Initial Framework for Automatic Data Layout in Fortran D: A Short Update on a Case Study

Ken Kennedy *Ulrich Kremer*

CRPC-TR93324-S
July, 1993

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

Initial Framework for Automatic Data Layout in Fortran D : A Short Update on a Case Study*

Ken Kennedy
Ulrich Kremer†

Department of Computer Science
Rice University
P.O. Box 1892
Houston, Texas 77251

The automatic data layout algorithm takes Fortran 77 programs written in a data-parallel programming style as input and generates Fortran D programs with data layout specifications [FHK+90]. The generated specifications determine the layout of all arrays in the input program.

The algorithm partitions the program into code segments, called phases. A phase represents a computation that does not require any data remapping. Alignment and distribution analysis are based on the phase concept. The initial automatic data layout algorithm generates alignment and distribution specifications for each phase.

Alignment and distribution analysis use a search space approach to determine a set of good data layout schemes for each phase. The final selection of a single data layout scheme for each phase considers the profitability of data remapping between phases. Each layout scheme in the local search space of a phase is evaluated by a static performance estimator. The static performance estimator is also used to determine the costs for data remapping between phases. The final data layout selection is formulated as a cost minimization problem over a weighted graph, where each node represents a layout scheme and edges represent cost functions for data remapping between schemes. A more detailed discussion can be found in [KMCKC93, Kre93b, Kre93a].

In this report, we will describe the general concept of our search space approach. We will provide the sizes of the exhaustive search spaces for alignment and distribution under

*This research was supported by the Center for Research on Parallel Computation (CRPC), a Science and Technology Center funded by NSF through Cooperative Agreement Number CCR-9120008. This work was also sponsored by DARPA under contract #DABT63-92-C-0038, and the IBM corporation. The content of this paper does not necessarily reflect the position or the policy of the U.S. Government and no official endorsement should be inferred.

†Corresponding author; e-mail: kremer@cs.rice.edu

some simplifying assumptions. We will discuss pruning heuristics for alignment analysis and will determine the maximal sizes of the resulting search spaces. As an indication of the feasibility of our approach, we will give the sizes of the exhaustive and pruned search spaces for two programs, namely Erlebacher (3-dimensional tridiagonal solver) and ADI (alternate direction implicit integration).

The report is organized as follows. The discussion of the phase concept is followed by a discussion of search space approaches to alignment and distribution analysis. The report is concluded with a study of the different approaches using the Erlebacher and ADI codes as examples.

1 Phase Concept

The concept of a phase is central to our search space approach. The following working definition of a phase is given in [Kre93a]:

Definition 1 *A phase is a loop nest such that for each induction variable that occurs in a subscript position of an array reference in the loop body, the phase contains the surrounding loop that defines the induction variable.*

A phase is minimal in the sense that it does not include surrounding loops that do not define induction variables occurring in subscript positions. In this report, we will ignore code outside of phases.

Although this definition works well in the context of codes like Erlebacher and ADI, it might not be sufficient for codes that exhibit a structure as described in the following example. A similar use of data structures occur in real codes like UTCOMP (oil reservoir simulation) and ARPS (regional weather prediction).

Example 1

```
real a(n,n,p)
. . .
do k = 1, p
  do j = 1, n
    do i = 2, n
      a(i,j,k) = a(i-1,j,k) + a(i+1,j,k)
    enddo
  enddo
  do j = 2, n
    do i = 1, n
      a(i,j,k) = a(i,j-1,k) + a(i,j+1,k)
    enddo
  enddo
enddo
```

If p is a small constant value, redistribution between the two innermost loops might be profitable. However, the working definition identifies the entire loop nest as a single phase.

In the context of this report, we will assume that the program is written in a style that allows the working definition to expose all potentially profitable points of data remapping in the program. In the above example, if $p = 2$, then unrolling the outermost loop and introducing a separate array for each unrolled loop body will result in phases with the right granularity. The transformed code is shown below.

```
real a1(n,n), a2(n,n)
. . .
do j = 1, n
  do i = 2, n
    a1(i,j) = a1(i-1,j) + a1(i+1,j)
  enddo
enddo
do j = 2, n
  do i = 1, n
    a1(i,j) = a1(i,j-1) + a1(i,j+1)
  enddo
enddo
do j = 1, n
  do i = 2, n
    a2(i,j) = a2(i-1,j) + a2(i+1,j)
  enddo
enddo
do j = 2, n
  do i = 1, n
    a2(i,j) = a2(i,j-1) + a2(i,j+1)
  enddo
enddo
```

Program transformations to facilitate automatic data layout are a topic of future research. For our initial data layout tool, we assume that a user will perform these transformation by hand, i.e. the tool considers these transformations a part of the process of rewriting the input program in a data-parallel programming style.

2 Alignment Analysis

In this report, alignments are restricted to inter-dimensional alignments only, i.e. transposes and embeddings. Offset alignments or stride alignments are not considered [KLS90]. The output of the alignment analysis are sets of alignment schemes, one set for each phase. An alignment scheme for a phase specifies an alignment of all arrays referenced in the phase. A phase may not reference all data objects in the program. All alignment schemes are specified

relative to the same *alignment space*. The alignment space of a program is determined by the maximal dimensionalities and maximal dimensional extents of its arrays.

In the remainder of this section, we will give the size of the exhaustive alignment search space, followed by a discussion of different methods to prune the exhaustive search space. We call a pruned search space a *candidate search space*, since its construction may involve the pruning and/or the generation of alignment schemes.

2.1 Exhaustive Alignment Search Space

For our complexity discussions, we assume that an embedding does only specify the aligned dimensions, not the position of the embedded object with respect to the non-aligned dimensions. This assumption allows us to treat every array as a d-dimensional array, where d is the dimensionality of the alignment space of the program. There are at most as many possibilities to embed a k-dimensional array in a d-dimensional alignment space, $1 \leq k \leq d$, as there are possibilities to align a d-dimensional array in a d-dimensional alignment space. Under the above assumptions our complexity estimates are conservative in the sense that they overestimate the size of the alignment search space.

