Compiler Optimizations for I/O-Intensive Computations

Mahmut Kandemir* Alok Choudhary* J. Ramanujam†

Abstract

This paper describes transformation techniques for out-of-core programs (i.e., those that deal with very large quantities of data) based on exploiting locality using a combination of loop and data transformations. Writing efficient out-of-core program is an arduous task. As a result, compiler optimizations directed at improving I/O performance are becoming increasingly important. We describe how a compiler can improve the performance of the code by determining appropriate file layouts for out-of-core arrays and finding suitable loop transformations. In addition to optimizing a single loop nest, our solution can handle a sequence of loop nests. We also show how to generate code when the file layouts are optimized. Experimental results obtained on an Intel Paragon distributed-memory message-passing multiprocessor demonstrate marked improvements in performance due to the optimizations described in this paper.

1 Introduction

As the speed of the disk subsystem is increasing at a much slower rate than the processor, interconnection network and memory subsystem speeds, any scalable parallel computer system running I/O-intensive applications must rely on some sort of software technology to optimize disk accesses. This is especially true for out-of-core parallel applications where a significant amount of time is spent in waiting for disk access. We believe that the time spent on disk subsystem can be reduced through at least two complementary techniques:

- reducing the number of data transfers between the disk subsystem and main memory, and
- reducing the volume of data transferred between the disk subsystem and main memory.

A user may accomplish these objectives by investing substantial effort trying to understand the peculiarities of the I/O subsystem, studying carefully the file access pattern, and modifying the programs to make them more I/O-conscious. This poses a severe problem in that user intervention based on low-level I/O decisions makes the program less portable. It appears to us that it is possible and in some cases necessary to leave the task of managing I/O to an optimizing compiler for the following reasons. First, current optimizing compilers are quite successful in restructuring the data access patterns of in-core computations (i.e., computations that do not use disk subsystem frequently) such that better cache and memory locality can be achieved. It is reasonable to expect that the same compiler technology can be used (at least partly) in optimizing the performance of the main memory-disk subsystem hierarchy as well. Second, although currently almost each scalable parallel machine has its own parallel file system that comes with a suite of commands to handle I/O, the standardization of I/O interface is underway. In fact, MPI-I/O [8] is a result of such an effort. We believe that an optimizing compiler can easily generate I/O code using such a standard interface much like most compilers for message-passing parallel architectures use MPI to generate communication code.

This paper presents a compiler approach for optimizing I/O accesses in regular scientific codes. Our approach is oriented towards minimizing the number as well as the volume of the data transfers between disk and main memory. It achieves this indirectly by reducing the number of I/O calls made from within the applications. Our experiments show that such an approach can lead to huge savings in disk access times.

The rest of this paper is organized as follows. Section 2 briefly discusses several approaches to locality optimization and summarizes the relevant work on data locality and out-of-core computations. Section 3 introduces our technique which is based on modifying both the loop access pattern and the file layouts and shows through an example how an input program can be transformed to out-of-core code. This technique is a direct extension of an approach originally designed for improving cache locality. We present the algorithm as well as the additional I/O related issues. Section 4 presents performance results on the Intel Paragon at Caltech. Section 5 concludes the paper with a summary and brief outline of the work-in-progress.

2 Related Work

Abu-Sufah et al. [1] were among the first to derive compiler transformations for out-of-core computations. They proposed optimizations to enhance the locality of programs in a virtual memory environment. Our approach is different from theirs in that we rely on explicit I/O rather than leaving the job to the virtual memory management system.

Compilation of out-of-core codes using explicit I/O has been the main focus of several studies [6, 5, 4, 19]. Brezany et al. [6] developed a run-time system called VIPIOS which can be used by an out-of-core compiler. Bordawekar et al. [5, 4] focused on stencil computations which can be re-ordered freely due to lack of flow dependences. They present several algorithms to optimize communication and to indirectly improve the I/O performance of the parallel out-of-core applications. Paleczny et al. [19] incorporate out-of-core compilation techniques into the Fortran D compiler. The main philosophy behind their approach is to choreograph I/O from disks along with the corresponding computation.

These previous techniques were all based on re-ordering the computation rather than on the re-organization of the data in files. In contrast, we show that locality can be significantly improved using a combined approach which includes both loop (iteration space) and data (file layout) transformations. This framework is potentially more powerful than any existing linear transformation technique as it can manipulate both loop and data spaces, and can apply non-singular linear transformations to both spaces. Due to the nature of the domain we are dealing with, our approach includes loop tiling as well.

