Solving Alignment using Elementary Linear Algebra

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

Cornell University

Abstract. 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 read-only data to eliminate communication. Our matrix-based approach leads to algorithms which are simpler and faster than existing algorithms for the alignment problem.

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.

* This research was supported by an NSF Presidential Young Investigator award CCR-8958543, NSF grant CCR-9008526, ONR grant N00014-93-1-0103, and a grant from Hewlett-Packard Corporation.

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 [HS91] — 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. M is a symbolic constant; the code computes the M-th element of the convolution of two arrays X and Y.

```
      DO 1 i = 1, M
1         Z = Z + X(i) * Y(M - i + 1)
```

Since the loop nest has just one loop, and all arrays are one-dimensional, the virtual processors are assumed to be organized as a one-dimensional grid 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 must have Z, $X(i)$ and $Y(M-i+1)$ in its local memory. These constraints can be expressed formally by defining the following functions.

$\mathbf{C}(\mathrm{i})$: i $\rightarrow$ T – map from loop iterations to virtual processors
$\mathbf{D}_z(\mathrm{i})$: i $\rightarrow$ T – processor that owns Z
$\mathbf{D}_x(\mathrm{i})$: i $\rightarrow$ T – map of array X
$\mathbf{D}_y(\mathrm{i})$: i $\rightarrow$ T – map of array Y

The constraints on these functions are the following.

$$for\ i \in 1..M \begin{cases} \mathbf{C}(i) = \mathbf{D}_z \\ \mathbf{C}(i) = \mathbf{D}_x(i) \\ \mathbf{C}(i) = \mathbf{D}_y(M-i+1) \end{cases}$$

If we enforce these constraints, we end up with the trivial solution in which all data and computations are mapped to a single processor, since Z can be mapped to just one processor. Alternatively, we can leave the constraint on Z unsatisfied. In that case, we can map the computation so that iteration i is mapped to processor i, and we can map arrays X and Y so that elements $X(i)$ and $Y(M-i+1)$ are mapped to processor i. This gives a non-trivial solution to the system of constraints; using this, we can compute all the partial products in parallel without moving X or Y array elements[2]. For future reference, we note that the map of loop iterations to virtual processors is a linear function of the iteration number; therefore, this function can be represented by the unit matrix **[1]**, with function application represented by matrix vector product. For example, iteration $\mathbf{i}$ is mapped to processor $[\mathbf{1}]\ \mathbf{i}$, which is simply $\mathbf{i}$, as desired. The map of array Y to the virtual processors is not linear but affine; the linear part is represented by the matrix **[-1]**, while the constant part of the map is **(M+1)**.

In this example, the solution to the alignment equations can be 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 **C** in the example). To make the problem tractable, it is standard to restrict the maps to linear 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. The null space problem can be solved through the use of integer-preserving Gaussian elimination of integer matrices, a standard tool in modern parallelizing compilers where it has found use in dependence analysis, loop transformations and code generation [Ban93, LP92]. One weakness of existing approaches to alignment is that they handle only linear functions; general affine functions, like the map of array Y, 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 read-only data is useful for exploiting parallelism. Suppose we change the convolution program so that we convolve the array X with itself (in the program shown above, Y gets replaced with X). In that case, the processor that performs iteration i must own both $X(i)$ and $X(M-i+1)$. The only linear (or affine) map that permits this is the trivial map that maps everything to a single processor. One way to use M processors is to replicate array X, and map one copy so that processor i owns $X(i)$, and map the other copy so that processor i owns $X(M-i+1)$. This permits us to run the loop in parallel without moving elements of X during the execution of the loop. Although we could replicate *all* read-only data on *all* processors, it is desirable to conserve space and replicate data only if it helps achieve parallel execution. In Section 4, we show that our framework permits a solution to the replication problem as well.

How does our work relate to previous work on alignment? Our work is closest

[2] We are not interested here in the question of whether parallel execution is actually worthwhile for this program — perhaps the sum can be done using a collective communication routine to get some benefit from parallel execution.