Lemma 1 *Assume that there are $s(x)$ distinct arrays accessed in phase x, all of which have d dimensions. Let X denote the set of all phases in the program. The exhaustive alignment search space has the following maximal size:*

$$\sum_{x \ in \ X} (d!)^{s(x)}$$

Proof: To specify the alignment of an array for the entire program it is sufficient to specify its alignment for all phases in which it is referenced. Due to the assumptions of the lemma, phase x references $s(x)$, d-dimensional arrays. Every possible permutation of d-dimensions for $s(x)$ arrays have to be considered in an exhaustive search space for the phase. The exhaustive alignment search space for phase x contains $(d!)^{s(x)}$ distinct alignments with respect to the program's alignment space.
□

Example 2 Assume a program has 100 phases, a 3-dimensional alignment space, and at most five distinct arrays referenced in each phase. The exhaustive search space has a maximal size of $100 * 6^5 = 777,600$.

2.2 Candidate Alignment Search Space

The candidate search space approach uses rules to select alignment schemes in the exhaustive search space. The basic algorithm works as follows. In the first step, called *intra-phase* alignment, alignment schemes of arrays accessed in each single phase are selected. As a result, each phase is associated with an alignment search space local to the phase. In the second step, local alignment search spaces are combined. This merging step is called *inter-phase* alignment.

Due to the basic structure of the alignment algorithm, different selection rules may be applied during the intra-phase and inter-phase alignment. In the following we will discuss different selection rules and their impact on the sizes of the resulting candidate alignment search spaces.

2.2.1 Intra-Phase Alignment

All of the discussed strategies to generate an intra-phase alignment search space look at individual statements in the phase first. For simplicity, we assume that alignment conflicts do not occur inside individual statements. Therefore, each statement generates a single alignment requirement. The following strategies may generate alignment search spaces of different sizes. The list of strategies is not complete.

Strategy A: Include Compromise Alignments

Each alignment scheme generated by a statement is added into the search space together with all compromise schemes between any pair of statments. No conflict resolution is performed. Compromise alignment schemes are defined as follows:

Definition 2 *A compromise alignment scheme is an alignment scheme that is not generated directly by a computation pattern in the program but which represents a compromise between alignment schemes directly generated by two or more computation patterns.*

Example 3

```
do i
  do j
    a(i,j) = b(i,j) + c(i,j)
    a(i,j) = b(j,i) + c(j,i)
  enddo
enddo
```

Alignment schemes that are generated by the statements are $\{A, B, C\}$ and $\{A, B^T, C^T\}$ for the first and second statement, respectively. Compromise alignment schemes are $\{A, B, C^T\}$ and $\{A, B^T, C\}$. The number of compromise alignment schemes can be exponential in the number of data objects accessed in the phase.

Strategy B: Individual Statement Alignments Only

Each alignment scheme generated by a statement is added into the search space. No conflict resolution is performed and no compromise alignment schemes are introduced.

Each alignment scheme of a phase has to specify the alignments of all data objects accessed in the phase. The following example will give some insights into the size of the candidate search space:

Example 4

```
do i
  do j
    a(i,j) = b(i,j)
    b(i,j) = a(j,i)

    c(i,j) = d(i,j)
    d(i,j) = c(j,i)

    e(i,j) = f(i,j)
    f(i,j) = e(j,i)
  enddo
enddo
```

The first and second statements generate the alignment $\{A, B\}$ and $\{A^T, B\}$, the third and forth statements generate the alignment $\{C, D\}$ and $\{C^T, D\}$, and the fifth and sixth statements generate the alignment $\{E, F\}$ and $\{E^T, F\}$. Without conflict resolution, this will result in eight possible alignments for the phase, namely:

$\{A, B, C, D, E, F\}$,
$\{A, B, C, D, E^T, F\}$,
$\{A, B, C^T, D, E, F\}$,
$\{A, B, C^T, D, E^T, F\}$,
$\{A^T, B, C, D, E, F\}$,
$\{A^T, B, C, D, E^T, F\}$,
$\{A^T, B, C^T, D, E, F\}$, and
$\{A^T, B, C^T, D, E^T, F\}$.

Even without adding any compromise schemes, the number of schemes for the phase will be exponential in the number of data objects accessed.

Strategy C: Conflict Resolution

This strategy has been used in the Crystal compiler developed at Yale University [LC90]. Following this strategy, the resulting alignment search space will contain only a single alignment scheme for each phase. Example 5 shows that Jingke Li's component affinity graph (CAG) has to be extended to work correctly in the context of the owner-computes rule.

Example 5

```
do i
  do j
    a(i,j) = b(j,i)
    b(i,j) = a(j,i)
    a(i,j) = b(i,j) + c(i,j)
  enddo
enddo
```

Using the alignment $\{A, B^T, C\}$ instead of $\{A, B, C\}$ in the code shown above avoids communication of elements of array a. Only elements of array b have to be communicated due to the last statement in the phase. In contrast, the alignment $\{A, B, C\}$ may lead to communication of elements of a and b due to the first two statements. Li's weighted CAG does not allow to distinguish these two alignment schemes. Each edge has a weight of 1. The CAG for the example code is shown in Figure 1.

For our initial data layout framework we will use conflict resolution for intra-phase alignment. A directed and/or weighted CAG seems to be a promising representation to overcome the deficiencies of Li's CAG. We are currently working on a new CAG representation.

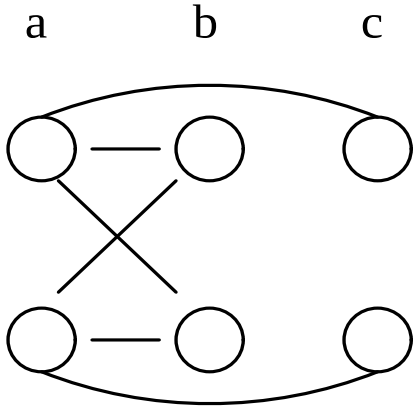

Figure 1: CAG for example code

2.2.2 Inter-Phase Alignment

The inter-phase alignment step can use strategies that introduce or eliminate alignment schemes based on the sets of alignment schemes determined for each phase in the intra-phase alignment step. Selection strategies may use compromise alignments, perform conflict resolution, or a combination of both. In this section, we will only discuss a selection strategy that is based on conflict resolution for intra-phase alignment.

Importing Alignment Schemes

Once a single alignment scheme has been determined for each phase, the schemes will be compared across phases. The motivation for importing alignment schemes is to allow the local consideration of alignment schemes that are efficient in other parts of the program.

For any pair of phases the alignment schemes are intersected and a new scheme is introduced if the intersection is non-empty and alignments of the data objects in the intersection differ in the two phases. The new scheme represents the alignment requirements of the other phase. We say that the new scheme is *imported* into the phase. Data objects of a phase that are not in the intersection have to be aligned in a way that is good for the imported scheme in order to produce a good alignment scheme for the entire phase.