Cormen and Colvin [9] introduce ViC* (Virtual C*), a preprocessor that transforms a C* program which uses out-of-core data structures into a program with appropriate library calls from ViC* library that read/write data from/to disks. We believe that data layout optimizations are complementary to computation re-ordering optimizations, and there are some programs that can benefit from a combined approach [7].

*CPDC, Department of Electrical and Computer Engineering, Northwestern University, Evanston, IL 60208. e-mail: {mtk,choudhar}@ece.nwu.edu

†Department of Electrical and Computer Engineering, Louisiana State University, Baton Rouge, LA 70803. e-mail: jxr@ee.lsu.edu

2.1 Data Locality Optimizations

Several techniques exist for optimizing data locality in loop nests. In general, these techniques can be divided into three categories: (a) Techniques based on loop transformations alone; (b) Techniques based on data transformations alone; and (c) Techniques based on combined loop and data transformations.

Techniques Based on Loop Transformations

These approaches attempt to improve locality characteristics of programs by modifying the iteration space traversal order. The extent of this modification maybe an important factor in determining the locality behavior of the program. The work in this category can be grouped into two sub-categories described below:

Non-singular Linear Loop Transformations: These transformations can be expressed using square non-singular matrices, called loop transformation matrices. A loop transformation should observe all data dependences [24] in the program. Apart from this, some approaches also require the loop transformation matrix to be of a specific form. The most widely used form is the permutation matrix [24]. This matrix contains only a single 1 in each row and each column; the remaining entries are all zero. Using this transformation matrix, an optimizing compiler can interchange the order of two loops in a loop nest in an attempt to improve spatial and/or temporal locality. A more comprehensive group includes unimodular transformation matrices. A unimodular matrix is an integer matrix with a determinant ± 1. An important characteristic of a unimodular transformation is that it preserves the volume of the iteration space. Li [16] and Ramanujam [20] show how to generate code when a loop nest is transformed using a general non-singular transformation matrix. In principle, the more general the loop transformation matrix is, the more loop nests can be handled with it and also the more difficult to generate code.

Iteration Space Tiling: In its general form tiling decomposes each loop in a loop nest into two loops [24, 23]. The outer loop is called the *tile loop* whereas the inner loop is referred to as the *element loop*. Then the tile loops are hoisted into upper levels in the nest whereas the element loops are positioned inside. Typically, the trip count (number of iterations) for the element loops is chosen such that the amount of data accessed for an execution of all the iterations of the element loops fit in the fastest level of the memory hierarchy. Tiling can be used as a complementary solution along with linear loop transformations. A suggested approach is first to use linear loop transformations to optimize locality and then use tiling to further improve it [17]. Wolf and Lam [23] use unimodular loop transformations and then apply tiling to the loops which carry reuse. Li [16] present a framework where general square loop transformation matrices are used. He also illustrates how to generate the optimized code automatically. McKinley at al. [17] present a locality optimization algorithm which uses loop permutation, distribution, and fusion. None of these approaches [23, 16, 17] consider layout optimizations. Since a loop transformation may not be able to exploit locality for all the arrays referenced in the nest, it may result in sub-optimal performance.

Techniques Based on Data Transformations

These techniques focus on modifying layouts of multi-dimensional arrays rather than modifying the loop access patterns. As before, the extent of this modification is the main factor which distinguishes between different approaches. The most important category is dimension re-ordering (or dimension re-indexing). This consists of dimension-wise layout transformations (e.g., converting layout of a two-dimensional array from column-major to row-major). In this category, the layout of a multi-dimensional array is seen as a nested traversal of its dimensions in a pre-determined order. Fortunately, these types of transformations can handle a large set of access patterns found in scientific applications. The second category consists of general data transformations that can be expressed by square non-singular transformation matrices. Using these transformations, an optimizing compiler can convert, for example, a column-major layout to a diagonal layout, if doing so improves spatial locality. An important difference between this and dimension re-ordering is that the former may cause some increase in the amount of the space allocated to the array. The last category in data transformations are blocked layouts. Anderson et al. [2] show that blocked layouts can be useful in optimizing locality for distributed-shared-memory multiprocessors.

Recently researchers have investigated the use of data transformations. Kandemir et al. [12], O'Boyle and Knijnenburg [18], and Leung and J. Zahorjan [15] use general data transformations to improve spatial locality. In [12], the explicit layout representations are used whereas in [18] the primary focus is on how to generate the optimized code given a data transformation matrix. Leung and Zahorjan [15] concentrate more on minimizing extra space consumption when array layouts are modified. The main limitation of data transformations is that they cannot optimize temporal locality directly, and a layout transformation impacts all the nests that access the array in question.

