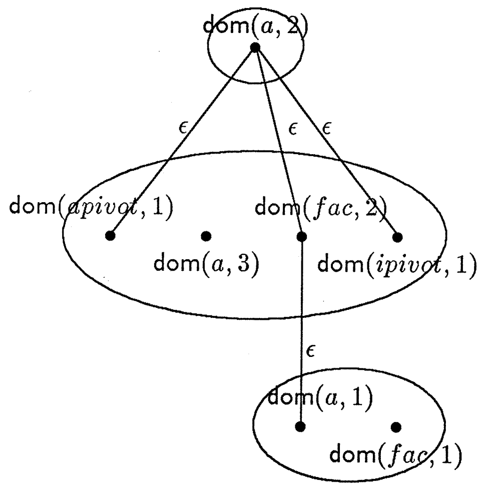

Figure 2: The optimal partition of the CAG shown in Figure 2.

The index domain of a, $\mathsf{dom}(a)$, is the target domain, since it is of highest dimensionality. $\mathsf{dom}(fac)$, which is 2-dimensional, is embedded as a plane lying diagonally within $\mathsf{dom}(a)$ by aligning its components with the first and third components of $\mathsf{dom}(a)$. The domains of *ipivot* and *apivot* are embedded in the domain of *fac* at the same location. Formally, these alignment morphisms are defined as the following functions mapping from the domain of a data field to the target domain $\mathsf{dom}(a)$:

$$g_1 : \mathsf{dom}(fac) \longrightarrow \mathsf{dom}(a),\ g_1(i,k) = (i,k,k)$$
$$g_2 : \mathsf{dom}(ipivot) \longrightarrow \mathsf{dom}(a),\ g_2(k) = (0,k,k)$$
$$g_3 : \mathsf{dom}(ipivot) \longrightarrow \mathsf{dom}(a),\ g_3(k) = (0,k,k)$$

For this particular example, applying the optimal alignment results in a 20% reduction in communication cost compared with a straightforward default alignment. [2]

[2]The default alignment function for an m dimensional index domain is to map its components to the first m components of the common domain with the original ordering.

7 Algorithms and Their Complexity

7.1 Component Alignment is NP-Complete

The component alignment problem described above, unfortunately, is expensive to solve. A special case of the problem is one in which all the index domains are of two dimensions. We can reduce the simple max cut (MAXCUT) problem [5] to this special case (with the number of index domains being the variable) and show it to be NP-complete. The proof is presented in the appendix. Due to this result, we do not expect any polynomial algorithm to find the optimal alignment for index domains of dimensions higher than two. Unfortunately, a naive exhaustive search algorithm is not practical: for a group of six 3-dimensional index domains, an exhaustive search algorithm may take two or more hours to find the optimal alignment on a Sun 3/50. Thus, we have devised the following heuristic algorithm.

7.2 A Heuristic Algorithm

The heuristic algorithm is essentially a greedy algorithm where a single index domain is chosen at each step for aligning with the target domain, and there is no back-tracking. We use the fact that the problem of aligning two index domains is just a bipartite graph matching problem and there exist efficient algorithms [10] to solve it.

The problem with such a greedy algorithm is that the quality of the alignment result is sensitive to that of the local alignment between two domains. A naive greedy algorithm that performs bipartite graph matching on the subgraph consisting of nodes and edges within two columns simply does not generate good results. The reason is that in this bipartite graph, the edge weights reflect adhesiveness only local to the two index domains. For example, suppose we have two edges $\langle \mathsf{dom}(a,2), \mathsf{dom}(b,1)\rangle$ and $\langle \mathsf{dom}(b,1), \mathsf{dom}(c,1)\rangle$ in a CAG. Even though there is no edge between $\mathsf{dom}(a,2)$ and $\mathsf{dom}(c,1)$, there is a preference to align these two nodes. What we really want is the closure of the adhesive relation. In the following heuristic algorithm, we augment the bipartite graph obtained from CAG with a new edge between every pair of nodes if they are connected in the original CAG. The weight of such an edge is defined precisely below.