A conflict resolution algorithm has to be used to merge the alignments of data objects in the intersection and outside of the intersection for each imported alignment scheme. To do this, the alignment graph of the phase is modified according to the imported scheme. All edges between data objects in the intersection are first removed and then substituted by edges that reflect the imported alignment scheme, i.e. the imported alignment subgraph has no conflicts and has the imported alignment scheme as its only alignment solution.

2.2.3 Maximal Size of Candidate Search Space

We assume that conflict resolution is used for intra-phase alignment and that alignment schemes are imported in the inter-phase alignment step. All arrays are assumed to have the same number of dimensions as the alignment space of the program.

Lemma 2 *Assume that the program has p phases. Let X denote the set of all phases in the program. Further assume that there are $s(x)$ distinct arrays accessed in phase x, all of which have d dimensions. The candidate alignment search space has the following maximal size:*

$$\sum_{x \ in \ X} MIN(\ (d!)^{s(x)}, p\ )$$

Proof: Prior to inter-phase alignment the local search spaces contain a single alignment scheme for each phase. During inter-phase alignment, each phase may import $p-1$ alignment schemes, one from each of the other phases, resulting in a local search space of size p.

The size of the local search space is bounded from above by the size of its exhaustive search space.
□

Example 6 Assume a program has 100 phases, a 3-dimensional alignment space, and at most five distinct arrays referenced in each phase. As shown in example 2 in Section 2.1, the exhaustive search space has a maximal size of 777,600. In contrast, the candidate search space has a maximal size of $p^2 = 10,000$, since $6^5 = 7776 > 100$.

2.3 Our Initial Approach

For real application programs, we expect only a small number of alignment conflicts. We will use conflict resolution for intra-phase alignment. In the inter-phase alignment step, local alignment schemes will be merged across phases as long as no conflicts occur. A single CAG represents the alignment requirements of all phases involved in the conflict-free merging. The alignment scheme of a single phase is the alignment scheme generated by the merged CAG restricted to the arrays referenced in the phase. Once a conflict occurs, the CAG of the conflicting phase will not be integrated but a new merging process will be started with the CAG that introduced the conflict as its initial representation. We are currently investigating how to merge the CAGs of phases in order to maximize the number of conflict-free merges.

As the result of this algorithm, a set of alignment schemes will be determined, each of which may represent the alignment requirements of a group of phases. Subsequently, every alignment scheme will import the alignment requirements of the other schemes.

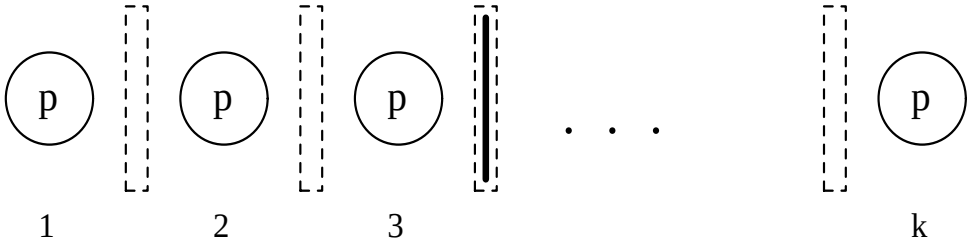

Figure 2: Possible partitions of k objects representing $p * p * p * ... * p = p^k$ processors

3 Distribution Analysis

A *distribution scheme* of the program's alignment space is characterized by (1) the distributed dimensions, and (2) the number of processors in each distributed dimension. We are only considering `BLOCK` distributions. In this section, we will discuss the sizes of different exhaustive distribution search spaces and issues related to a candidate search space.

3.1 Exhaustive Distribution Search Space

Lemma 3 *Let* d *denote the number of dimensions of the alignment space. Assuming that at least one dimension has to be partitioned, the number of possible dimensional partitionings,* part*, is*

$$\text{part} = \sum_{i=1}^{d} \left(\begin{array}{c} d \\ i \end{array} \right) = 2^d - 1$$

Proof The index i keeps track of the number of dimensions to be partitioned. For any given i, $1 \leq i \leq d$, there are

$$\left(\begin{array}{c} d \\ i \end{array} \right)$$

choices of i dimensions out of d dimensions. □

Lemma 4 *Let* procs *denote the number of processors used. Assume that the number of processors is a power of some prime number p, i.e. procs* $= p^k$*. Assuming that at least one dimension is partitioned, the number of distribution schemes for a d-dimensional alignment space,* size*, is*

$$\text{size} \;=\; \sum_{i=1}^{d} \left(\begin{array}{c} d \\ i \end{array} \right) * \left(\begin{array}{c} k-1 \\ i-1 \end{array} \right) \;=\; \left(\begin{array}{c} k+d-1 \\ d-1 \end{array} \right)$$

Proof The index i keeps track of the number of dimensions to be partitioned. There are $k-1$ possible choices to place a partition for p^k processors to create two groups of processors as shown in Figure 2. For any given i, $1 \leq i \leq d$, there are

$$\left(\begin{array}{c} k-1 \\ i-1 \end{array} \right)$$

choices of $i-1$ slots out of $k-1$ possible slots to divide the number of processors into i distinct, non-empty groups, each assigned to one distributed dimension. Note that the number of choices is 0 if $i > k$. The resulting values for *size* form a Pascal's triangle. □

The following table gives the values of *size* for different numbers of processors used and alignment spaces of up to seven dimensions. The number of processors is assumed to be a power of two, i.e. $procs = 2^k$.

#procs	#dimensions						
	1	2	3	4	5	6	7
4	1	3	6	10	15	21	28
8	1	4	10	20	35	56	84
16	1	5	15	35	70	126	210
32	1	6	21	56	126	252	462
64	1	7	28	84	210	462	924
128	1	8	36	120	330	792	1716
256	1	9	45	165	495	1287	3003
512	1	10	55	220	715	2002	5005
1024	1	11	66	286	1001	3003	8008

Lemma 5 *Let d be constant. Then* size $= \Theta(k^{d-1})$.

