Solving Alignment Using Elementary Linear Algebra

Vladimir Kotlyar, David Bau, Induprakas Kodukula, Keshav Pingali, and Paul Stodghill

Department of Computer Science
Cornell University
Ithaca NY 14853
USA

Summary. Data and computation alignment is an important part of compiling sequential programs to architectures with non-uniform memory access times. In this paper, we show that elementary matrix methods can be used to determine communication-free alignment of code and data. We also solve the problem of replicating data to eliminate communication. Our matrix-based approach leads to algorithms which work well for a variety of applications, and which are simpler and faster than other matrix-based algorithms in the literature.

1. Introduction

A key problem in generating code for non-uniform memory access (NUMA) parallel machines is data and computation placement — that is, determining what work each processor must do, and what data must reside in each local memory. The goal of placement is to exploit parallelism by spreading the work across the processors, and to exploit locality by spreading data so that memory accesses are local whenever possible. The problem of determining a good placement for a program is usually solved in two phases called *alignment* and *distribution*. The alignment phase maps data and computations to a set of virtual processors organized as a Cartesian grid of some dimension (a *template* in HPF Fortran terminology). The distribution phase folds the virtual processors into the physical processors. The advantage of separating alignment from distribution is that we can address the collocation problem (determining which iterations and data should be mapped to the same processor) without worrying about the load balancing problem.

Our focus in this paper is alignment. A complete solution to this problem can be obtained in three steps.

1. Determine the constraints on data and computation placement.
2. Determine which constraints should be left unsatisfied.
3. Solve the remaining system of constraints to determine data and computation placement.

[1] An earlier version of this paper was presented in the 7th Annual Workshop on Languages and Compilers for Parallel Computers (LCPC), Ithaca, 1994.

In the first step, data references in the program are examined to determine a system of equations in which the unknowns are functions representing data and computation placements. Any solution to this system of equations determines a so-called *communication-free* alignment [6] — that is, a map of data elements and computations to virtual processors such that all data required by a processor to execute the iterations mapped to it are in its local memory. Very often, the only communication-free alignment for a program is the trivial one in which every iteration and datum is mapped to a single processor. Intuitively, each equation in the system is a constraint on data and computation placement, and it is possible to overconstrain the system so that the trivial solution is the only solution. If so, the second step of alignment determines which constraints must be left unsatisfied to retain parallelism in execution. The cost of leaving a constraint unsatisfied is that it introduces communication; therefore, the constraints left unsatisfied should be those that introduce as little communication as possible. In the last step, the remaining constraints are solved to determine data and computation placement.

The following loop illustrates these points. It computes the product Y of a sub-matrix $A(11:N+10, 11:N+10)$ and a vector X:

```
DO i=1,N
  DO j=1,N
    Y(i) = Y(i) + A(i+10,j+10)*X(j)
```

For simplicity, assume that the virtual processors are organized as a one-dimensional grid $\mathcal{T}$. Let us assume that computations are mapped by iteration number — that is, a processor does all or none of the work in executing an iteration of the loop. To avoid communication, the processor that executes iteration (i,j) must have $A(i+10, j+10)$, $Y(i)$ and $X(j)$ in its local memory. These constraints can be expressed formally by defining the following functions that map loop iterations and array elements to virtual processors:

$$\begin{array}{lcll}
\mathbf{C} & : & (i,j) \to \mathcal{T} & \text{processor that performs iteration } (i,j) \\
\mathbf{D}_A & : & (i,j) \to \mathcal{T} & \text{processor that owns } A(i,j) \\
\mathbf{D}_Y & : & i \to \mathcal{T} & \text{processor that owns } Y(i) \\
\mathbf{D}_X & : & j \to \mathcal{T} & \text{processor that owns } X(j)
\end{array}$$

The constraints on these functions are the following.

$$\forall i,j \text{ s.t. } 1 \le i,j \le N \ : \left\{ \begin{array}{l} \mathbf{C}(i,j) = \mathbf{D}_A(i+10, j+10) \\ \mathbf{C}(i,j) = \mathbf{D}_Y(i) \\ \mathbf{C}(i,j) = \mathbf{D}_X(j) \end{array} \right.$$

If we enforce all of the constraints, the only solution is the trivial solution in which all data and computations are mapped to a single processor. In this case, we say that our system is *overconstrained.* If we drop the constraint on X, we have a non-trivial solution to the resulting system of constraints, which maps iteration (i,j) to processor i, and maps array elements $A(i+10, j+10)$, $X(i)$ and $Y(i)$ to processor i. Note that all these maps are affine functions

— for example, the map of array A to the virtual processors can be written as follows:

$$\mathbf{D}_A(a,b) = \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} - \begin{pmatrix} 10 \\ 10 \end{pmatrix} \tag{1.1}$$

Since there is more than one processor involved in the computation, we have parallel execution of the program. However, elements of X must be communicated at runtime.

In this example, the solution to the alignment equations was determined by inspection, but how does one solve such systems of equations in general? Note that the unknowns are general functions, and that each function may be constrained by several equations (as is the case for $\mathbf{C}$ in the example). To make the problem tractable, it is standard to restrict the maps to linear (or affine) functions of loop indices. This restriction is not particularly onerous in general – in fact, it permits more general maps of computation and data than are allowed in HPF. The unknowns in the equations now become matrices, rather than general functions, but it is still not obvious how such systems of matrix equations can be solved. In Section 2., we introduce our linear algebraic framework that reduces the problem of solving systems of alignment equations to the standard linear algebra problem of determining a basis for the null space of a matrix. One weakness of existing approaches to alignment is that they handle only *linear* functions; general affine functions, like the map of array A, must be dealt with in *ad hoc* ways. In Section 3., we show that our framework permits affine functions to be handled without difficulty.

In some programs, replication of arrays is useful for exploiting parallelism. Suppose we wanted to parallelize all iterations of our matrix-vector multiplication loop. The virtual processor (i,j) would execute the iteration (i,j) and own the array element $A(i+10, j+10)$. It would also require the array element $X(j)$. This means that we have to replicate the array X along the i dimension of the virtual processor grid. In addition, element $Y(i)$ must be computed by reducing (adding) values computed by the set of processors $(i,*)$. In Section 4., we show that our framework permits a solution to the replication/reduction problem as well.

Finally, we give a systematic procedure for dropping constraints from overconstrained systems. Finding an optimal solution that trades off parallelism for communication is very difficult. First, it is hard to model accurately the cost of communication and the benefit of parallelism. For example, parallel matrix-vector product is usually implemented either by mapping rows of the matrix to processors (so-called 1-D alignment) or by mapping general submatrices to processors (so-called 2-D alignment). Which mapping is better depends very much on the size of the matrix, and on the communication to computation speed ratio of the machine [9]. Second, even for simple parallel models and restricted cases of the alignment problem, finding the optimal solution is known to be NP-complete problem [10]. Therefore, we must fall back on heuristics. In Section 5., we discuss our heuristic. Not surprisingly, our heuristic is skewed to "do the right thing" for kernels like matrix-vector product which are extremely important in practice.

How does our work relate to previous work on alignment? Our work is closest in spirit to that of Huang and Sadayappan who were the first to formulate the problem of communication-free alignment in terms of systems of equational constraints [6]. However, they did not give a general method for solving these equations. Also, they did not handle replication of data. Anderson and Lam sketched a solution method [1], but their approach is unnecessarily complex, requiring the determination of cycles in bipartite graphs, computing pseudo-inverses *etc* – these complications are eliminated by our approach.

The equational, matrix-based approach described in this paper is not the only approach that has been explored. Li and Chen have used graph-theoretic methods to trade off communication for parallelism for a limited kind of alignment called *axis alignment*[10]. More general heuristics for a wide variety of cost-of-communication metrics have been studied by Chatterjee, Gilbert and Schreiber[2, 3], Feautrier [5] and Knobe et al [8, 7].

To summarize, the contributions of this paper are the following.

1. We show that the problem of determining communication-free partitions of computation and data can be reduced to the standard linear algebra problem of determining a basis for the null space of a matrix , which can be solved using fairly standard techniques (Section 2.2).
2. Previous approaches to alignment handle linear maps, but deal with affine maps in fairly *ad hoc* ways. We show that affine maps can be folded into our framework without difficulty (Section 3.).
3. We show how replication of arrays is handled by our framework (Section 4.).
4. We suggest simple and effective heuristic strategies for deciding when communication should be introduced (Section 5.).