Techniques Based on Combined Loop and Data Transformations

These techniques apply both loop and data transformations for optimizing locality. Both transformations may be of different extents. In principle, the most general transformations are non-singular loop and data transformations. The work on integrated loop and data transformations is relatively new. Cierniak and Li [7] use both loop and data transformations for optimizing locality. Their loop transformation matrices can contain only 1s and 0s, and their data transformations are dimension re-ordering. Kandemir et al. [11] also use dimension re-ordering for array layout transformations. However, their loop transformations use general non-singular matrices and cover a larger search space than the approach given in [7]. In [13] Kandemir et al. proposed an integrated optimization approach that uses general non-linear loop and data transformations for optimizing data locality. In this paper we use that algorithm for optimizing out-of-core computations.

3 Our Approach

In this section we present a compiler-directed approach that employs both loop and data transformations. Our loop and data transformation matrices are non-singular square matrices. To the best of our knowledge, this is the most general framework that uses *linear* transformations for optimizing locality in I/O-intensive codes. Another important characteristic of our approach is that we optimize locality globally, i.e., for several loop nests simultaneously.

Our objective is to optimize the I/O accesses of out-of-core applications through compiler analysis. Since the data sizes in these applications may exceed the size of main memory, data should be divided into chunks called *data tiles*, and the program should operate on one chunk (from each array) at a time that is brought from disk into memory. When the operation on this chunk is complete, the chunk should be stored back on disk (if it is modified). In such applications, the primary issue is to exploit the main memory–disk subsystem hierarchy rather than cache–main memory hierarchy; that is, a data tile brought into memory should be reused as much as possible. Also the number of I/O calls required to bring the said

data tile into memory should be minimized. Note that the latter problem is peculiar to I/O-intensive codes.

Notice that the necessity of working with data tiles implies that loop tiling should be used. Thus, tiling, which is an optional optimization for in-core computations, is a "must" for out-of-core programs. Given a series of loop nests that access (possibly different) subsets of out-of-core arrays declared in the program, our optimization strategy proceeds as follows:

(1) Transform the program into a sequence of independent loop nests using loop fusion, distribution, and code sinking.

(2) Build an *interference graph* and identify the *connected components*. The interference graph is a bipartite graph (V_n, V_a, E) where V_n is the set of loop nests, V_a is the set of arrays, and E is the set of edges between loop nodes and array nodes. There is an edge $e \in E$ between $v_a \in V_a$ and $v_n \in V_n$ if and only if v_n references v_a.

(3) For each connected component:

- **(3.a)** Order the loop nests according to a *cost* criterion using profile information.
- **(3.b)** Optimize the most costly nest using only data transformations and then tile this nest.
- **(3.c)** For each of the remaining nests in the connected component (according to their order):
  - **(3.c.a)** Optimize the nest using loop and data transformations taking the file layouts found so far into account and then tile the nest.
  - **(3.c.b)** Propagate the file layouts found so far to the remaining nests.

A few points need to be noted. First, our method of tiling a nest for out-of-core computations is different from the traditional tiling used to exploit cache locality. We will discuss this issue in detail later on in this paper. Second, the loop transformations found should preserve the data dependences in the program. We ensure this by using Bik and Wijshoff's completion technique [3]. Third, a data transformation in its most general form can increase the space requirements of the original array. This is because of the requirement in conventional languages that arrays have a rectilinear shape. While in in-core programs this extra space may not be an issue, in out-of-core computations, this may not be the case. Later on in the paper we also discuss how we alleviate this problem to some extent.

Figure 1 illustrates using an example how Steps (1) and (2) of the optimization strategy works. The original program on the left part of the figure consists of a sequence of two imperfectly nested loop nests. Within the loop nests are the names of the arrays accessed separated by commas. All arrays are assumed to be out-of-core and reside in files on disk(s). As a first step, the compiler transforms these loop nests to a sequence of perfectly nested loops using a combination of loop fusion, loop distribution, and code sinking [24]. In this example, we assume that this can be achieved using loop fusion for the first imperfectly nested loop nest and using loop distribution for the second. In the next step, the compiler builds an interference graph, and runs a connected component algorithm on it. The two connected components shown in the figure correspond to two program fragments which access disjoint sets of arrays. The first fragment accesses arrays U, V, and W whereas the second one accesses arrays X and Y. Since connected components do not have a common array, the rest of the approach can operate on a single connected component at a time.