Proof Define constant c as $d-1$.

$$\left(\begin{array}{c} k+c \\ c \end{array} \right) \;=\; \frac{(k+c)!}{c!\;k!} \;=\; \frac{1}{c!} \;*\; \prod_{i=1}^{c}(k+i) \;=:\; T$$

$$T \leq \frac{1}{c!} \;*\; (k+c)^c \;=\; O(k^c)$$

$$T \geq \frac{1}{c!} \;*\; k^c \;=\; \Omega(k^c)$$

Substituting $d-1$ for c in $\Omega(k^c) \leq T \leq O(k^c)$ yields the claimed result. □

Lemma 6 *Let* procs *denote the number of processors used. Assume that the number of processors is a power of some prime number p multiplied by a prime number c, i.e. $procs = p^k * c$, with $c \neq p$. Assuming that at least one dimension is partitioned, the number of distribution schemes for a* d*-dimensional alignment space ,* size*, is*

$$\text{size} \;=\; \sum_{i=1}^{d} \left(\begin{array}{c} d \\ i \end{array} \right) * \left(\begin{array}{c} k-1 \\ i-1 \end{array} \right) * i \;+\; \sum_{i=2}^{d} \left(\begin{array}{c} d \\ i \end{array} \right) * \left(\begin{array}{c} k-1 \\ i-2 \end{array} \right) * i$$

$$= d * \begin{pmatrix} k+d-1 \\ d-1 \end{pmatrix}$$

Proof The index i keeps track of the number of dimensions to be partitioned. The first sum gives the number of possible distribution schemes if each distributed dimension is assigned a number of processors that is a multiple of p. In terms of Figure 2, the additional factor c can end up in any of the i groups created by the partitioning. The second sum takes care of the case where a single dimension has exactly c processors assigned to it. This means that the p^k processors need to be partitioned into only $i-1$ groups, since the i-th group will consist of c alone. There are i possible positions for the group consisting of c with respect to the other $i-1$ groups.

The proof of the second part of the equation is as follows:

(1) :

$$\sum_{i=1}^{d} \begin{pmatrix} d \\ i \end{pmatrix} * \begin{pmatrix} k-1 \\ i-1 \end{pmatrix} * i \quad \sum_{i=1}^{d} \frac{d!}{i!\ (d-i)!} * \begin{pmatrix} k-1 \\ i-1 \end{pmatrix} * i$$

$$= \sum_{i=1}^{d} \frac{d!}{(i-1)!\ (d-i)!} * \begin{pmatrix} k-1 \\ i-1 \end{pmatrix}$$

$$= d * \sum_{i=1}^{d} \frac{(d-1)!}{(i-1)!\ (d-i)!} * \begin{pmatrix} k-1 \\ i-1 \end{pmatrix}$$

$$= d * \sum_{i=1}^{d} \begin{pmatrix} d-1 \\ i-1 \end{pmatrix} * \begin{pmatrix} k-1 \\ i-1 \end{pmatrix}$$

$$= d * \begin{pmatrix} d-1 \\ d-1 \end{pmatrix} * \begin{pmatrix} k-1 \\ d-1 \end{pmatrix} + d * \sum_{i=1}^{d-1} \begin{pmatrix} d-1 \\ i-1 \end{pmatrix} * \begin{pmatrix} k-1 \\ i-1 \end{pmatrix}$$

$$= d * \begin{pmatrix} d \\ d \end{pmatrix} * \begin{pmatrix} k-1 \\ d-1 \end{pmatrix} + d * \sum_{i=1}^{d-1} \begin{pmatrix} d-1 \\ i-1 \end{pmatrix} * \begin{pmatrix} k-1 \\ i-1 \end{pmatrix}$$

(2) :

$$\sum_{i=2}^{d} \begin{pmatrix} d \\ i \end{pmatrix} * \begin{pmatrix} k-1 \\ i-2 \end{pmatrix} * i = \sum_{i=2}^{d} \frac{d!}{i!\ (d-i)!} * \begin{pmatrix} k-1 \\ i-2 \end{pmatrix} * i$$

$$= \sum_{i=2}^{d} \frac{d!}{(i-1)!\ (d-i)!} * \begin{pmatrix} k-1 \\ i-2 \end{pmatrix}$$

$$= d * \sum_{i=2}^{d} \frac{(d-1)!}{(i-1)!\ (d-i)!} * \begin{pmatrix} k-1 \\ i-2 \end{pmatrix}$$

$$= d * \sum_{i=2}^{d} \binom{d-1}{i-1} * \binom{k-1}{i-2}$$

$$= d * \sum_{i=1}^{d-1} \binom{d-1}{i} * \binom{k-1}{i-1}$$

(1) + (2) :

$$\text{size} = d * \binom{d}{d} * \binom{k-1}{d-1} + d * \sum_{i=1}^{d-1} \binom{k-1}{i-1} * \left[\binom{d-1}{i-1} + \binom{d-1}{i} \right]$$

$$= d * \binom{d}{d} * \binom{k-1}{d-1} + d * \sum_{i=1}^{d-1} \binom{k-1}{i-1} * \binom{d}{i}$$

$$= d * \sum_{i=1}^{d} \binom{d}{i} * \binom{k-1}{i-1}$$

$$= d * \binom{k+d-1}{d-1} \text{ (due to Lemma 4).}$$

□

The following table gives the values of *size* for different numbers of processors used and alignment spaces of up to seven dimensions. The number of processors is assumed to be a power of two times a single number, i.e. $procs = 2^k * c$, where c is relative prime to two and no factorization of c is allowed.

#procs	#dimensions						
	1	2	3	4	5	6	7
$2 * c$	1	4	9	16	25	36	49
$4 * c$	1	6	18	40	75	126	196
$8 * c$	1	8	30	80	175	336	588
$16 * c$	1	10	45	140	350	756	1470
$32 * c$	1	12	63	224	630	1512	3234
$64 * c$	1	14	84	336	1050	2772	6468
$128 * c$	1	16	108	480	1650	4752	12012
$256 * c$	1	18	135	660	2475	7722	21021
$512 * c$	1	20	165	880	3575	12012	35035

Lemma 7 *Let d be constant. Then* size $= \Theta(k^{d-1})$.

Proof Follows immediately from Lemma 5 and Lemma 6. □

Using the results from Lemma 4 and Lemma 6, the following table lists the sizes of search spaces for different numbers of processors.