Algorithm. Heuristic_Alignment(G)

This algorithm runs in N steps, where N is the number of columns in a component alignment graph G. In each step, an arbitrary column is aligned to the target column by applying the optimal matching procedure to a bipartite graph constructed from the nodes in the two columns. The graph contains global information about alignment preference.

```
1. choose the target column $C_T \leftarrow$ a column of $G$ with the maximum number of nodes;
2. $G_1 \leftarrow G$;
3. While $G_1$ is not empty, do
    (a) Pick a new column, $C_x$.
    (b) $G_x \leftarrow$ Form_Bipartite_Graph($C_T, C_x, G_1$);
    (c) $M \leftarrow$ Optimal_Alignment($G_x$);
    (d) $G_1 \leftarrow$ Reduce_Graph($M, C_T, C_x, G_1$);
```

Procedure. Form_Bipartite_Graph(C_T,C_x,G_1)

This procedure takes as input a CAG G_1 and two columns of nodes C_T and C_x, and constructs a bipartite graph with the closure of adhesive relation incorporated. Two nodes in the bipartite graph are connected by an edge if there is a path between these two nodes in G_1. The weight of such an edge is the sum of all the edge weights in the connected component of G_1 that contains these two nodes.

For each node pair (x, y), where $x \in C_T$ and $y \in C_x$, do

1. $G_2 \leftarrow$ the graph resulted from removing all the nodes in C_T and C_x, except for x and y, and all the edges that are incident on those nodes from G_1;
2. Set up an edge between x and y if they are connected in G_2;
3. If the edge exists, assign its weight to be the sum of the weights of the connected component that x and y belong to.

Procedure. Optimal_Alignment(G)

This is for finding the optimal weighted matching for a bipartite graph G. See [10] for polynomial time algorithms.

Procedure. Reduce_Graph(M,C_T,C_x,G_1)

Merge columns C_x and C_T by combining the matched nodes according to the matching M. "Clean" the graph by replacing multiple edges between two nodes with a single edge whose weight is the sum of their weights, and deleting all self cycles.

7.3 Experimental Results

To see how the heuristic algorithm might work in general, we have conducted the following experiment: apply three different domain alignment algorithms to a large number of synthetic component alignment graphs. The three algorithms are: a *naive* algorithm that simply aligns two domains at a time with the naive construction of the bipartite graph; the *heuristic* algorithm described above; and an *exhaustive-search* algorithm which produces the optimal result.

The data in each trial of the experiment is a synthetic component alignment graph with six columns each consisting of three nodes (i.e. six 3-dimensional index domains). The edges in the graph are randomly generated according to a fixed density (i.e. an edge appears in the graph with the probability equal to the specified density). Edge weights are integers randomly chosen between 1 and 10, inclusive.

Fig 3 shows the total edge weight of the resulting graph. The naive algorithm generates alignments which are at least 30% or more costly than the optimal ones. The alignments generated by the heuristic algorithm deviate from the optimal results by less than 10% in most of the cases. As to the times these algorithms take, both the naive and the heuristic algorithms run in a few seconds to a few minutes depending on the graph density while the exhaustive algorithm runs in twenty minutes (density 0.1) to a few hours.

8 Conclusion

Our primary goal is to automate the process of allocating multiple arrays on a distributed memory machine with minimized communication cost. Our technique does not depend on a particular hardware, but a class of machines which are captured by the reference cost model presented in the paper.

We feel that the importance of this array allocation problem cannot be over emphasized. In the future we will have distributed memory machines with tens of thousands of processors; since the ratio of message collision can be very high for random communication patterns on a large scale machine, reducing communication and regulating communication patterns on these machines will become increasingly more crucial to their performance. This paper makes both methodological and algorithmic contributions to the problem. In the small number of applications written in Crystal, we found domain alignment to be useful in reducing communication costs of target programs. However, to know the exact impact of domain alignment as an optimization technique, we plan to make a comprehensive survey of a large number of applications and study their performance improvement.