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 [HS91]. However, they did not give a general method for solving these equations. Also, they did not handle replication of data. Recently, Anderson and Lam gave a solution method but their algorithm is quite complex, requiring the determination of cycles in bipartite graphs, computing pseudo-inverses etc. [AL93] — these complications are eliminated by our approach. Complementary to our work is the solution of the (NP-complete) problem of determining the best choice of constraints to leave unsatisfied in overconstrained systems of equations. Heuristics for this problem were first discussed by Li and Chen[LC89] for a limited kind of alignment called axis alignment. More general heuristics for a wide variety of cost-of-communication metrics have been studied by Chatterjee, Gilbert and Schreiber[CGS93, CGST92], Feautrier [Fea92] and Knobe et al [KN90, KLD92]. Incorporating such heuristics into our framework should give a clean and complete approach to alignment.

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 (Section 2.2).
2. We show that this linear algebra problem can be solved through the use of integer preserving Gaussian elimination (Section 2.3).
3. Existing 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).
4. Finally, we show how replication of read-only data is handled by our framework (Section 4).

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 problem of determining a basis for the null space of a matrix. Finally, we show how this standard linear algebra problem 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 2 j = 1,100
         DO 2 k = 1,100
2           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:

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

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

$$\begin{cases} \mathbf{C} = \mathbf{D}_B\mathbf{F}_1 \\ \mathbf{C} = \mathbf{D}_A\mathbf{F}_1 \\ \mathbf{C} = \mathbf{D}_A\mathbf{F}_2 \end{cases} \tag{1}$$

We will refer to the equation scheme $\mathbf{C} = \mathbf{DF}$ as the fundamental equation of alignment. One way to solve systems of such equations is to set $\mathbf{C}$ and $\mathbf{D}$ 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.

The general principle behind 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 solved to find communication-free maps of computations and data.

2.2 Reduction to Null Space Computation

We solve alignment equations by reduction to the standard linear algebra problem of determining a basis for the null space of a matrix. Consider a single equation.

$$\mathbf{C} = \mathbf{D}\mathbf{F}$$

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

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

This equation is of the form $\mathbf{U}\mathbf{V} = 0$ 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 $\mathbf{V}^T$. Alternatively, a standard result in linear algebra states that such a basis can be obtained from a basis for the orthogonal complement of the range space of the matrix $\mathbf{V}$, written as range$(\mathbf{V})^\perp$ [GVL89].

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:

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

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

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

where the zeros are placed so that:

$$\mathbf{U}\mathbf{V}_q = \mathbf{C}_j - \mathbf{D}_k\mathbf{F}_\ell \tag{2}$$

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

$$\mathbf{U}\mathbf{V} = 0 \tag{3}$$

The reader can verify that the system of equations (1) can be converted into the following matrix equation:

$$\begin{bmatrix} \mathbf{C} & \mathbf{D}_A & \mathbf{D}_B \end{bmatrix} \begin{bmatrix} \mathbf{I} & \mathbf{I} & \mathbf{I} \\ \mathbf{0} & -\mathbf{F}_1 & -\mathbf{F}_2 \\ -\mathbf{F}_1 & \mathbf{0} & \mathbf{0} \end{bmatrix} = \mathbf{0} \tag{4}$$

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

Fig. 1. Algorithm `LINEAR-ALIGNMENT`.