2. Linear Alignment

To avoid introducing too many ideas at once, we restrict attention to linear subscripts and linear maps in this section. First, we show that the alignment problem can be formulated using systems of equational constraints. Then, we show that the problem of solving these systems of equations can be reduced to the standard problem of determining a basis for the null space of a matrix, which can be solved using integer-preserving Gaussian elimination.

2.1 Equational Constraints

The equational constraints for alignment are simply a formalization of an intuitively reasonable statement: 'to avoid communication, the processor that performs an iteration of a loop nest must own the data referenced in that iteration'. We discuss the formulation of these equations in the context of the following example:

```
DO j=1,100
  DO k=1,100
    B(j,k) = A(j,k) + A(k,j)
```

If $\mathbf{i}$ is an iteration vector in the iteration space of the loop, the alignment constraints require that the processor that performs iteration $\mathbf{i}$ must own $B(\mathbf{F}_1\mathbf{i})$, $A(\mathbf{F}_1\mathbf{i})$ and $A(\mathbf{F}_2\mathbf{i})$, where $\mathbf{F}_1$ and $\mathbf{F}_2$ are the following matrices:

$$\mathbf{F}_1 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \qquad \mathbf{F}_2 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

Let $\mathbf{C}$, $\mathbf{D}_A$ and $\mathbf{D}_B$ be $p \times 2$ matrices representing the maps of the computation and arrays A and B to a p-dimensional processor template; p is an unknown which will be determined by our algorithm. Then, the alignment problem can be expressed as follows: find $\mathbf{C}$, $\mathbf{D}_A$ and $\mathbf{D}_B$ such that

$$\forall\ \mathbf{i} \in \ \text{iteration space of loop}: \begin{cases} \mathbf{Ci} = \mathbf{D}_B\mathbf{F}_1\mathbf{i} \\ \mathbf{Ci} = \mathbf{D}_A\mathbf{F}_1\mathbf{i} \\ \mathbf{Ci} = \mathbf{D}_A\mathbf{F}_2\mathbf{i} \end{cases}$$

To 'cancel' the $\mathbf{i}$ on both sides of each equation, we will simplify the problem and require that the equations hold for all 2-dimensional integer vectors, regardless of whether they are in the bounds of the loop or not. In that case, the constraints simply become equations involving matrices, as follows: find $\mathbf{C}$, $\mathbf{D}_A$ and $\mathbf{D}_B$ such that

$$\begin{cases} \mathbf{C} &= \mathbf{D}_B\mathbf{F}_1 \\ \mathbf{C} &= \mathbf{D}_A\mathbf{F}_1 \\ \mathbf{C} &= \mathbf{D}_A\mathbf{F}_2 \end{cases} \tag{2.1}$$

We will refer to the equation scheme $\mathbf{C} = \mathbf{DF}$ as the fundamental equation of alignment.

The general principle behind the formulation of alignment equations should be clear from this example. Each data reference for which alignment is desired gives rise to an alignment equation. Data references for which subscripts are not linear functions of loop indices are ignored; therefore, such references may give rise to communication at runtime. Although we have discussed only a single loop nest, it is clear that this framework of equational constraints can be used for multiple loop nests as well. The equational constraints from each loop nest are combined to form a single system of simultaneous equations, and the entire system is solved to find communication-free maps of computations and data.

2.2 Reduction to Null Space Computation

One way to solve systems of alignment equations is to set $\mathbf{C}$ and $\mathbf{D}$ matrices to the zero matrix of some dimension. This is the trivial solution in which all computations and data are mapped to a single processor, processor 0. This solution exploits no parallelism; therefore, we want to determine a non-trivial solution if it exists. We do this by reducing the problem to the standard linear algebra problem of determining a basis for the null space of a matrix.

Consider a single equation.

$$\mathbf{C} \quad = \quad \mathbf{DF}$$

This equation can be written in block matrix form as follows:

$$\begin{pmatrix} \mathbf{C} & \mathbf{D} \end{pmatrix} \begin{pmatrix} \mathbf{I} \\ -\mathbf{F} \end{pmatrix} \quad = \quad \mathbf{0}$$

Now it is of the form $\mathbf{UV} = 0$ where $\mathbf{U}$ is an unknown matrix and $\mathbf{V}$ is a known matrix. To see the connection with null spaces, we take the transpose of this equation and we see that this is the same as the equation $\mathbf{V}^T\mathbf{U}^T = 0$. Therefore, $\mathbf{U}^T$ is a matrix whose columns are in the null space of $\mathbf{V}^T$. To exploit parallelism, we would like the rank of $\mathbf{U}^T$ to be as large as possible. Therefore, we must find a basis for the null space of matrix $\mathbf{V}^T$. This is done using integer-preserving Gaussian elimination, a standard algorithm in the literature [4, 12].

The same reduction works in the case of multiple constraints. Suppose that there are s loops and t arrays. Let the computation maps of the loops be $\mathbf{C}_1, \mathbf{C}_2, \ldots, \mathbf{C}_s$, and the array maps be $\mathbf{D}_1, \mathbf{D}_2, \ldots, \mathbf{D}_t$. We can construct a block row with all the unknowns as follows:

$$\mathbf{U} \quad = \quad \begin{pmatrix} \mathbf{C}_1 & \mathbf{C}_2 & \ldots & \mathbf{C}_s & \mathbf{D}_1 & \ldots & \mathbf{D}_t \end{pmatrix}$$

For each constraint of the form $\mathbf{C}_j = \mathbf{D}_k\mathbf{F}_\ell$, we create a block column:

$$\mathbf{V}_q \quad = \quad \begin{pmatrix} \mathbf{0} \\ \mathbf{I} \\ \mathbf{0} \\ -\mathbf{F}_\ell \\ \mathbf{0} \end{pmatrix}$$

where the zeros are placed so that:

$$\mathbf{UV}_q \quad = \quad \mathbf{C}_j - \mathbf{D}_k\mathbf{F}_\ell \tag{2.2}$$

Putting all these block columns into a single matrix $\mathbf{V}$, the problem of finding communication-free alignment reduces once again to a matrix equation of the form

$$\mathbf{UV} \quad = \quad 0 \tag{2.3}$$

The reader can verify that the system of equations (2.1) can be converted into the following matrix equation:

Input:. A set of alignment constraints of the form $\mathbf{C}_j = \mathbf{D}_k\mathbf{F}_\ell$.
Output:. Communication-free alignment matrices $\mathbf{C}_j$ and $\mathbf{D}_k$.

1. Assemble block columns as in (2.2).
2. Put all block columns $\mathbf{V}_q$ into one matrix $\mathbf{V}$.
3. Compute a basis $\mathbf{U}^T$ for the null space of $\mathbf{V}^T$.
4. Set template dimension to number of rows of $\mathbf{U}$.
5. Extract the solution matrices $\mathbf{C}_j$ and $\mathbf{D}_k$ from $\mathbf{U}$.
6. Reduce the solution matrix $\mathbf{U}$ as described in Section 2.4.

Fig. 2.1. Algorithm `LINEAR-ALIGNMENT`.

$$\begin{pmatrix} \mathbf{C} & \mathbf{D}_A & \mathbf{D}_B \end{pmatrix} \begin{pmatrix} \mathbf{I} & \mathbf{I} & \mathbf{I} \\ \mathbf{0} & -\mathbf{F}_1 & -\mathbf{F}_2 \\ -\mathbf{F}_1 & \mathbf{0} & \mathbf{0} \end{pmatrix} = \mathbf{0} \tag{2.4}$$

A solution matrix is:

$$\mathbf{U} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \tag{2.5}$$

which gives us:

$$\mathbf{C} = \mathbf{D}_A = \mathbf{D}_B = \begin{pmatrix} 1 & 1 \end{pmatrix} \tag{2.6}$$

Since the number of rows of $\mathbf{U}$ is one, the solution requires a one dimensional template. Iteration (i, j) is mapped to processor $i + j$. Arrays A and B are mapped identically so that the 'anti-diagonals' of these matrices are mapped to the same processor.

The general algorithm is outlined in Figure 2.1.