Edge Density	*# Edges*	*Total Edge Weights of the Resulting Graph*				
		Naive	N/E	*Heuristic*	H/E	*Exhaustive*
0.1	17	54	2.45	31	1.41	22
0.1	18	61	2.03	43	1.43	30
0.1	14	51	3.40	23	1.53	15
0.1	10	28	2.80	10	1.00	10
0.1	11	39	1.34	33	1.14	29
Average		46.6	2.20	28	1.32	21.2
0.2	27	135	2.29	72	1.22	59
0.2	32	101	1.53	66	1.00	66
0.2	32	120	1.82	72	1.09	66
0.2	33	126	1.56	86	1.06	81
0.2	34	130	1.48	89	1.01	88
0.2	31	98	1.40	79	1.13	70
Average		102	1.70	64.2	1.07	60
0.3	39	154	1.50	110	1.07	103
0.3	43	129	1.32	98	1.00	98
0.3	39	150	1.44	136	1.31	104
0.3	55	203	1.33	160	1.05	153
0.3	44	154	1.36	120	1.06	113
0.3	43	167	1.64	105	1.03	102
0.3	45	168	1.33	127	1.01	126
0.3	55	188	1.46	138	1.07	129
0.3	53	204	1.45	159	1.13	141
Average		168.6	1.42	128.1	1.08	118.8
0.4	62	235	1.28	184	1.00	184
0.4	68	222	1.17	189	1.00	189
0.4	62	216	1.41	174	1.14	153
0.4	64	232	1.41	168	1.02	164
0.4	66	266	1.34	214	1.08	199
Average		234.2	1.32	185.8	1.04	177.8
0.5	66	292	1.42	213	1.03	206
0.5	71	301	1.31	248	1.08	229
0.5	79	347	1.50	232	1.00	232
0.5	70	255	1.28	216	1.09	199
0.5	77	350	1.22	295	1.02	288
Average		309	1.34	240.8	1.04	230.8

Figure 3: Experimental Results of Three Alignment Algorithms.

Acknowledgement

We would like to thank Cheng-Yee Lin for the experimental work. Our thanks also go to Young-il Choo, Ron Pinter, and Michel Jacquemin for many helpful discussions. We thank Joe Rodrigue for reading the manuscript and his valuable comments. We gratefully acknowledge the generous support of the office of Naval Research under Contract No. N00014-89-5-1906.

References

[1] M. Annaratone, E. Arnould, T. Gross, H.T. Kung, M. Lam, O. Menzilcioglu, and J.A. Webb. The WARP computer: Architecture, implementation, and performance. *IEEE Transaction on Computers*, C-36(12):1523–1538, December 1987.

[2] Ramune Arlauskas. iPSC/2 system: A second generation hypercube. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, January 1988.

[3] David Callahan and Ken Kennedy. Compiling programs for distributed-memory multiprocessors. *The Journal of Supercomputing*, 2(2):151–170, 1988.

[4] Marina Chen, Young-il Choo, and Jingke Li. Compiling parallel programs by optimizing performance. *The Journal of Supercomputing*, 1(2):171–207, July 1988.

[5] M.R. Garey and D.S. Johnson. *Computers and intractability.* W.H. Freeman and Co., 1979.

[6] Ching-Tien Ho and S. Lennart Johnsson. Stable dimension permutations on Boolean cubes. Technical Report YALEU/DCS/RR-617, Department of Computer Science, Yale University, October 1988.

[7] C.A.R. Hoare. Communicating sequential processes. *CACM*, 21(8):666–677, 1978.

[8] Young il Choo and Marina Chen. A theory of parallel-program optimization. Technical Report YALEU/DCS/TR-608, Dept. of Computer Science, Yale University, July 1988.

[9] S. Lennart Johnsson and Ching-Tien Ho. Spanning graphs for optimum broadcasting and personalized communication in hypercubes. Technical Report YALEU/DCS/RR-610, Dept. of Computer Science, Yale University, November 1987.