2.3 Solving the matrix equation

To solve the equation $\mathbf{UV} = \mathbf{0}$, we have to get $\mathbf{V}$ into a 'rank-revealing' form. The idea is to use row and column operations as needed to get the matrix into a form in which its rank can be determined by inspection. There are many ways to accomplish this, and we describe one such way. Suppose $\mathbf{V} \in \mathbb{Z}^{M \times N}$ and $rank(\mathbf{V}) = r$. Then by doing integer preserving Gaussian elimination with full pivoting [Edm67] we can establish the following factorization:

$$\mathbf{HVP} = \begin{bmatrix} \mathbf{R}_{11} & \mathbf{R}_{12} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \tag{5}$$

where $\mathbf{H}$ is an $M \times M$ invertible matrix representing the row operations, $\mathbf{P}$ is an $N \times N$ unimodular matrix representing the column operations (which are permutations), $\mathbf{R}_{11}$ is an $r \times r$ upper triangular invertible matrix. It is a property of this factorization that the last $N - r$ rows of $\mathbf{H}$ span the null-space of $\mathbf{V}$ and therefore give us the solution:

$$\mathbf{U} = \mathbf{H}(M - (N - r) + 1 : M, 1 : M) \tag{6}$$

This also means that during the elimination we need to store only $\mathbf{H}$, the composition of row operations. The general algorithm is outlined in Figure 1.

For Equation (4), a solution matrix is:

$$\mathbf{U} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

which gives us:

$$\mathbf{C} = \mathbf{D}_A = \mathbf{D}_B = \begin{bmatrix} 1 & 1 \end{bmatrix}$$

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.

2.4 Remarks

Suppose that the matrix of constraints $\mathbf{V}$ has N rows and M columns and has rank r. Then Gaussian elimination with full pivoting requires $O(rMN + r^3)$ *arithmetic* operations. Note that r is at most $\min(M, N)$. A subtle point is that this complexity measure ignores the sizes of matrix entries during the elimination, but it is a well known result that the sizes of numbers involved grow polynomially at the worst[Edm67]. By exploiting the fact that the matrices arising in alignment are sparse and by reordering them so that symbolic constants are used late in the elimination, it may be possible to speed up this algorithm if necessary.

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:

$$\mathbf{Ci} + \mathbf{b} = \mathbf{C}(\mathbf{i} + \mathbf{z}) + \mathbf{b} \Leftrightarrow \mathbf{Cz} = \mathbf{0} \tag{7}$$

We can now easily incorporate (7) into our matrix system (3) by adding the following block column to $\mathbf{V}$:

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

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

$$\mathbf{V}_p = \begin{bmatrix} \mathbf{0} \\ \mathbf{F_1} \\ \mathbf{0} \\ -\mathbf{F_2} \\ \mathbf{0} \end{bmatrix} \tag{8}$$

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

Finally, one practical note. It is possible for Algorithm `LINEAR-ALIGNMENT` to produce a solution $\mathbf{U}$ which has p rows, even though all $\mathbf{C}_j$ and $\mathbf{D}_k$ produced by Step 5 have rank less than p. In other words, the solution may use only a low dimensional subspace of the entire template. 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. For lack of space, we do not describe this modification; the interested reader is referred to an associated technical report which can be obtained from the authors.

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 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.

$$\text{Computation maps: } C_j(\mathbf{i}) = \mathbf{C}_j\mathbf{i} + \mathbf{c}_j \tag{9}$$

$$\text{Data maps: } D_k(\mathbf{a}) = \mathbf{D}_k\mathbf{a} + \mathbf{d}_k \tag{10}$$

$$\text{Data access functions: } F_\ell(\mathbf{i}) = \mathbf{F}_\ell\mathbf{i} + \mathbf{f}_\ell \tag{11}$$

$\mathbf{C}_j$, $\mathbf{D}_k$ 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

$$\forall \mathbf{i} \in \mathbb{Z}^n : \mathbf{C}_j\mathbf{i} + \mathbf{c}_j = \mathbf{D}_k(\mathbf{F}_\ell\mathbf{i} + \mathbf{f}_\ell) + \mathbf{d}_k \tag{12}$$