2.3 Remarks

Our framework is robust enough that we can add additional constraints to computation and data maps without difficulty. For example, if a loop in a loop nest carries a dependence, we may not want to spread iterations of that loop across processors. More generally, dependence information can be characterized by a distance vector $\mathbf{z}$, which for our purposes says that iterations $\mathbf{i}$ and $\mathbf{i}+\mathbf{z}$ have to be executed on the same processor. In terms of our alignment model:

$$\mathbf{Ci} + \mathbf{b} \quad = \quad \mathbf{C}(\mathbf{i} + \mathbf{z}) + \mathbf{b} \quad \Leftrightarrow \quad \mathbf{Cz} \quad = \quad \mathbf{0} \tag{2.7}$$

We can now easily incorporate (2.7) into our matrix system (2.3) by adding the following block column to $\mathbf{V}$:

$$\mathbf{V}_{dep} \quad = \quad \begin{pmatrix} \mathbf{0} \\ \mathbf{z} \\ \mathbf{0} \end{pmatrix}$$

where zeros are placed to that $\mathbf{UV}_{dep} = \mathbf{Cz}$. Adding this column to $\mathbf{V}$ will ensure that any two dependent iterations end up on the same processor.

In some circumstances, it may be necessary to align two data references without aligning them with any computation. This gives rise to equations of the form $\mathbf{D_1F_1} = \mathbf{D_2F_2}$. Such equations can be incorporated into our framework by adding block columns of the form

$$\mathbf{V}_p = \begin{pmatrix} \mathbf{0} \\ \mathbf{F_1} \\ \mathbf{0} \\ -\mathbf{F_2} \\ \mathbf{0} \end{pmatrix} \tag{2.8}$$

where the zeros are placed so that $\mathbf{UV}_p = \mathbf{D_1F_1} - \mathbf{D_2F_2}$.

2.4 Reducing the Solution Basis

Finally, one practical note. It is possible for Algorithm `LINEAR-ALIGNMENT` to produce a solution $\mathbf{U}$ which has p rows, even though all $\mathbf{C}_j$ produced by Step 5 have rank less than p. A simple example where this can happen is a program with two loop nests which have no data in common. Mapping the solution into a lower dimensional template can be left to the distribution phase of compiling; alternatively, an additional step can be added to Algorithm `LINEAR-ALIGNMENT` to solve this problem directly in the alignment phase. This modification is described next.

Suppose we compute a solution which contains two computation alignments:

$$\mathbf{U} = \begin{pmatrix} \mathbf{C}_1 & \mathbf{C}_2 & \dots \end{pmatrix} \tag{2.9}$$

Let r be the number of rows in $\mathbf{U}$. Let r_1 be the rank of $\mathbf{C}_1$, and let r_2 be the rank of $\mathbf{C}_2$. Assume that $r_1 < r_2$. We would like to have a solution basis where the first r_1 rows of $\mathbf{C}_1$ are linearly independent, as are the first r_2 rows of $\mathbf{C}_2$ — that way, if we decide to have an r_1-dimensional template, we are guaranteed to keep r_1 degrees of parallelism for the second loop nest, as well.

Mathematically, the problem is to find a sequence of row transformations $\mathbf{T}$ such that the first r_1 rows of $\mathbf{TC}_1$ are linearly independent and so are the first r_2 rows of $\mathbf{TC}_2$.

A detailed procedure is given in the appendix. Here, we describe the intuitive idea. Suppose that we have already arranged the first r_1 rows of $\mathbf{C}_1$ to be linearly independent. Inductively, assume that the first $k < r_2$ rows of $\mathbf{C}_2$ are linearly independent as well. We want to make the $k+1$-st row of $\mathbf{C}_2$ linearly independent of the previous k rows. If it already is, we go the next row. If not, then there must be a row $m > k+1$ of $\mathbf{C}_2$ which is linearly independent of the first k rows. It is easy to see that if we add the m-th row to the $k+1$-st row, we will make the latter linearly independent of the first k rows. Notice that this can mess up $\mathbf{C}_1$! Fortunately, it can be shown that if we add a suitably large multiple of the m-th row, we can be sure that the first r_1 rows of $\mathbf{C}_1$ remain independent. This algorithm can be easily generalized to any number of $\mathbf{C}_i$ blocks.

3. Affine Alignment

In this section, we generalize our framework to affine functions. The intuitive idea is to 'encode' affine subscripts as linear subscripts by using an extra dimension to handle the constant term. Then, we apply the machinery in Section 2. to obtain linear computation and data maps. The extra dimension can be removed from these linear maps to 'decode' them back into affine maps.

We first generalize the data access functions $\mathbf{F}_i$ so that they are affine functions of the loop indices. In the presence of such subscripts, aligning data and computation requires affine data and computation maps. Therefore, we introduce the following notation.

$$\begin{aligned} \text{Computation maps:} \quad & C_j(\mathbf{i}) = \mathbf{C}_j\mathbf{i} + \mathbf{c}_j && (3.1) \\ \text{Data maps:} \quad & D_k(\mathbf{a}) = \mathbf{D}_k\mathbf{a} + \mathbf{d}_k && (3.2) \\ \text{Data access functions:} \quad & F_\ell(\mathbf{i}) = \mathbf{F}_\ell\mathbf{i} + \mathbf{f}_\ell && (3.3) \end{aligned}$$

$\mathbf{C}_j$, $\mathbf{D}_k$ and $\mathbf{F}_\ell$ are matrices representing the linear parts of the affine functions, while $\mathbf{c}_j$, $\mathbf{d}_k$ and $\mathbf{f}_\ell$ represent constants. The alignment constraints from each reference are now of the form

$$\forall \mathbf{i} \in \mathbb{Z}^n \quad : \quad \mathbf{C}_j\mathbf{i} + \mathbf{c}_j = \mathbf{D}_k(\mathbf{F}_\ell\mathbf{i} + \mathbf{f}_\ell) + \mathbf{d}_k \tag{3.4}$$

3.1 Encoding affine constraints as linear constraints

Affine functions can be encoded as linear functions by using the following identity.

$$\mathbf{T}\mathbf{x} + \mathbf{t} \quad = \quad \begin{pmatrix} \mathbf{T} & \mathbf{t} \end{pmatrix} \begin{pmatrix} \mathbf{x} \\ 1 \end{pmatrix} \tag{3.5}$$

where $\mathbf{T}$ is a matrix, and $\mathbf{t}$ and $\mathbf{x}$ are vectors. We can put (3.4) in the form:

$$\begin{aligned} \begin{pmatrix} \mathbf{C}_j & \mathbf{c}_j \end{pmatrix} \begin{pmatrix} \mathbf{i} \\ 1 \end{pmatrix} &= \mathbf{D}_k \begin{pmatrix} \mathbf{F}_\ell & \mathbf{f}_\ell \end{pmatrix} \begin{pmatrix} \mathbf{i} \\ 1 \end{pmatrix} + \mathbf{d}_k \\ &= \begin{pmatrix} \mathbf{D}_k & \mathbf{d}_k \end{pmatrix} \begin{pmatrix} \mathbf{F}_\ell & \mathbf{f}_\ell \\ \mathbf{0} & 1 \end{pmatrix} \begin{pmatrix} \mathbf{i} \\ 1 \end{pmatrix} && (3.6) \end{aligned}$$

Now we let:

$$\hat{\mathbf{C}}_j = \begin{pmatrix} \mathbf{C}_j & \mathbf{c}_j \end{pmatrix} \qquad \hat{\mathbf{D}}_k = \begin{pmatrix} \mathbf{D}_k & \mathbf{d}_k \end{pmatrix} \qquad \hat{\mathbf{F}}_\ell = \begin{pmatrix} \mathbf{F}_\ell & \mathbf{f}_\ell \\ \mathbf{0} & 1 \end{pmatrix} \tag{3.7}$$

(3.6) can be written as:

$$\forall \mathbf{i} \in \mathbb{Z}^d \quad : \quad \hat{\mathbf{C}}_j \begin{pmatrix} \mathbf{i} \\ 1 \end{pmatrix} = \hat{\mathbf{D}}_k \hat{\mathbf{F}}_\ell \begin{pmatrix} \mathbf{i} \\ 1 \end{pmatrix} \tag{3.8}$$

As before, we would like to 'cancel' the vector $\begin{pmatrix} \mathbf{i} \\ 1 \end{pmatrix}$ from both sides of the equation. To do this, we need the following result.

Lemma 3.1. *Let* $\mathbf{T}$ *be a matrix,* $\mathbf{t}$ *a vector. Then*

$$\forall \mathbf{x} \begin{pmatrix} \mathbf{T} & \mathbf{t} \end{pmatrix} \begin{pmatrix} \mathbf{x} \\ 1 \end{pmatrix} = \mathbf{0}$$

if and only if $\mathbf{T} = \mathbf{0}$ *and* $\mathbf{t} = \mathbf{0}$.