#procs	#dimensions						
	1	2	3	4	5	6	7
$8 = 2^3$	1	4	10	20	35	56	84
$16 = 2^4$	1	5	15	35	70	126	210
$24 = 2^3 * 3$	1	8	30	80	175	336	588
$32 = 2^5$	1	6	21	56	126	252	462
$40 = 2^3 * 5$	1	8	30	80	175	336	588
$48 = 2^4 * 3$	1	10	45	140	350	756	1470
$56 = 2^3 * 7$	1	8	30	80	175	336	588
$64 = 2^6$	1	7	28	84	210	462	924
$96 = 2^5 * 3$	1	12	63	224	630	1512	3234
$128 = 2^7$	1	8	36	120	330	792	1716
$192 = 2^6 * 3$	1	14	84	336	1050	2772	6468
$256 = 2^8$	1	9	45	165	495	1287	3003
$512 = 2^9$	1	10	55	220	715	2002	5005
$576 = 2^6 * 9$	1	14	84	336	1050	2772	6468
$768 = 2^8 * 3$	1	18	135	660	2475	7722	21021
$1024 = 2^{10}$	1	11	66	286	1001	3003	8008

It is not clear whether an exhaustive search space based on $p = 2^k$ or $p = 2^k * c$ processors can be used as an approximation to some search space where the number of processors cannot be expressed by one of the factorizations. The answer to this question is still open.

3.2 Candidate Distribution Search Space

A candidate distribution search space is constructed in two steps similar to the construction of the candidate alignment search space (see Section 2.2). Assume that the alignment space of the program has d dimensions. Let $a(x)$ denote the size of the alignment search space for phase x, i.e. there are $a(x)$ alignment schemes associated with phase x after alignment analysis.

3.2.1 Intra-Phase Distribution

The selection rules for candidate distribution schemes for each phase can be based on a set of cost functions, one cost function for each possible combination of distributed dimensions of a given alignment scheme. The cost functions are parameterized with respect to the number

of processors in each distributed dimension. Each cost function determines the execution time of the phase for an alignment scheme and a distribution. There are $(2^d - 1) * a(x)$ such cost functions for each phase.

We are currently investigating different selection strategies. One possibility is to minimize each individual cost function. The minimization may be performed using techniques based on a hill-climbing method in combination with static performance estimation [Who92, BFKK91]. The resulting search space contains a data layout for each alignment scheme in the alignment search space. A data layout consists of an alignment scheme together with the distribution scheme that minimizes its estimated performance. Therefore, the search space contains a(x) data layout schemes.

3.2.2 Inter-Phase Distribution

One possible approach to inter-phase distribution is not to perform any propagation of distribution schemes across the program. The resulting final data layout search space of the entire program is the union of search spaces of each phase. Strategies for inter-phase distribution are a main topic of our current research.

4 Case Study

Erlebacher is a benchmark program written by Thomas Eidson at the Institute for Computer Applications in Science and Engineering (ICASE). It performs 3-dimensional tridiagonal solves using Alternating Direction Implicit (ADI) integration. For this study we are using an inlined version of the code, written in data-parallel programming style. The code has 300 lines without comments. To transform the original program into data-parallel style, we performed array kill analysis by hand and renamed and replicated arrays appropriately.

The second code also uses Alternating Direction Implicit (ADI) integration. This code has been provided by Charles Koelbel at the Center for Research on Parallel Computation (CRPC) at Rice University. Both codes are shown in the appendix.

4.1 Erlebacher

The program has 40 phases and a 3-dimensional alignment space. The following table gives the distribution of the phases with respect to the number of distinct arrays they reference.

#arrays:	1	2	3	4	5
#phases:	4	18	15	0	3

The exhaustive alignment search space contains $4*3!+18*(3!)^2+15*(3!)^3+3*(3!)^5 = 27240$ alignment schemes. There are no intra-phase alignment conflicts. Inter-phase conflicts may result from different embeddings of the same array into the alignment space of the program. This is a weakness of the naive algorithm that imports alignment schemes in the inter-phase alignment step. This weakness is due to the fact that the intra-phase step requires a

selection of a single alignment scheme. In the case of an embedding, this requirement forces a premature choice of a single embedding for a phase. In contrast, our proposed algorithm will generate a single alignment scheme for the entire program since no alignment conflicts occur while merging the CAGs of each phase. Therefore, our proposed alignment search space contains 40 alignment schemes, one scheme for each phase.

The Erlebacher code shows that our candidate search space approach should be able to recognize total embeddings, i.e. an alignment of the embedded object with a subspace of the alignment space. Such an embedding is profitable, for instance, in the case of arrays dux and wrk$x, where wrk$x(i,j) should be aligned with the subspace dux(64,i,j).

Assuming 64 processors, the exhaustive distribution search space contains 28 distribution schemes for the 3-dimensional alignment space. Therefore, the space of all considered data layouts has a size of $28 * 27240 = 762720$. In contrast, an approach that uses our candidate alignment search space and an exhaustive distribution search space will have to consider $28 * 40 = 1120$ data layout schemes. Using a candidate distribution search space may reduce the number of candidate data layouts even further.

4.2 ADI

The program has 6 phases and a 2-dimensional alignment space. The following table gives the distribution of the phases with respect to the number of distinct arrays they reference.

#arrays:	1	2	3
#phases:	0	2	4

The exhaustive alignment search space contains $2 * (2!)^2 + 4 * (2!)^3 = 40$ alignment schemes. There are no intra-phase or inter-phase alignment conflicts. The program contains only 2-dimensional arrays and therefore no embedding is required. Our candidate search space approach will generate a single alignment scheme for all 6 phases. Therefore, the search space contains 6 alignment schemes.

Assuming 64 processors, the exhaustive distribution search space contains 7 distribution schemes for the 2-dimensional alignment space. Therefore, the space of all considered data layouts has a size of $7 * 40 = 280$. In contrast, an approach that uses our candidate alignment search space and an exhaustive distribution search space will have to consider $7 * 6 = 42$ data layout schemes. As in the Erlebacher example, using a candidate distribution search space may reduce the number of candidate data layouts even further.

Acknowledgement

We would like to thank Seema Hiranandani and John Mellor-Crummey for their helpful comments.

References

[BFKK91] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. A static performance estimator to guide data partitioning decisions. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Williamsburg, VA, April 1991.

[FHK+90] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Technical Report TR90-141, Dept. of Computer Science, Rice University, December 1990.

[KLS90] K. Knobe, J. Lukas, and G. Steele, Jr. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8(2):102–118, February 1990.

[KMCKC93] U. Kremer, J. Mellor-Crummey, K. Kennedy, and A. Carle. Automatic data layout for distributed-memory machines in the D programming environment. In Christoph W. Kessler, editor, *Automatic Parallelization — New Approaches to Code Generation, Data Distribution, and Performance Prediction*, pages 136–152. Vieweg Advanced Studies in Computer Science, Verlag Vieweg, Wiesbaden, Germany, 1993. Also available as technical report CRPC-TR93-298-S, Rice University.