[10] Eugene L Lawler. *Combinatorial Optimization: Networks and Metroids.* Holt, Rinehart and Winston, 1976.

[11] C. Leiserson, F. Rose, and J. Saxe. Optimizing synchronous circuitry by retiming. In *Third Caltech Conference on VLSI*, pages 87–116. Caltech, March 1983.

[12] Jingke Li. *Compiling Crystal for Hypercube Machines.* PhD thesis, Yale University, (In preparation).

[13] Ncube handbook. Technical report, NCUBE, 1986.

[14] Matthew Rosing and Robert B. Schnabel. An overview of dino – a new language for numerical computation on distributed memory multiprocessors. Technical Report CU-CS-385-88, University of Colorado, March 1988.

[15] P.S. Tseng, M. Lam, and H.T. Kung. The domain parallel computation model on WARP. In *Proceedings of SPIE Symposium.* Society of Photo-Optical Instrumentation Engineers, August 1988.

Appendix

Gaussian Elimination Program in Crystal

The program implements the standard Gaussian elimination algorithm. In the forward elimination phase, the program iterates over the columns of the input matrix. In iteration k a pivot element is chosen from the elements in column k at or below the diagonal (say element (j, k)) and rows k and j are exchanged. Then, the elements in the column below the diagonal are eliminated using the pivot element. In the back substitution phase, the resulting vector is obtained in n steps where n is the input matrix size.

! Index domains:

$\mathsf{dom}\ D = D_1 \times D_2 \times D_0,$

$\mathsf{dom}\ D_0 = [0..n],$

$\mathsf{dom}\ D_1 = [1..n],$

$\mathsf{dom}\ D_2 = [1..(n+1)],$

! Data fields (A0 and n are inputs):

! Forward elimination phase:

$$\mathsf{dfield}\ a(i,j,k) : D = \left\{ \begin{array}{l} k = 0 \rightarrow A0[i,j], \\ i < k \rightarrow a(i,j,k-1), \\ i = k \rightarrow a(ipivot(k), j, k-1), \\ i = ipivot(k) \rightarrow a(k,j,k-1) - a(i,j,k-1) * fac(i,k), \\ \mathsf{else} \rightarrow a(i,j,k-1) - a(ipivot(k),j,k-1) * fac(i,k) \end{array} \right\},$$

! Pivot elements:

$\mathsf{dfield}\ apivot(k) : D_1 = \backslash\mathsf{max}\ \{|a(i,k,k-1)| \mid k <= i <= n\},$

! Indices of pivot elements:

$\mathsf{dfield}\ ipivot(k) : D_1 = \backslash\mathsf{max}\ \{i \mid k <= i <= n : |a(i,k,k-1)| = apivot(k)\},$

! Pivoting factors:

$$\mathsf{dfield}\ fac(i,k) : D_1 \times D_1 = \left\{ \begin{array}{l} i <= k \rightarrow 1.0, \\ i = ipivot(k) \rightarrow \\ \quad a(k,k,k-1)/a(ipivot(k),k,k-1), \\ \mathsf{else} \rightarrow a(i,k,k-1)/a(ipivot(k),k,k-1) \end{array} \right\},$$

! Backward substitution phase:

$$\mathsf{dfield}\ x(i,j) : D_1 \times D_1 = \left\{ \begin{array}{l} i = j \rightarrow (a(i,n+1,n) - psum(i,j+1))/a(i,j,n), \\ i < j \rightarrow x(i+1,j) \end{array} \right\},$$

$$\mathsf{dfield}\ psum(i,j) : D_1 \times D_2 = \left\{ \begin{array}{l} j = n+1 \rightarrow 0.0, \\ i <= j \rightarrow psum(i,j+1) + a(i,j,n) * x(i,j) \end{array} \right\},$$

! Resulting vector:

$?[x(1,j) \mid 1 <= j <= n],$

NP-Completeness Result

We call an undirected graph G with even number of nodes a *multi-pair* graph if the nodes are grouped in pairs and no edge exits between the two nodes of the same pair. A simple example of a multi-pair graph is the new graph obtained by putting together two copies of a given graph and pairing the nodes according to the isomorphic relation between the two copies.