Proof:. In particular, we can let $\mathbf{x} = \mathbf{0}$. This gives us:

$$\begin{pmatrix} \mathbf{T} & \mathbf{t} \end{pmatrix} \begin{pmatrix} \mathbf{0} \\ 1 \end{pmatrix} \;=\; \mathbf{t} \;=\; \mathbf{0}$$

So $\mathbf{t} = \mathbf{0}$. Now, for any $\mathbf{x}$:

$$\begin{pmatrix} \mathbf{T} & \mathbf{t} \end{pmatrix} \begin{pmatrix} \mathbf{x} \\ 1 \end{pmatrix} \;=\; \begin{pmatrix} \mathbf{T} & \mathbf{0} \end{pmatrix} \begin{pmatrix} \mathbf{x} \\ 1 \end{pmatrix} \;=\; \mathbf{Tx} \;=\; \mathbf{0}$$

which means that $\mathbf{T} = \mathbf{0}$, as well. $\square$.

Using Lemma 3.1, we can rewrite (3.8) as follows:

$$\hat{\mathbf{C}}_j \;=\; \hat{\mathbf{D}}_k \hat{\mathbf{F}}_\ell \tag{3.9}$$

We can now use the techniques in Section 2. to reduce systems of such equations to a single matrix equation as follows:

$$\hat{\mathbf{U}}\hat{\mathbf{V}} \;=\; \mathbf{0} \tag{3.10}$$

In turn, this equation can be solved using the Algorithm `LINEAR-ALIGNMENT` to determine $\hat{\mathbf{U}}$. To illustrate this process, we use the example from Section 1.:

```
DO i=1,N
  DO j=1,N
    Y(i) = Y(i) + A(i+10,j+10)*X(j)
```

Suppose we wish to satisfy the constraints for Y and A. The relevant array access functions are:

$$\begin{array}{lllllll} \mathbf{F}_A & = & \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} & \mathbf{f}_A & = & \begin{pmatrix} 10 \\ 10 \end{pmatrix} \\ \mathbf{F}_Y & = & \begin{pmatrix} 1 & 0 \end{pmatrix} & \mathbf{f}_Y & = & \begin{pmatrix} 0 \end{pmatrix} \\ \hat{\mathbf{F}}_A & = & \begin{pmatrix} 1 & 0 & 10 \\ 0 & 1 & 10 \\ 0 & 0 & 1 \end{pmatrix} & \hat{\mathbf{F}}_Y & = & \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \end{array} \tag{3.11}$$

The reader can verify that the matrix equation to be solved is the following one.

$$\hat{\mathbf{U}}\hat{\mathbf{V}} \;=\; \mathbf{0} \tag{3.12}$$

where:

$$\hat{\mathbf{U}} = \begin{pmatrix} \hat{\mathbf{C}} & \hat{\mathbf{D}}_A & \hat{\mathbf{D}}_Y \end{pmatrix} \qquad \hat{\mathbf{V}} = \begin{pmatrix} \mathbf{I} & \mathbf{I} \\ -\hat{\mathbf{F}}_A & \mathbf{0} \\ \mathbf{0} & -\hat{\mathbf{F}}_Y \end{pmatrix}$$

And the solution is the following matrix.

$$\hat{\mathbf{U}} = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & -10 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \end{pmatrix} \tag{3.13}$$

From this matrix, we can read off the following maps of computation and data.

$$\hat{\mathbf{C}} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad \hat{\mathbf{D}}_A = \begin{pmatrix} 1 & 0 & -10 \\ 0 & 0 & 1 \end{pmatrix}$$
$$\hat{\mathbf{D}}_Y = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

This says that iteration i of the loop and element $X(i)$ are mapped to the following virtual processor.

$$\hat{\mathbf{C}}\begin{pmatrix} \mathbf{i} \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} i \\ j \\ 1 \end{pmatrix} = \begin{pmatrix} i \\ 1 \end{pmatrix}$$

Notice that although the space of virtual processors has two dimensions (because of the encoding of constants), the maps of the computation and data use only a one-dimensional subspace of the virtual processor space. To obtain a clean solution, it is desirable to remove the extra dimension introduced by the encoding.

We have already mentioned that there is always a trivial solution that maps everything to the same virtual processor $p = 0$. Because we have introduced affine functions, it is now possible to map everything to the same virtual processor $p \neq 0$. In our framework it is reflected in the fact that there is always a row

$$\mathbf{w}^T = \begin{pmatrix} 0 & 0 & \ldots & 0 & 1 & 0 & \ldots & 0 & 1 & \ldots & 0 & 0 & 1 \end{pmatrix} \tag{3.14}$$

(with zeros placed appropriately) in the *row space* of the solution matrix $\hat{\mathbf{U}}$. To "clean up" the solution notice that we can always find a vector $\mathbf{x}$ such that $\mathbf{x}^T\hat{\mathbf{U}} = \mathbf{w}^T$. Moreover, let k be the position of some non-zero element in $\mathbf{x}$ and let $\mathbf{J}$ be an identity matrix with the k-th row replaced by $\mathbf{x}^T$ ($\mathbf{J}$ is non-singular). Then the k-th row of $\hat{\mathbf{U}}' = \mathbf{J}\hat{\mathbf{U}}$ is equal to $\mathbf{w}^T$ and is linearly independent from the rest of the rows. This means that we can safely remove it from the solution matrix. Notice that this procedure is exactly equivalent to removing k-th row from $\hat{\mathbf{U}}$. A more detailed description is given in the appendix.

Algorithm `AFFINE-ALIGNMENT` is summarized in Figure 3.1.

Input:. A set of alignment constraints as in Equation (3.4).
Output:. Communication-free alignment mappings characterized by $\mathbf{C}_j$, $\mathbf{c}_j$, $\mathbf{D}_k$, $\mathbf{d}_k$.

1. Assemble $\hat{\mathbf{F}}_\ell$ matrices as in Equation 3.6.
2. Assemble block columns $\mathbf{V}_q$ as in Equation (2.2) using $\hat{\mathbf{F}}_\ell$ instead of $\mathbf{F}_\ell$.
3. Put all block columns $\mathbf{V}_q$ into one matrix $\hat{\mathbf{V}}$.
4. Compute a basis $\hat{\mathbf{U}}^T$ for null-space of $\hat{\mathbf{V}}^T$ as in the Step 3 of `LINEAR-ALIGNMENT` algorithm.
5. Eliminate redundant row(s) in $\hat{\mathbf{U}}$.
6. Extract the solution matrices from $\hat{\mathbf{U}}$.

Fig. 3.1. Algorithm `AFFINE-ALIGNMENT`.

4. Replication

As we discussed in Section 1., communication-free alignment may require replication of data. Currently, we allow replication only of read-only arrays or of the arrays which are updated using reduction operations. In this section, we show how replication of data is handled in our linear algebra framework. We use a matrix-vector multiplication loop (MVM) as a running example.

```
DO i=1,N
  DO j=1,N
    Y(i) = Y(i) + A(i,j)*X(j)
```

We are interested in deriving the parallel version of this code which uses 2-D alignment — that is, it uses a 2-dimensional template in which processor (i, j) performs iteration `(i,j)`. If we keep the alignment constraint for A only, we get the solution:

$$\mathbf{C} = \mathbf{D}_A = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \tag{4.1}$$

which means that iteration `(i,j)` is executed on the processor with coordinates (i, j). This processor also owns the array element $A(i, j)$. For the computation, it needs $X(j)$ and $Y(i)$. This requires that X be replicated along the i dimension of the processor grid, and Y be reduced along the j dimension. We would like to derive this information automatically.

4.1 Formulation of replication

To handle replication, we associate a pair of matrices $\mathbf{R}$ and $\mathbf{D}$ with each data reference for which alignment is desired; as we show next, the fundamental equational scheme for alignment becomes $\mathbf{RC} = \mathbf{DF}$.

Up to this point, data alignment was specified using a matrix $\mathbf{D}$ which mapped array element $\mathbf{a}$ to logical processor $\mathbf{Da}$. If $\mathbf{D}$ has a non-trivial null-space, then elements of the array belonging to the same coset of the null-space get placed onto the same virtual processor; that is,

$$\begin{array}{rcl} \mathbf{Da}_1 & = & \mathbf{Da}_2 \\ & \Leftrightarrow & \\ \mathbf{a}_1 - \mathbf{a}_2 & \in & \text{null}(\mathbf{D}) \end{array}$$

When we allow replication, the mapping of array elements to processors can be described as follows. Array element $\mathbf{a}$ is mapped to processor $\mathbf{p}$ if

$$\mathbf{Rp} \;=\; \mathbf{Da}$$

The mapping of the array is now a many-to-many relation that can be described in words as follows:

- Array elements that belong to the same coset of null($\mathbf{D}$) are mapped onto the same processors.
- Processors that belong to the same coset of null($\mathbf{R}$) own the same data.