3.1 Encoding affine constraints as linear constraints

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

$$\mathbf{T}\mathbf{x} + \mathbf{t} = \begin{bmatrix} \mathbf{T} & \mathbf{t} \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix} \tag{13}$$

where $\mathbf{T}$ is a matrix, and $\mathbf{t}$ and $\mathbf{x}$ are vectors. We can put (12) in the form:

$$\begin{aligned} \begin{bmatrix} \mathbf{C}_j & \mathbf{c}_j \end{bmatrix} \begin{bmatrix} \mathbf{i} \\ 1 \end{bmatrix} &= \mathbf{D}_k \begin{bmatrix} \mathbf{F}_\ell & \mathbf{f}_\ell \end{bmatrix} \begin{bmatrix} \mathbf{i} \\ 1 \end{bmatrix} + \mathbf{d}_k \\ &= \begin{bmatrix} \mathbf{D}_k & \mathbf{d}_k \end{bmatrix} \begin{bmatrix} \mathbf{F}_\ell & \mathbf{f}_\ell \\ \mathbf{0} & 1 \end{bmatrix} \begin{bmatrix} \mathbf{i} \\ 1 \end{bmatrix} \end{aligned} \tag{14}$$

Now we let:

$$\hat{\mathbf{C}}_j = \begin{bmatrix} \mathbf{C}_j & \mathbf{c}_j \end{bmatrix} \quad \hat{\mathbf{D}}_k = \begin{bmatrix} \mathbf{D}_k & \mathbf{d}_k \end{bmatrix} \quad \hat{\mathbf{F}}_\ell = \begin{bmatrix} \mathbf{F}_\ell & \mathbf{f}_\ell \\ \mathbf{0} & 1 \end{bmatrix} \tag{15}$$

(14) can be written as:

$$\forall \mathbf{i} \in \mathbb{Z}^d : \hat{\mathbf{C}}_j \begin{bmatrix} \mathbf{i} \\ 1 \end{bmatrix} = \hat{\mathbf{D}}_k\hat{\mathbf{F}}_\ell \begin{bmatrix} \mathbf{i} \\ 1 \end{bmatrix} \tag{16}$$

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

Lemma 1. *Let* $\mathbf{T}$ *be a matrix,* $\mathbf{t}$ *a vector. Then*

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

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:

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

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

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

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

Using Lemma 1, we can rewrite (16) as follows:

$$\hat{\mathbf{C}}_j = \hat{\mathbf{D}}_k \hat{\mathbf{F}}_\ell \tag{17}$$

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

$$\hat{\mathbf{U}}\hat{\mathbf{V}} = \mathbf{0} \tag{18}$$

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

```
    DO 3  i = 1, M
3       Z = Z + X(i) * Y(M - i + 1)
```

The array access functions are:

$$\begin{array}{ll} \mathbf{F}_X = [1] & \mathbf{f}_X = [0] \\ \mathbf{F}_Y = [-1] & \mathbf{f}_Y = [M+1] \\ \hat{\mathbf{F}}_X = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} & \hat{\mathbf{F}}_Y = \begin{bmatrix} -1 & M+1 \\ 0 & 1 \end{bmatrix} \end{array} \tag{19}$$

Assume that we want to compute the $X(i)Y(M-i+1)$ products in parallel and them sum them up using a collective communication procedure. The reader can verify that the matrix equation to be solved is the following one.

$$\hat{\mathbf{U}}\hat{\mathbf{V}} = \mathbf{0} \tag{20}$$

where:

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

Using the techniques in Section 2.3, it can be verified that a solution to the equation is the following matrix.

$$\hat{\mathbf{U}} = \begin{bmatrix} -1 \; 2+M & -1 \; 2+M & 1 \; 1 \\ -1 \; 1+M & -1 \; 1+M & 1 \; 0 \end{bmatrix} \tag{21}$$

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

$$\hat{\mathbf{C}} = \hat{\mathbf{D}}_X = \begin{bmatrix} -1 \; 2+M \\ -1 \; 1+M \end{bmatrix} \qquad \hat{\mathbf{D}}_Y = \begin{bmatrix} 1 \; 1 \\ 1 \; 0 \end{bmatrix}$$