[Kre93a] U. Kremer. Automatic data layout for distributed-memory machines. Technical Report CRPC-TR93-299-S, Center for Research on Parallel Computation, Rice University, February 1993. (thesis proposal).

[Kre93b] U. Kremer. NP-completeness of dynamic remapping. In *Proceedings of the Fourth Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, December 1993. Also available as technical report CRPC-TR93-330-S (D Newsletter #8), Rice University.

[LC90] J. Li and M. Chen. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. In *Frontiers90: The 3rd Symposium on the Frontiers of Massively Parallel Computation*, College Park, MD, October 1990.

[Who92] S. Wholey. Automatic data mapping for distributed-memory parallel computers. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.

Appendix: Erlebacher code (inlined)

```
      program testh
c
      parameter (ioprt=6)
c
      real u(64,64,64),dux(64,64,64),duy(64,64,64),duz(64,64,64)
      real dx,dy,dz
      real wrk$x(64,64),wrk$y(64,64),wrk$z(64,64)
      real a$x(64),b$x(64),c$x(64),d$x(64),e$x(64)
      real a$y(64),b$y(64),c$y(64),d$y(64),e$y(64)
      real a$z(64),b$z(64),c$z(64),d$z(64),e$z(64)
      double precision dt,dtx,dty,dtz,dclock
c
      dx=0.4
      dy=0.5
      dz=0.6
c
c==BEGIN==call dotest=============================================—
c
c*** generate data
c
c==BEGIN==call genvar=============================================—
c
      do 10 kk=1,64
        do 10 jj=1,64
          do 10 ii=1,64
            u(ii,jj,kk)   =0.0
   10 continue
c
c..END....call genvar.....................................—
c
c*** time derivative calculation
c
      call gsync()
      dtx = dclock()
c==BEGIN==call dx3d6p(dx)=========================================—
c
c   routine:
c      computes the x-derivatives of u storing the result in ud
c      uses a 6th order Pade scheme (Periodic domain)
c      periodic in x (evenly spaced grid)
c      first index corresponds to x direction
c
c  first derivative
c
        c13    = 1./3.
        c79dx  = 7./(9.*dx)
        c136dx = 1./(36.*dx)
c
c  set up the left hand side and factor the periodic matrix
c
        do 130 i=1,64
          a$x(i) = c13
          b$x(i) = 1.
          c$x(i) = c13
  130   continue
c==BEGIN==call tridvpf(a$x,b$x,c$x,d$x,e$x)=====================—
c      factorization stage of vectorized tridiagonal solver
c      equations are periodic
c      all equations have the same coefficients

c     generate first elements of LU
c
c     b$x(1) = 1.0/b$x(1)
      b$x(1) = 1.0
      c$x(1) = c$x(1)*b$x(1)
      e$x(1) = a$x(1)*b$x(1)
      d$x(1) = c$x(64)
c
c     generate n=2 to n=64-2 elements of LU
c
      do 110 k=2,64-2
        b$x(k) = b$x(k) - a$x(k)*c$x(k-1)
        b$x(k) = 1.0/b$x(k)
        c$x(k) = c$x(k)*b$x(k)
        e$x(k) = -a$x(k)*e$x(k-1)*b$x(k)
        d$x(k) = -d$x(k-1)*c$x(k-1)
  110 continue
c
c     generate 64-1 elements
c
      b$x(64-1) = b$x(64-1) - a$x(64-1)*c$x(64-2)
      b$x(64-1) = 1.0/b$x(64-1)
      e$x(64-1) = (c$x(64-1) - a$x(64-1)*e$x(64-2))*b$x(64-1)
      d$x(64-1) = a$x(64) - d$x(64-2)*c$x(64-2)
```

```
c
c     generate the n-th element
c
      tot = 0.
      do 120 k=1,64-1
        tot = tot + d$x(k)*e$x(k)
  120 continue
      b$x(64) = b$x(64) - tot
c     b$x(64) = 1.0/b$x(64)
c..END....call tridvpf(a$x,b$x,c$x,d$x,e$x).....................—
c
c  set up right hand side and do forward/backward substitution
c
        do 140 k=1,64
        do 140 j=1,64
          dux(1,j,k) = c79dx*(u(2,j,k) - u(64,j,k))
     &             + c136dx*(u(3,j,k) - u(64-1,j,k))
          dux(2,j,k) = c79dx*(u(3,j,k) - u(1,j,k))
     &             + c136dx*(u(4,j,k) - u(64,j,k))
  140    continue
        do 141 k=1,64
        do 141 j=1,64
          dux(64-1,j,k) = c79dx*(u(64,j,k) - u(64-2,j,k))
     &                  + c136dx*(u(1,j,k)    - u(64-3,j,k))
          dux(64,j,k) = c79dx*(u(1,j,k) - u(64-1,j,k))
     &                + c136dx*(u(2,j,k) - u(64-2,j,k))
  141    continue
c
        do 150 k=1,64
        do 150 j=1,64
        do 150 i=3,64-2
            dux(i,j,k) = c79dx*(u(i+1,j,k) - u(i-1,j,k))
     &               + c136dx*(u(i+2,j,k) - u(i-2,j,k))
  150    continue
c==BEGIN==call tridvpi(a$x,b$x,c$x,d$x,e$x)=====================—
c
c     perform forward substitution
c
      do 210 k=1,64
      do 210 j=1,64
        dux(1,j,k) = dux(1,j,k)*b$x(1)
  210 continue
c
      do 220 k=1,64
      do 220 j=1,64
      do 220 i=2,64-1
        dux(i,j,k)=(dux(i,j,k)-a$x(i)*dux(i-1,j,k))*b$x(i)
  220 continue
c
      do 230 k=1,64
      do 230 j=1,64
        wrk$x(j,k) = 0.
  230 continue
c
      do 240 k=1,64
      do 240 j=1,64
      do 240 i=1,64-1
        wrk$x(j,k) = wrk$x(j,k) + d$x(i)*dux(i,j,k)
  240 continue
c
      do 250 k=1,64
      do 250 j=1,64
        dux(64,j,k) = (dux(64,j,k) - wrk$x(j,k))*b$x(64)
  250 continue
c
c     perform backward substitution
c
      do 260 k=1,64
      do 260 j=1,64
        dux(64-1,j,k)=dux(64-1,j,k) - e$x(64-1)*dux(64,j,k)
  260 continue
c
      do 270 k=1,64
      do 270 j=1,64
      do 270 i=64-2,1,-1
        dux(i,j,k)=
     *     dux(i,j,k)-c$x(i)*dux(i+1,j,k)-e$x(i)*dux(64,j,k)
  270 continue
c
c..END....call tridvpi(a$x,b$x,c$x,d$x,e$x).....................—
c..END....call dx3d6p(dx).....................................—
      dtx = dclock() - dtx
      call gdhigh(dtx, 1, dt)
c
      call gsync()
      dty = dclock()
c==BEGIN==call dy3d6p(dy)=======================================—
c
c   routine:
c      computes the y-derivatives of u storing the result in ud
c      uses a 6th order Pade scheme (Periodic domain)
```

```
c      periodic in y (evenly spaced grid)
c      second index corresponds to y direction
c
c  first derivative
c
       c13    = 1./3.
       c79dy  = 7./(9.*dy)
       c136dy = 1./(36.*dy)
c
c  set up the left hand side and factor the periodic matrix
c
       do 330 i=1,64
          a$y(i) = c13
          b$y(i) = 1.
          c$y(i) = c13
  330    continue
c==BEGIN==call tridvpf(a$y,b$y,c$y,d$y,e$y)======================—
c      factorization stage of vectorized tridiagonal solver
c      equations are periodic
c      all equations have the same coefficients

c     generate first elements of LU
c
c     b$y(1) = 1.0/b$y(1)
      b$y(1) = 1.0
      c$y(1) = c$y(1)*b$y(1)
      e$y(1) = a$y(1)*b$y(1)
      d$y(1) = c$y(64)
c
c     generate n=2 to n=64-2 elements of LU
c
      do 310 k=2,64-2
         b$y(k) = b$y(k) - a$y(k)*c$y(k-1)
         b$y(k) = 1.0/b$y(k)
         c$y(k) = c$y(k)*b$y(k)
         e$y(k) = -a$y(k)*e$y(k-1)*b$y(k)
         d$y(k) = -d$y(k-1)*c$y(k-1)
  310 continue
c
c     generate 64-1 elements
c
      b$y(64-1) = b$y(64-1) - a$y(64-1)*c$y(64-2)
      b$y(64-1) = 1.0/b$y(64-1)
      e$y(64-1) = (c$y(64-1) - a$y(64-1)*e$y(64-2))*b$y(64-1)