From this, it is easy to see that the fundamental equation of alignment becomes $\mathbf{RC} = \mathbf{DF}$. The replication-free scenario is just a special case when $\mathbf{R}$ is $\mathbf{I}$. Not all arrays in a procedure need to be replicated — for example, if an array is involved in a non-reduction dependence or it is very large, we can disallow replication of that array. Notice that the equation $\mathbf{RC} = \mathbf{DF}$ is *non-linear* if both $\mathbf{R}$ and $\mathbf{C}$ are unknown. To make the solution tractable, we first compute $\mathbf{C}$ based on the constraints for the non-replicated arrays. Once $\mathbf{C}$ is determined, the equation is again linear in the unknowns $\mathbf{R}$ and $\mathbf{D}$. Intuitively, this means that we first drop some constraints from the non-replicated alignment system, and then try to satisfy these constraints via replication.

We need to clarify what "fixing $\mathbf{C}$" means. When we solve the alignment system (2.3), we obtain a basis $\mathbf{C}_{\text{basis}}$ for all solutions to the loop alignment. The solutions can be expressed parametrically as

$$\mathbf{C} = \mathbf{TC}_{\text{basis}} \tag{4.2}$$

for any matrix $\mathbf{T}$. Now the replication equation becomes

$$\mathbf{RTC}_{\text{basis}} = \mathbf{DF} \tag{4.3}$$

and we are faced again with a non-linear system ($\mathbf{T}$ is another unknown)! The key observation is that if we are considering a single loop nest, then $\mathbf{T}$ becomes redundant since we can "fold it" into $\mathbf{R}$. This lets us solve the replication problem for a single loop nest.

In our MVM example, once the loop alignment has been fixed as in (4.1), the system of equations for the replication of X and Y is:

Input:. Replication constraints of the form $\mathbf{RC} = \mathbf{DF}$.
Output:. Matrices $\mathbf{R}$, $\mathbf{D}$ and $\mathbf{C}_{basis}$ that specify alignment with replication.
1. Find $\mathbf{C}_{basis}$ by solving the alignment system for the non-replicated arrays using the Algorithm `AFFINE-ALIGNMENT`. If all arrays in the loop nest are allowed to be replicated, then set $\mathbf{C}_{basis} = \mathbf{I}$.
2. Find $(\mathbf{R}, \mathbf{D})$ pairs that specify replication by solving the $\mathbf{RC}_{basis} = \mathbf{DF}$ equations.

Fig. 4.1. Algorithm `SINGLE-LOOP-REPLICATION-ALIGNMENT`.

$$\begin{aligned} \mathbf{R}_X\mathbf{C} &= \mathbf{D}_X\mathbf{F}_X \\ \mathbf{R}_Y\mathbf{C} &= \mathbf{D}_Y\mathbf{F}_Y \end{aligned}$$

These can be solved independently or put together into a block-matrix form $\mathbf{UV} = \mathbf{0}$:

$$\begin{aligned} \mathbf{U} &= \begin{pmatrix} \mathbf{R}_X & \mathbf{R}_Y & \mathbf{D}_X & \mathbf{D}_Y \end{pmatrix} \\ \mathbf{V} &= \begin{pmatrix} \mathbf{C} & \mathbf{0} \\ \mathbf{0} & \mathbf{C} \\ \mathbf{D}_X & \mathbf{0} \\ \mathbf{0} & \mathbf{D}_Y \end{pmatrix} \end{aligned} \tag{4.4}$$

and solved using the standard methods. The solution to this system:

$$\begin{aligned} \mathbf{R}_X &= \begin{pmatrix} 0 & 1 \end{pmatrix} & \mathbf{D}_X &= \begin{pmatrix} 1 \end{pmatrix} \\ \mathbf{R}_Y &= \begin{pmatrix} 1 & 0 \end{pmatrix} & \mathbf{D}_Y &= \begin{pmatrix} 1 \end{pmatrix} \end{aligned} \tag{4.5}$$

which is the desired result: columns of the processor grid form the cosets of null($\mathbf{R}_X$) and rows of the processor grid form the cosets of null($\mathbf{R}_Y$).

The overall Algorithm `SINGLE-LOOP-REPLICATION-ALIGNMENT` is summarized in Figure 4.1.

5. Heuristics

In practice, systems of alignment constraints are usually over-determined, so it is necessary to drop one or more constraints to obtain parallel execution. As we mentioned in the introduction, it is very difficult to determine which constraints must be dropped to obtain an optimal solution. In this section, we discuss our heuristic which is motivated by scalability analysis of common computational kernels.

5.1 Lessons from some common computational kernels

We motivate our ideas by the following example. Consider a loop nest that computes matrix-matrix product:

```
DO i=1,n
  DO j=1,n
    DO k=1,n
       C(i,j) = C(i,j) + A(i,k)*B(k,j)
```

[9] provides the description of various parallel algorithms for matrix-matrix multiplication. It is shown that the *best scalability* is achieved by an algorithm which organizes the processors into a 3-D grid. Let p, q and r be the processor indices in the grid. Initially, A is partitioned in 2-D blocks along the p-r "side" of the grid. That is, if we let A^{pr} be a block of A, then it is initially placed on processor with the coordinates $(p, 0, r)$. Similarly, each block B^{rq} is placed on processor $(0, q, r)$. Our goal is to accumulate the block C^{pq} of the result on the processor $(p, q, 0)$.

At the start of the computation, A is replicated along the second (q) dimension of the grid. B is replicated along the first dimension (p). Therefore, we end up with processor (p, q, r) holding a copy of A^{pr} and B^{rq}. Then each processor computes the local matrix-matrix product:

$$D^{pqr} = A^{pr} * B^{rq} \tag{5.1}$$

It is easy to see that the blocks of C are related to these local products by:

$$C^{pq} = \sum_r D^{pqr} \tag{5.2}$$

Therefore, after the local products are computed, they are reduced along the r dimension of the grid.

We can describe this computation using our algebraic framework. There is a 3-D template and the computation alignment is an identity. Each of the arrays is replicated. For example the values of $\mathbf{D}$ and $\mathbf{R}$ for the array A are:

$$\mathbf{R} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \; \mathbf{D} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \tag{5.3}$$

By collapsing different dimensions of the 3-D grid, we get 2-D and 1-D versions of this code. In general, it is difficult for a compiler to determine which version to use — the optimal solution depends on the size of the matrix, and on the overhead of communication relative to computation of the parallel machine [9]. On modern machines where the communication overhead is relatively small, the 3-D algorithm is preferable, but most alignment heuristics we have seen would not produce this solution — note that all arrays are communicated in this version! These heuristics are much more likely to "settle" for the 2-D or 1-D versions, with some of the arrays kept local.

Similar considerations apply to other codes such as matrix-vector product, 2-D and 3-D stencil computations, and matrix factorization codes [9]. Consider stencil computations. Here is a typical example:

```
DO i=1,N
  DO j=1,N
```

```
A(i,j) = ...B(i-1,j)...B(i+1,j)...
         ...B(i,j)...B(i,j-1)...B(i,j+1))
```

In general, stencil computations are characterized by array access functions of the form $\mathbf{Fi} + \mathbf{f}_k$, where the linear part $\mathbf{F}$ is the same for most of the accesses. The difference in the offset induces nearest-neighbor communication. We will analyze the communication/computation cost ratio for 1-D and 2-D partitioning of this example. For the 1-D case, the N-by-N iteration space is cut up into N/P-by-N blocks. If N is large enough, then each processor has to communicate with its "left" and "right" neighbors, and the volume of communication is $2N$. We can assume that the communication between the different pairs of neighbors happens at the same time. Therefore, the total communication time is $\Theta(2N)$. The computation done on each processor is $\Theta(N^2/P)$, so the ratio of communication to computation is $\Theta(P/N)$. In the 2-D case, the iteration space is cut up into $N/\sqrt{P}$-by-$N/\sqrt{P}$ blocks. Each processor now has four neighbors to communicate with, and the volume of communication is $4N/\sqrt{P}$. Therefore, the ratio for this case is $\Theta(\sqrt{P}/N)$. We conclude that 2-D case *scales* better than 1-D case[2]. In general, if we have a d-dimensional stencil-like computation, then it pays to have a d-dimensional template.