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

$$\begin{bmatrix} -1 \; 2+M \\ -1 \; 1+M \end{bmatrix} \begin{bmatrix} i \\ 1 \end{bmatrix} = \begin{bmatrix} M+2-i \\ M+1-i \end{bmatrix}$$

As a check, note that $Y(M-i+1)$ is mapped to the same processor.

$$\begin{bmatrix} 1 \; 1 \\ 1 \; 0 \end{bmatrix} \begin{bmatrix} M+i-1 \\ 1 \end{bmatrix} = \begin{bmatrix} M+2-i \\ M+1-i \end{bmatrix}$$

Therefore, no communication is required to compute the $X(i)Y(M-i+1)$ products.

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. One way to remove the extra dimension is to use the general procedure mentioned in Section 2.4. A faster approach is to use the fact that vector

$$\mathbf{w} = \begin{bmatrix} 0 \; 0 \ldots 0 \; 1 \; 0 \ldots 0 \; 1 \ldots 0 \; 0 \; 1 \end{bmatrix}^T$$

(with zeros placed appropriately) is always orthogonal to $\hat{\mathbf{V}}^T$. For lack of space, we defer discussion of this to the associated technical report. Algorithm `AFFINE-ALIGNMENT` is summarized in Figure 2.

4 Replication

As we discussed in Section 1, communication-free alignment may require replication of read-only data. In this section, we show how replication of data is handled in our linear algebra framework. We have not worked out all the details of replication, so we will restrict attention to a single loop nest here. We use the rank-1 update loop as a running example. In this loop, elements of matrix A are updated by the corresponding element of the outer product of the vectors x and y, as shown below.

Input: A set of alignment constraints as in Equation (12).
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 14.
2. Assemble block columns $\mathbf{V}_q$ as in Equation (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 $\hat{\mathbf{U}}$ if desired.
6. Extract the solution matrices from $\hat{\mathbf{U}}$.

Fig. 2. Algorithm `AFFINE-ALIGNMENT`.

```
      DO 4 i = 1,100
         DO 4 j = 1,100
4           A(i,j) = A(i,j) + x(i) * y(j)
```

Assuming that we have a two dimensional processor template, let element (i, j) of array A be mapped to logical processor (i, j). The value of the element $x(i)$ is needed to compute all elements of A in row i. In other words, x needs to be replicated column-wise. Similarly, all elements of A in the column j need the value of $y(j)$. In other words, y needs to be replicated row-wise. We would like to derive this information automatically.

4.1 Formulation of replication

To handle replication, we associate a pair of matrices $(\mathbf{R}, \mathbf{D})$ with each data reference for which alignment is desired, and the fundamental equational scheme for alignment then 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,

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

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 not read-only or it is very large, we can disallow replication of that array. In the rank-1 update example, we choose to replicate vectors x and y, but not matrix A since A is not read-only. For this problem, the system of equations is the following:

$$\begin{aligned} \mathbf{C} &= \mathbf{D}_A \\ \mathbf{R}_x\mathbf{C} &= \mathbf{D}_x\ (1\ 0) \\ \mathbf{R}_y\mathbf{C} &= \mathbf{D}_y\ (0\ 1) \end{aligned}$$

We solve such systems of equations in two steps as follows.

1. Find communication-free alignment of non-replicated data and iterations, as was described in Sections 2 and 3. In this phase, we only take into account constraints involving non-replicated arrays.
2. Find the alignment (possibly with replication) of the rest of the arrays. We use the computation alignment ($\mathbf{C}$) found in the first step to solve the replication equations $\mathbf{RC} = \mathbf{DF}$, as described below.