Consider the domain alignment problem where all domains are two dimensional. Its component alignment graph has the property that every column consists of exactly two nodes. Since no edge exists in any column of a CAG (due to the construction rule that self reference patterns are ignored), a CAG is a multi-pair graph.

The problem of finding the optimal alignment can now be defined as follows:

Component Alignment Problem (ALIGN).

Instance: A multi-pair graph $G = (V, E)$ and a positive integer K.

Question: Can V be decomposed into two equal-size sets V_A and V_B by putting one node of each pair into V_A and the other into V_B, such that the number of edges bridging V_A and V_B is $< K$?

To show this problem to be NP-complete, the most natural NP-complete problem to use seems to be the minimum cut into bounded sets (MINCUT) problem. However, there is a little problem in getting the reduction to work. When we put together two copies of a general undirected graph (an instance of MINCUT) to form a multi-pair graph (an instance of ALIGN), the latter is already in the optimal form with respect to the ALIGN problem. To resolve this problem, we introduce the following problem.

Dual component alignment problem (D-ALIGN).

Instance: A multi-pair graph $G = (V, E)$ and a positive integer K.

Question: Can V be decomposed into two equal-size sets V_A and V_B by putting one node of each pair into V_A and the other into V_B, such that the number of edges between V_A and V_B is $> K$?

Lemma The coordinate alignment problem and its dual problem are equivalent.

Proof: This is simply because we can define a dual graph for each multi-pair graph G in which non-edges become edges and edges become non-edges. A min-cut solution to G corresponds to a max-cut solution to the dual graph. Q.E.D.

Theorem The dual coordinate alignment problem is NP-complete.

Proof: The problem is obviously in NP. To show it is NP-hard, we reduce an NP-complete problem, the *simple maxcut problem* (MAXCUT), to D-ALIGN. We show that for each instance of MAXCUT, we can construct an instance for D-ALIGN such that a solution to the latter can be transformed into a solution to the former.

First, we construct an instance. Let $G = (V, E)$ be an undirected graph serving as an instance of MAXCUT. Create an isomorphic copy of G and call it $G' = (V', E')$. Let $\hat{G} = (V + V', E + E')$. Nodes in $V + V'$ are paired according to the isomorphic relation. $\hat{G}$ is a multi-pair graph and hence an instance of D-ALIGN.

Second, we convert a solution of D-ALIGN to that of MAXCUT. Suppose there is an algorithm solving D-ALIGN. Provide an input $\hat{G}$ to the algorithm, we get back a solution $(\hat{V_A}, \hat{V_B}, \hat{C})$, where $\hat{V_A}$ and $\hat{V_B}$ decompose the edge set of $\hat{G}$ and $\hat{C}$ is the set of edges which have one endpoint in $\hat{V_A}$ and one endpoint in $\hat{V_B}$.

The cut set $\hat{C}$ can be decomposed into two disjoint sets C and C', where C contains only edges from G and C' contains only edges from G'. Due to the isomorphic relation between G and G' and the rule of decomposition defined in D-ALIGN, we know that C and C' are isomorphic to each other. We claim that C is a maxcut (an edge set) of G. Assume it were not. Let C'' be a maxcut of G. Then $|C''| > |C|$. Let V_A'' and V_B'' be the decomposition of V corresponding to C''. For G', we have the isomorphic counterparts, C''', V_A''' and V_B'''. Now that $V_A'' + V_B'''$, $V_A''' + V_B''$ and $C'' + C'''$ define a solution to D-ALIGN. But $|C'' + C'''| > |C + C'|$. This contradicts the result that $\hat{V_A}$, $\hat{V_B}$ and $\hat{C}$ is a solution to D-ALIGN. Q.E.D.