The situation is somewhat different in matrix and vector products and matrix factorization codes (although the final result is the same). Let us consider matrix-vector product together with some vector operation between X and Y:

```
DO i=1,N
  DO j=1,N
    Y(i) = Y(i) + A(i,j) * X(j)

DO i=1,N
  X(i) = ...Y(i)...
```

This fragment is typical of many iterative linear system solvers ([11]). One option is to use a 1-D template by leaving the constraint for X in the matrix-vector product loop unsatisfied. The required communication is an all-to-all broadcast of the elements of X. The communication cost is $\Theta(N \log(P))$. The computation cost is $\Theta(N^2/P)$. This gives us communication to computation ratio of $\Theta(\log(P)P/N)$.

In the 2-D version, each processor gets an $N/\sqrt{P}$-by-$N/\sqrt{P}$ block of the iteration space and A. X and Y are partitioned in $\sqrt{P}$ pieces placed along the diagonal of the processor grid ([9]). The algorithm is somewhat similar to matrix-matrix multiplication: each block of X gets broadcast along the

[2] In fact, the total volume of communication is smaller in the 2-D case, despite the fact that we had fewer alignment constraints satisfied (this paradoxical result arises from the fact that the amount of communication is a function not just of alignment but of distribution as well).

column dimension, each block of Y is computed as the sum-reduction along the row dimension. Note that because each broadcast or reduction happens in parallel, the communication cost is $\Theta(\log(\sqrt{P})N/\sqrt{P}) = \Theta(\log(P)N/\sqrt{P})$. This results on the communication to computation ratio of $\Theta(\log(P)\sqrt{P}/N)$. Although the total volume of communication is roughly the same for the 1-D and 2-D case, the cost is asymptotically smaller in the 2-D case. Intuitively, the reason is that we were able to parallelize communication itself.

To reason about this in our framework, let us focus on matrix-vector product, and see what kind of replication for X we get for the 1-D and 2-D case. In the 1-D case, the computation alignment is:

$$\mathbf{C} = \begin{pmatrix} 1 & 0 \end{pmatrix} \tag{5.4}$$

The replication equation $\mathbf{RC} = \mathbf{DF}$ for X is:

$$\mathbf{R}_X \begin{pmatrix} 1 & 0 \end{pmatrix} = \mathbf{D}_X \begin{pmatrix} 0 & 1 \end{pmatrix} \tag{5.5}$$

The only solution is:

$$\mathbf{R}_X = \mathbf{D}_X = \begin{pmatrix} 0 \end{pmatrix} \tag{5.6}$$

This means that every processor gets all elements of X — *i.e.*, it is an all-to-all broadcast. We have already computed the alignments for the 2-D case in Section 4.. Because $\mathbf{R}_X$ has rank 1, we have a "parallelizable" broadcasts — that is, the broadcast along different dimensions of the processor grid can happen simultaneously. In general, if the replication matrix has rank r and the template has dimension d, then we have broadcasts along $d - r$ dimensional subspaces of the template. The larger r, the more of these broadcasts happen at the same time. In the extreme case $r = d$ we have a replication-free alignment, which requires no communication, at all.

5.2 Implications for alignment heuristic

The above discussion suggests the following heuristic strategy.

- If a number of constraints differ only in the offset of the array access function, use only one of them.
- If there is a d-dimensional DOALL loop (or loop with reductions), use a d-dimensional template for it and try to satisfy conflicting constraints via replication. Keep the d-dimensional template if the rank of the resulting replication matrices is greater than zero.
- If the above strategy fails, use a greedy strategy based on array dimensions as a cost measure. That is, try to satisfy the alignment constraints for the largest array first (intuitively, we would like large arrays to be "locked in place" during the computation). This is the strategy used by Feautrier [5].

Intuitively, this heuristic is biased in favor of exploiting parallelism in DO-ALL loops, since communication can be performed in parallel before the computation starts. This is true even if there are reductions in the loop nest, because the communication required to perform reductions can also be parallelized. This bias in favour of exploiting parallelism in DO-ALL loops at the expense of communication is justified on modern machines.

6. Conclusion

We have presented a simple framework for the solution of the alignment problem. This framework is based on linear algebra, and it permits the development of simple and fast algorithms for a variety of problems that arise in alignment.

References

1. Jennifer M. Anderson and Monica S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 112 – 125, June 1993.
2. Siddartha Chatterjee, John Gilbert, and Robert Schreiber. The alignment-distribution graph. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing. Sixth International Workshop.*, number 768 in LNCS. Springer-Verlag, 1993.
3. Siddartha Chatterjee, John Gilbert, Robert Schreiber, and Shang-Hua Teng. Optimal evaluation of array expressions on massively parallel machines. Technical Report CSL-92-11, XEROX PARC, December 1992.
4. Henri Cohen. *A Course in Computational Algebraic Number Theory.* Graduate Texts in Mathematics. Springer-Verlag, 1995.
5. Paul Feautrier. Toward automatic distribution. Technical Report 92.95, IBP/MASI, December 1992.
6. C.-H. Huang and P. Sadayappan. Communication-free hyperplane partitioning of nested loops. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing. Fourth International Workshop. Santa Clara, CA.*, number 589 in LNCS, pages 186–200. Springer-Verlag, August 1991.
7. Kathleen Knobe, Joan D. Lucas, and William J. Dally. Dynamic alignment on distributed memory systems. In *Proceedings of the Third Workshop on Compilers for Parallel Computers*, July 1992.
8. Kathleen Knobe and Venkataraman Natarajan. Data optimization: minimizing residual interprocessor motion on SIMD machines. In *Proceedings of the 3rd Symposium on the Frontiers of Massively Parallel Computation - Frontiers 90*, pages 416–423, October 1990.
9. Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing. Design and Analysis of Algorithms.* The Benjamin/Cummings Publishing Company, 1994.
10. Jingke Li and Marina Chen. Index domain alignment: minimizing cost of cross-referencing between distributed arrays. Technical Report YALEU/DCS/TR-725, Department of Computer Science, Yale University, September 1989.

11. Youcef Saad. Kyrlov subspace methods on supercomputers. *SIAM Journal on Scientific and Statistical Computing*, 10(6):1200–1232, November 1989.
12. Michael Wolfe. *High Performance Compilers for Parallel Computing.* Addison–Wesley, Redwood City, CA, 1996.

Acknowledgement. This research was supported by an NSF Presidential Young Investigator award CCR-8958543, NSF grant CCR-9503199, ONR grant N00014-93-1-0103, and a grant from Hewlett-Packard Corporation.

A. Reducing the solution matrix

As we mentioned in Section 2.4, it is possible for our solution procedure to produce a matrix $\mathbf{U}$ which has more rows than the rank of any of the computation alignments $\mathbf{C}_i$. Intuitively, this means that we end up with a template that has a larger dimension than can be exploited in any loop nest in the program. Although the extra dimensions can be 'folded' away during the distribution phase, we show how the problem can be eliminated by adding an extra step to our alignment procedure. First, we discuss two ways in which this problem can arise.

A.1 Unrelated constraints

Suppose we have two loops with iteration alignments $\mathbf{C}_1$ and $\mathbf{C}_2$ and two arrays A and B with data alignments $\mathbf{D}_A$ and $\mathbf{D}_B$. Furthermore, only A is accessed in loop 1 via access function $\mathbf{F}_A$ and only B is accessed in loop 2 via access function $\mathbf{F}_B$ [3]. The alignment equations in this case are:

$$\mathbf{C}_1 = \mathbf{D}_A\mathbf{F}_A \tag{A.1}$$

$$\mathbf{C}_2 = \mathbf{D}_B\mathbf{F}_B \tag{A.2}$$

We can assemble this into a combined matrix equation:

$$\begin{aligned}
\mathbf{U} &= \begin{pmatrix} \mathbf{C}_1 & \mathbf{C}_2 & \mathbf{D}_A & \mathbf{D}_B \end{pmatrix} \\
\mathbf{V} &= \begin{pmatrix} \mathbf{I} & 0 \\ 0 & \mathbf{I} \\ -\mathbf{F}_A & 0 \\ 0 & -\mathbf{F}_B \end{pmatrix} \\
\mathbf{UV} &= \mathbf{0}
\end{aligned} \tag{A.3}$$

Say, $\mathbf{C}_1^*$ and $\mathbf{D}_A^*$ are the solution to (A.1). And $\mathbf{C}_2^*$ and $\mathbf{D}_B^*$ are the solution to (A.2). Then it is not hard to see that the following matrix is a solution to (A.3):

$$\mathbf{U} = \left(\begin{array}{c|c|c|c} \mathbf{C}_1^* & \mathbf{0} & \mathbf{D}_A^* & \mathbf{0} \\ \mathbf{0} & \mathbf{C}_2^* & \mathbf{0} & \mathbf{D}_B^* \end{array}\right) \tag{A.4}$$

So we have obtained a processor space with the dimension being the sum of the dimensions allowed by (A.1) (say, p_1) and (A.2) (say p_2). However, these dimensions are not fully utilized since only the first p_1 dimensions are used in loop 1, and only the remaining p_2 dimensions are used in loop 2.