Let $\mathbf{C}_{basis}$ be the computation alignment found in the first step. It is easy to see that multiplying $\mathbf{C}_{basis}$ and each $\mathbf{D}$ found in this step by a matrix $\mathbf{T}$ gives another solution to the replication-free equations. In the second step, we implicitly determine such a matrix $\mathbf{T}$ so that the remaining equations are satisfied. The replication equations for the second step can be written as follows:

$$\mathbf{RTC}_{basis} = \mathbf{DF} \tag{22}$$

In the case of a *single loop nest*, all $\mathbf{R}$ matrices in equations like (22) are multiplied on the left by the same $\mathbf{T}$ matrix. This means we can omit $\mathbf{T}$ from the equations[3], and solve the system of simplified equations:

$$\begin{aligned} \mathbf{RC}_{basis} &= \mathbf{DF} \\ &\Leftrightarrow \\ \begin{bmatrix} \mathbf{R}\ \mathbf{D} \end{bmatrix} \begin{bmatrix} \mathbf{C}_{basis} \\ -\mathbf{F} \end{bmatrix} &= \mathbf{0} \end{aligned} \tag{23}$$

We now use the procedure described in Section 2 to solve a system of equations of the form of (23). Algorithm `SINGLE-LOOP-REPLICATION-ALIGNMENT` is summarized in Figure 3.

We are currently working on extending this idea to handle multiple loop nests.

[3] This has the effect of transforming the virtual processor space by matrix $\mathbf{T}$

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. 3. Algorithm `SINGLE-LOOP-REPLICATION-ALIGNMENT`.

4.2 Example

We illustrate the above discussion using the rank-1 update example. The only replication-free equation is the one due to array A. Solving this equation gives us $\mathbf{D}_A = \mathbf{C}_{basis} = \mathbf{I}$. Now the equation for the second step of Algorithm `SINGLE-LOOP-REPLICATION-ALIGNMENT` is the following:

$$\begin{bmatrix} \mathbf{R}_x \ \mathbf{R}_y \ \mathbf{D}_x \ \mathbf{D}_y \end{bmatrix} \begin{bmatrix} \mathbf{C}_{basis} & \mathbf{0} \\ \mathbf{0} & \mathbf{C}_{basis} \\ -\mathbf{F}_x & \mathbf{0} \\ \mathbf{0} & -\mathbf{F}_y \end{bmatrix} =$$

$$\begin{bmatrix} \mathbf{R}_x \ \mathbf{R}_y \ \mathbf{D}_x \ \mathbf{D}_y \end{bmatrix} \begin{bmatrix} 1 & 0\,0\,0 \\ 0 & 1\,0\,0 \\ 0 & 0\,1\,0 \\ 0 & 0\,0\,1 \\ -1 & 0\,0\,0 \\ 0 & 0\,0\,-1 \end{bmatrix} = \mathbf{0}$$

This gives us a solution with $\mathbf{R}_x = \begin{bmatrix} 0\,1 \end{bmatrix}$, $\mathbf{R}_y = \begin{bmatrix} 1\,0 \end{bmatrix}$ and $\mathbf{D}_x = \mathbf{D}_y = 1$. Therefore, X is replicated in the j dimension, and Y is replicated in the i dimension, as desired.

5 Conclusion

We have presented a simple framework for the solution of the communication-free 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. A final step would be to integrate a good heuristic to determine which constraints should be left unsatisfied when the system of alignment constraints is overconstrained.

References

[AL93] 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.

[Ban93] U. Banerjee. *Loop transformations for restructuring compilers.* Kluwer Publishing, 1993.

[CGS93] 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.

[CGST92] 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.

[Edm67] Jack Edmonds. Systems of distinct representatives and linear algebra. *Journal of research of national bureau of standards (Sect. B)*, 71(4):241–245, 1967.

[Fea92] Paul Feautrier. Toward automatic distribution. Technical Report 92.95, IBP/MASI, December 1992.

[GVL89] Gene H. Golub and Charles F. Van Loan. *Matrix Computations.* The John Hopkins University Press, second edition, 1989.

[HS91] 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.

[KLD92] 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.

[KN90] 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.

[LC89] 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.

[LP92] W. Li and K. Pingali. Access normalization: loop restructuring for NUMA compilers. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992.