      d$y(64-1) = a$y(64) - d$y(64-2)*c$y(64-2)
c
c     generate the n-th element
c
      tot = 0.
      do 320 k=1,64-1
         tot = tot + d$y(k)*e$y(k)
  320 continue
      b$y(64) = b$y(64) - tot
c     b$y(64) = 1.0/b$y(64)
c..END....call tridvpf(a$y,b$y,c$y,d$y,e$y).....................—
c
c  set up right hand side and do forward/backward substitution
c
       do 340 k=1,64
       do 340 i=1,64
          duy(i,1,k) = c79dy*(u(i,2,k) - u(i,64,k))
     1             + c136dy*(u(i,3,k) - u(i,64-1,k))
          duy(i,2,k) = c79dy*(u(i,3,k) - u(i,1,k))
     1             + c136dy*(u(i,4,k) - u(i,64,k))
  340    continue
       do 341 k=1,64
       do 341 i=1,64
          duy(i,64-1,k) = c79dy*(u(i,64,k)
     1                         - u(i,64-2,k))
     2                  + c136dy*(u(i,1,k) - u(i,64-3,k))
          duy(i,64,k) = c79dy*(u(i,1,k) - u(i,64-1,k))
     1                + c136dy*(u(i,2,k) - u(i,64-2,k))
  341    continue
c
       do 350 k=1,64
       do 350 j=3,64-2
       do 350 i=1,64
           duy(i,j,k) = c79dy*(u(i,j+1,k) - u(i,j-1,k))
     1                + c136dy*(u(i,j+2,k) - u(i,j-2,k))
  350    continue
c
c==BEGIN==call tridvpj(a$y,b$y,c$y,d$y,e$y)======================—
c
c     perform forward substitution
c
      do 410 k=1,64
      do 410 i=1,64
         duy(i,1,k) = duy(i,1,k)*b$y(1)
  410 continue
c
      do 420 k=1,64
```

```
      do 420 j=2,64-1
      do 420 i=1,64
        duy(i,j,k)=(duy(i,j,k)-a$y(j)*duy(i,j-1,k))*b$y(j)
  420 continue
c
      do 430 k=1,64
      do 430 i=1,64
        wrk$y(i,k) = 0.
  430 continue
c
      do 440 k=1,64
      do 440 j=1,64-1
      do 440 i=1,64
        wrk$y(i,k) = wrk$y(i,k) + d$y(j)*duy(i,j,k)
  440 continue
c
      do 450 k=1,64
      do 450 i=1,64
        duy(i,64,k) = (duy(i,64,k) - wrk$y(i,k))*b$y(64)
  450 continue
c
c     perform backward substitution
c
      do 460 k=1,64
      do 460 i=1,64
        duy(i,64-1,k)=duy(i,64-1,k) - e$y(64-1)*duy(i,64,k)
  460 continue
c
      do 470 k=1,64
      do 470 j=64-2,1,-1
      do 470 i=1,64
        duy(i,j,k)=
     *     duy(i,j,k)-c$y(j)*duy(i,j+1,k)-e$y(j)*duy(i,64,k)
  470 continue
c
c..END....call tridvpj(a$y,b$y,c$y,d$y,e$y).....—
c..END....call dy3d6p(dy).....—
      dty = dclock() - dty
      call gdhigh(dty, 1, dt)
c
      call gsync()
      dtz = dclock()
c==BEGIN==call dz3d6p(dz)=======================================—
c
c   routine:
c      computes the z-derivatives of u storing the result in ud
c      uses a 6th order Pade scheme (Periodic domain)
c      periodic in z (evenly spaced grid)
c      third index corresponds to z direction
c
c  first derivative
c
        c13    = 1./3.
        c79dz  = 7./(9.*dz)
        c136dz = 1./(36.*dz)
c
c  set up the left hand side and factor the periodic matrix
c
        do 530 i=1,64
          a$z(i) = c13
          b$z(i) = 1.
          c$z(i) = c13
  530    continue
c==BEGIN==call tridvpf(a$z,b$z,c$z,d$z,e$z)=====================—
c      factorization stage of vectorized tridiagonal solver
c      equations are periodic
c      all equations have the same coefficients

c     generate first elements of LU
c
c     b$z(1) = 1.0/b$z(1)
      b$z(1) = 1.0
      c$z(1) = c$z(1)*b$z(1)
      e$z(1) = a$z(1)*b$z(1)
      d$z(1) = c$z(64)
c
c     generate n=2 to n=64-2 elements of LU
c
      do 510 k=2,64-2
        b$z(k) = b$z(k) - a$z(k)*c$z(k-1)
        b$z(k) = 1.0/b$z(k)
        c$z(k) = c$z(k)*b$z(k)
        e$z(k) = -a$z(k)*e$z(k-1)*b$z(k)
        d$z(k) = -d$z(k-1)*c$z(k-1)
  510 continue
c
c     generate 64-1 elements
c
      b$z(64-1) = b$z(64-1) - a$z(64-1)*c$z(64-2)
      b$z(64-1) = 1.0/b$z(64-1)
      e$z(64-1) = (c$z(64-1) - a$z(64-1)*e$z(64-2))*b$z(64-1)
```

```
      d$z(64-1) = a$z(64) - d$z(64-2)*c$z(64-2)
c
c     generate the n-th element
c
      tot = 0.
      do 520 k=1,64-1
         tot = tot + d$z(k)*e$z(k)
  520 continue
      b$z(64) = b$z(64) - tot
c     b$z(64) = 1.0/b$z(64)
c..END....call tridvpf(a$z,b$z,c$z,d$z,e$z).....................—
c
c  set up right hand side and do forward/backward substitution
c
        do 540 j=1,64
        do 540 i=1,64
           duz(i,j,1) = c79dz*(u(i,j,2) - u(i,j,64))
     1             + c136dz*(u(i,j,3) - u(i,j,64-1))
           duz(i,j,2) = c79dz*(u(i,j,3) - u(i,j,1))
     1            + c136dz*(u(i,j,4) - u(i,j,64))
  540    continue
        do 541 j=1,64
        do 541 i=1,64
           duz(i,j,64-1) = c79dz*(u(i,j,64)
     1                     - u(i,j,64-2))
     2                + c136dz*(u(i,j,1) - u(i,j,64-3))
           duz(i,j,64) = c79dz*(u(i,j,1) - u(i,j,64-1))
     1               + c136dz*(u(i,j,2) - u(i,j,64-2))
  541    continue
c
        do 550 k=3,64-2
        do 550 j=1,64
        do 550 i=1,64
            duz(i,j,k) = c79dz*(u(i,j,k+1) - u(i,j,k-1))
     1             + c136dz*(u(i,j,k+2) - u(i,j,k-2))
  550    continue
c==BEGIN==call tridvpk(a$z,b$z,c$z,d$z,e$z)=====================—
c
c     perform forward substitution
c
      do 610 j=1,64
      do 610 i=1,64
         duz(i,j,1) = duz(i,j,1)*b$z(1)
  610 continue
c
      do 620 k=2,64-1
      do 620 j=1,64
      do 620 i=1,64
         duz(i,j,k)=(duz(i,j,k)-a$z(k)*duz(i,j,k-1))*b$z(k)
  620 continue
c
      do 630 j=1,64
      do 630 i=1,64
         wrk$z(i,j) = 0.
  630 continue
c
      do 640 k=1,64-1
      do 640 j=1,64
      do 640 i=1,64
           wrk$z(i,j) = wrk$z(i,j) + d$z(k)*duz(i,j,k)
  640 continue
c
      do 650 j=1,64
      do 650 i=1,64
         duz(i,j,64) = (duz(i,j,64) - wrk$z(i,j))*b$z(64)
  650 continue
c
c     perform backward substitution
c
      do 660 j=1,64
      do 660 i=1,64
         duz(i,j,64-1)=duz(i,j,64-1) - e$z(64-1)*duz(i,j,64)
  660 continue
c
      do 670 k=64-2,1,-1
      do 670 j=1,64
      do 670 i=1,64
         duz(i,j,k)=
     *     duz(i,j,k)-c$z(k)*duz(i,j,k+1)-e$z(k)*duz(i,j,64)
  670 continue
c
c..END....call tridvpk(a$z,b$z,c$z,d$z,e$z).....................—
c..END....call dz3d6p(dz).....................................—
      dtz = dclock() - dtz
      call gdhigh(dtz, 1, dt)
c
      dt=dtx+dty+dtz
c
c..END....call dotest.....................................—

c==BEGIN==call prntout=========================================—
```

```
c
      il=64
      jl=64
      kl=64
c
      write(ioprt,2100)
      write(ioprt,2000) il,jl,kl
      write(ioprt,2010) dtx,dty,dtz
      write(ioprt,2020) dt
      write(ioprt,2110)
c
      stop
 2100 format(/' ============== Erlebacher ============'/)
 2000 format(' Grid size (x,y,z): ',3i5)
 2010 format(' Time for derivative(dx,dy,dz):          ',3f12.6)
 2020 format(' Time for derivative(total):             ',f12.6)
 2110 format(/' ======================================='/)
c..END....call prntout.....................................—
      end
```