This problem is relatively easy to solve. In general, we can model the alignment constraints as an undirected *alignment constraint graph* G whose

[3] For simplicity we are considering linear alignments and subscripts. For affine alignments and subscripts the argument is exactly the same after the appropriate encoding.

vertices are the unknown $\mathbf{D}$ and $\mathbf{C}$ alignment matrices; an edges (x, y) represents an alignment equation constraining vertex x and vertex y. alignments. We solve the constraints in each connected component separately, and choose a template with dimension equal to the maximum of the dimensions required for the connected components.

A.2 General Procedure

Unfortunately, extra dimensions can arise even when there is only one component in the alignment constraint graph. Consider the following program fragment:

```
DO i=1,n
  DO j=1,n
    ...A(i,0,j)...
```

The alignment equation for this loop is:

$$\mathbf{C} \quad = \quad \mathbf{D}_A \begin{pmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{pmatrix}$$

The solution is:

$$\mathbf{U} = \begin{pmatrix} \mathbf{C} & \mathbf{D} \end{pmatrix} = \left(\begin{array}{cc|ccc} 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \end{array}\right)$$

So we have $\text{rank}(\mathbf{C}) = 2 < \text{rank}(\mathbf{U})$. If we use this solution, we end up placing the unused dimension of A onto an extra dimension of virtual processor space.

We need a way of modifying the solution matrix $\mathbf{U}$ so that:

$$\text{rank}(\mathbf{U}) \quad = \quad \max_i\{\text{rank}(\mathbf{C}_i)\} \tag{A.5}$$

For this, we apply elementary (unimodular) row operations [4] to $\mathbf{U}$ so that we end up with a matrix $\mathbf{U}'$ in which the *first* $\text{rank}(\mathbf{C}_i)$ rows of each $\mathbf{C}_i$ component form a *row basis* for the rows of this component. We will say that each component of $\mathbf{U}'$ is *reduced*. By taking the first $\max_i\{\text{rank}(\mathbf{C}_i)\}$ rows of $\mathbf{U}'$ we obtain a desired solution $\mathbf{W}$.

In our example matrix $\mathbf{U}$ is not reduced: the first two rows of $\mathbf{C}$ do not for a basis for all rows of $\mathbf{C}$. But if we add the third row of $\mathbf{U}$ to the second row, we get $\mathbf{U}'$ with desired property:

$$\mathbf{U}' \quad = \quad \left(\begin{array}{cc|ccc} 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 \end{array}\right)$$

[4] Multiplying a row by ± 1 and adding a multiple of one row to another are elementary row operations.

Now by taking the first two rows of $\mathbf{U}'$ we obtain a solution which does not induce unused processor dimensions.

The question now becomes: how do we systematically choose a sequence of row operations on $\mathbf{U}$ in order to reduce its components? Without loss of generality, lets assume that $\mathbf{U}$ only consists of $\mathbf{C}_i$ components:

$$\mathbf{U} = \begin{pmatrix} \mathbf{C}_1 & \mathbf{C}_2 & \dots & \mathbf{C}_s \end{pmatrix} \tag{A.6}$$

Let:

- q be the number of rows in $\mathbf{U}$. Also, by construction of $\mathbf{U}$, $q = \text{rank}(\mathbf{U})$.
- r_i be the rank of C_i for $i = 1, \dots, s$.
- $r_{max} = \max_i\{\text{rank}(\mathbf{C}_i)\}$. Notice that $r_{max} \neq q$, in general.

We want to find matrix $\mathbf{W}$, so that:

- number of rows in $\mathbf{W}$ equals to r_{max}.
- each component of $\mathbf{W}$ has the same rank as the corresponding component of $\mathbf{U}$.

Here is the outline of our algorithm:

1. Perform elementary row operations on $\mathbf{U}$ to get $\mathbf{U}'$ in which every component is reduced.
2. Set $\mathbf{W}$ to the first r_{max} rows of $\mathbf{U}'$.

The details are filled in below.

We need the following Lemma:

Lemma A.1. *Let $\mathbf{a}_1, \dots, \mathbf{a}_r, \mathbf{a}_{r+1}, \dots, \mathbf{a}_n$ be some vectors. Furthermore assume that the first r vectors form a basis for the span $\mathbf{a}_1, \dots, \mathbf{a}_n$. Let:*

$$\mathbf{a}_k = \sum_{j=1}^{r} \beta_j \mathbf{a}_j \tag{A.7}$$

be the representation of $\mathbf{a}_k$ in the basis above. Then the vectors $\mathbf{a}_1, \dots, \mathbf{a}_{r-1}$, $\mathbf{a}_r + \alpha\mathbf{a}_k$ are linearly independent (and form a basis) if and only if:

$$1 + \alpha\beta_r \neq 0 \tag{A.8}$$

Proof:.

$$\begin{aligned} \mathbf{a}_r + \alpha\mathbf{a}_k &= \mathbf{a}_r + \alpha \sum_{j=1}^{r} \beta_j \mathbf{a}_j \\ &= \mathbf{a}_r(1 + \alpha\beta_r) + \alpha \sum_{j=1}^{r-1} \beta_j \mathbf{a}_j \end{aligned} \tag{A.9}$$

Now if in the equation (A.9) $(1+\alpha\beta_r) = 0$, then the vectors $\mathbf{a}_1, \dots, \mathbf{a}_{n-1}, \mathbf{a}_r + \alpha\mathbf{a}_k$ are linearly dependent. Vice versa, if $(1 + \alpha\beta_r) \neq 0$, then these vectors are independent by the assumption on the original first r vectors. □

Lemma A.1 forms a basis for an inductive algorithm to reduce all components of $\mathbf{U}$. Inductively assume that we have already reduced $\mathbf{C}_1, \ldots, \mathbf{C}_{k-1}$. Below we show how to reduce $\mathbf{C}_k$, while keeping the first $k-1$ components reduced.

Let

$$\mathbf{C}_j = \begin{pmatrix} \mathbf{a}_1^T \\ \mathbf{a}_2^T \\ \vdots \\ \mathbf{a}_q^T \end{pmatrix}$$

we want the first r_j rows to be linearly independent. Assume inductively that the first $i-1$ rows $(i < r_j)$ are already linearly independent. There are two cases for the i-th row $(\mathbf{a}_i^T)$:

1. $\mathbf{a}_i^T$ is linearly independent from the previous rows. In this case we just move to the next row.
2. $\mathbf{a}_i = \sum_{\ell=1}^{i-1} \gamma_\ell \mathbf{a}_\ell$, i.e. $\mathbf{a}_i$ is linearly dependent on the previous rows. Note that since $r_j = rank(\mathbf{C}_j) > i$, there is a row $\mathbf{a}_p$, which is linearly independent from the first $i-1$ rows. Because of this the rows $\mathbf{a}_1, \ldots, \mathbf{a}_{i-1}, \mathbf{a}_i + \alpha \mathbf{a}_p$ are linearly independent for any $\alpha \neq 0$.
   Lemma A.1 tells us that we can choose α so that the previous components are kept reduced. We have to solve a system of inequalities like:
   $$\begin{cases} 1 + \alpha\beta_{r_1}^{(1)} & \neq 0 \\ 1 + \alpha\beta_{r_2}^{(2)} & \neq 0 \\ \vdots \\ 1 + \alpha\beta_{r_{k-1}}^{(k-1)} & \neq 0 \end{cases} \tag{A.10}$$
   where $\beta_{r_1}^{(1)}, \ldots, \beta_{r_{k-1}}^{(k-1)}$ come from the inequalities (A.8) for each component. $\beta_{r_i}^{(i)}$'s are rational numbers: $\beta_{r_i}^{(i)} = \eta_i/\xi_i$. So we have to solve a system of inequalities:
   $$\begin{cases} \alpha\eta_1 & \neq -\xi_1 \\ \alpha\eta_2 & \neq -\xi_2 \\ \vdots \\ \alpha\eta_{k-1} & \neq -\xi_{k-1} \end{cases} \tag{A.11}$$
   It is easy to see that $\alpha = \max_i\{|\xi_i|\} + 1$ is a solution.

The full algorithm for communication-free alignment `ALIGNMENT-WITH-FIX-UP` is outlined in Figure A.1.

B. A Comment on Affine Encoding

Finally, we make a remark about affine encoding. A sanity check for alignment equations is that there should always be a trivial solution which places

Input:. A set of encoded affine alignment constraints as in Equation (3.4).
Output:. Communication-free alignment mappings characterized by $\mathbf{C}_j$, $\mathbf{c}_j$, $\mathbf{D}_k$, $\mathbf{d}_k$ which do not induce unused processor dimensions.
1. Form alignment constraint graph G.
2. For each connected component of G:
   a) Assemble the system of constraints and solve it as described in Algorithm `AFFINE-ALIGNMENT` to get the solution matrix $\hat{\mathbf{U}}$.
   b) Remove the extra row of of $\hat{\mathbf{U}}$ that was induced by affine encoding. (Section B.)
   c) If necessary apply the procedure described in Section A.2 to reduce the computation alignment components of $\hat{\mathbf{U}}$.

Fig. A.1. Algorithm `ALIGNMENT-WITH-FIXUP`

everything onto one processor. In the case of linear alignment functions and linear array accesses, we have a solution $\mathbf{U} = \mathbf{0}$. When we use affine functions, this solution is still valid, but there is more. We should be able to express a solution $\hat{\mathbf{U}} \neq \mathbf{0}$ that places everything on a single non-zero processor. Such a solution would have $\mathbf{C}_i = \mathbf{0}$, $\mathbf{D}_j = \mathbf{0}$, $\mathbf{c}_i = \mathbf{d}_j = 1$. Or, using our affine encoding:

$$\begin{aligned}\hat{\mathbf{C}}_i &= \left(\begin{array}{cccc|c} 0 & 0 & \dots & 0 & 1\end{array}\right)\\ \hat{\mathbf{D}}_j &= \left(\begin{array}{ccccc|c} 0 & 0 & \dots & 0 & 0 & 1\end{array}\right)\end{aligned}$$

Below, we prove that solution of this form always exists; moreover, this gives rise to an extra processor dimension which can be eliminated without using the algorithm of Section A..

Let the matrix of unknowns be:

$$\hat{\mathbf{U}} = \left(\begin{array}{cccccc} \hat{\mathbf{C}}_1 & \dots & \hat{\mathbf{C}}_s & \hat{\mathbf{D}}_1 & \dots & \hat{\mathbf{D}}_t \end{array}\right)$$

Also let:

- m_i be the number of columns of C_i for $i = 1, \dots, s$. (m_i is the dimension of the ith loop.)
- m_{s+i} be the number of columns of D_i for $i = 1, \dots, t$. (m_{s+i} is the dimension of the $(s+i)$th array.)
- $\mathbf{e}_k \in \mathbb{Z}^k$, $\mathbf{e}_k = \left(\begin{array}{ccccc} 0 & 0 & \dots & 0 & 1\end{array}\right)^T$. ($k-1$ zeros followed by a 1.)
- $\mathbf{w} \in \mathbb{Z}^{(m_1+m_2+\dots+m_{s+t})}$ as in:

$$\mathbf{w} = \left(\begin{array}{c} \mathbf{e}_{m_1} \\ \mathbf{e}_{m_2} \\ \vdots \\ \mathbf{e}_{m_s} \\ \mathbf{e}_{m_{s+1}} \\ \vdots \\ \mathbf{e}_{m_{s+t}} \end{array}\right)$$

It is not hard to show that $\mathbf{w}^T\hat{\mathbf{V}} = \mathbf{0}$. In particular, we can show that vector $\mathbf{w}$ is orthogonal to every block column $\hat{\mathbf{V}}_q$ that is assembled into $\hat{\mathbf{V}}$. Suppose that $\hat{\mathbf{V}}_q$ corresponds to the equation:

$$\hat{\mathbf{C}}_i = \hat{\mathbf{D}}_j\hat{\mathbf{F}}_k$$

Therefore:

$$\hat{\mathbf{V}}_q = \begin{pmatrix} \mathbf{0} \\ \mathbf{I} \\ \mathbf{0} \\ -\hat{\mathbf{F}}_k \\ \mathbf{0} \end{pmatrix}$$

Note that $\hat{\mathbf{V}}_q$ has m_i columns (the dimension of the ith loop) and the last column looks like (check the definition of $\hat{\mathbf{F}}$ in Section 3.1:

$$\begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ 0 \\ \vdots \\ 0 \\ -1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

with 1 and -1 placed in the same positions as the 1s in $\mathbf{w}$. It is clear that $\mathbf{w}$ is orthogonal to this column of $\hat{\mathbf{V}}_q$. $\mathbf{w}$ is also orthogonal to the other columns of $\hat{\mathbf{V}}_q$, since only the last column has non-zeros, where $\mathbf{w}$ has 1s.

How can we remove an extra dimension in $\hat{\mathbf{U}}$ that corresponds to $\mathbf{w}$? Note that in general $\hat{\mathbf{U}}$ will not have a row that is a multiple of $\mathbf{w}$! Suppose that $\hat{\mathbf{U}}$ has $r = \text{rank}(\hat{\mathbf{U}})$ rows:

$$\hat{\mathbf{U}} = \begin{pmatrix} \mathbf{u}_1^T \\ \mathbf{u}_2^T \\ \vdots \\ \mathbf{u}_r^T \end{pmatrix}$$

Since $\mathbf{w}^T\hat{\mathbf{V}} = \mathbf{0}$, we have that

$$\mathbf{w} \in \text{null}(\hat{\mathbf{V}}^T)$$

But rows of $\hat{\mathbf{U}}$ form a basis for null$(\hat{\mathbf{V}}^T)$. Therefore:

$$\mathbf{w} \quad \in \quad \mathrm{span}(\mathbf{u}_1, \ldots, \mathbf{u}_r) \tag{B.1}$$

Let $\mathbf{x}$ be the solution to:

$$\mathbf{x}^T \hat{\mathbf{U}} \quad = \quad \mathbf{w}^T$$

One of the coordinates of $\mathbf{x}$, say x_ℓ, must be non-zero. Form the matrix $\mathbf{J}(\mathbf{x})$ by substituting the ℓth row of an r-by-r identity matrix with $\mathbf{x}^T$:

$$\mathbf{J}(\mathbf{x}) \quad = \quad \begin{pmatrix} 1 & 0 & 0 & 0 & \ldots & 0 & \ldots & 0 \\ 0 & 1 & 0 & 0 & \ldots & 0 & \ldots & 0 \\ 0 & 0 & 1 & 0 & \ldots & 0 & \ldots & 0 \\ 0 & 0 & 0 & 1 & \ldots & 0 & \ldots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_1 & x_2 & x_3 & x_4 & \ldots & x_\ell & \ldots & x_r \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \ldots & 0 & \ldots & 1 \end{pmatrix}$$

$\mathbf{J}(\mathbf{x})$ is non-singular, because $x_\ell \neq 0$. Therefore $\hat{\mathbf{U}}' = \mathbf{J}(\mathbf{x})\hat{\mathbf{U}}$ has the same rank as $\hat{\mathbf{U}}$ and it is also a basis for the solutions to our alignment system:

$$\hat{\mathbf{U}}'\hat{\mathbf{V}} = \mathbf{J}(\mathbf{x})\hat{\mathbf{U}}\hat{\mathbf{V}} = \mathbf{J}(\mathbf{x})\mathbf{0} = \mathbf{0}$$

But by construction:

$$\hat{\mathbf{U}}' \quad = \quad \begin{pmatrix} \mathbf{u}_1^T \\ \mathbf{u}_2^T \\ \vdots \\ \mathbf{u}_{\ell-1}^T \\ \mathbf{w}^T \\ \mathbf{u}_{\ell+1}^T \\ \vdots \mathbf{u}_r^T \end{pmatrix}$$

Now we can just remove the $\mathbf{w}^T$ row to get non-trivial solutions! Notice that we don't really have to form $\mathbf{J}(\mathbf{x})$ — we have to find $\mathbf{x}$ (using Gaussian elimination) and then remove the ℓth row from $\hat{\mathbf{U}}$ such that $x_\ell \neq 0$.