Appendix: ADI code

```
1     REAL x(N, N), a(N, N), b(N, N)
2     DO iter = 1, MAXITER
3         // ADI forward & backward sweeps along rows
4         DO j = 2, N
5             DO i = 1, N
6                 x(i, j) = x(i, j) − x(i, j−1) ∗ a(i, j) / b(i, j−1)
7                 b(i, j) = b(i, j) − a(i, j) ∗ a(i, j) / b(i, j−1)
8             ENDDO
9         ENDDO
10        DO i = 1, N
11            x(i, N) = x(i, N) / b(i, N)
12        ENDDO
13        DO j = N−1, 1, −1
14            DO i = 1, N
15                x(i, j) = ( x(i, j) − a(i, j+1) ∗ x(i, j+1) ) / b(i, j)
16            ENDDO
17        ENDDO
18        // ADI forward & backward sweeps along columns
19        DO j = 1, N
20            DO i = 2, N
21                x(i, j) = x(i, j) − x(i−1, j) ∗ a(i, j) / b(i−1, j)
22                b(i, j) = b(i, j) − a(i, j) ∗ a(i, j) / b(i−1, j)
23            ENDDO
24        ENDDO
25        DO j = 1, N
26            x(N, j) = x(N, j) / b(N, j)
27        ENDDO
28        DO j = 1, N
29            DO i = N−1, 1, −1
30                x(i, j) = ( x(i, j) − a(i+1, j) ∗ x(i+1, j) ) / b(i, j)
31            ENDDO
32        ENDDO
33    ENDDO
```