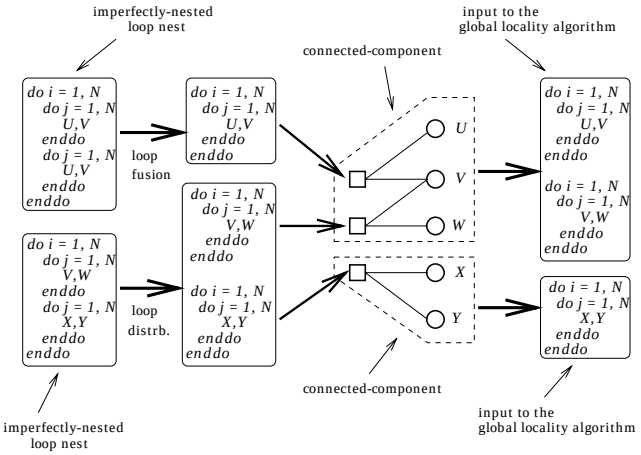

Figure 1: Example application of file locality optimization algorithm.

3.1 Motivation

In this subsection we illustrate through an example why a combined approach to locality is required. Consider the following program fragment assuming that the arrays are out-of-core and the default file layout is *column-major* for all arrays. For example, this fragment may correspond to the first connected component in Figure 1.

```
do i = 1, N
  do j = 1, N
    U(i,j) = V(j,i) + 1.0
  end do
end do

do i = 1, N
  do j = 1, N
    V(i,j) = W(j,i) + 2.0
  end do
end do
```

An approach based on linear loop transformations alone (e.g., [16], [17]) cannot optimize spatial locality for both arrays in the first nest as there are spatial reuses in orthogonal directions. The same is true for the second nest also. Therefore, two out of four references will go unoptimized. An approach based on linear data transformations alone, on the other hand, (e.g., [12], [18]) can select row-major layout for U and column-major layout for W. Since there are conflicting layout requirements for array V, one of the references will be unoptimized. The approach discussed in this paper proceeds as follows. Assuming that the first nest is costlier than the second, it first focuses on this nest and (using data transformations) selects row-major layout for U and column-major layout for V. Then it moves to the second nest. Since the layout for V has already been determined, it takes this into account and interchanges the loops so that the locality for array V will be good assuming column-major layout. This new loop order imposes array accesses along the rows of array W; consequently, our approach selects row-major layout for this array. To sum up, using a combination of loop and data transformations we are able to optimize locality for all the references in this program fragment.

3.2 Technical Details

3.2.1 Hyperplanes and array layouts

Our approach uses a simple linear algebra concept called a *hyperplane*. We focus on programs where array subscript expressions

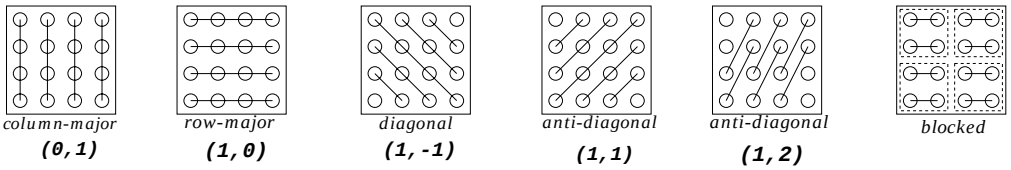

Figure 2: Example file layouts and their hyperplane vectors.

and loop bounds are affine functions of the enclosing loop indices and symbolic (loop-invariant) constants. In such a program, a reference to an m–dimensional array appearing in a k–dimensional loop nest can be represented by an access (or reference) matrix $\mathcal{L}$ of size $m \times k$ and an offset vector $\bar{o}$ of size m [23]. For example, the reference $V(j,i)$ in the first nest of the example fragment shown earlier can be represented by $\mathcal{L}\bar{I} + \bar{o}$, where

$$\mathcal{L} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \bar{I} = \begin{pmatrix} i \\ j \end{pmatrix}, \text{and } \bar{o} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}.$$

In an m-dimensional data space, a *hyperplane* can be defined as a set of tuples

$$\{(a_1, a_2, ..., a_m) \mid g_1 a_1 + g_2 a_2 + ... + g_m a_m = c\}$$

where g_1, g_2,...,g_m (at least one which is nonzero) are rational numbers called *hyperplane coefficients* and c is a rational number called hyperplane constant [21]. We use a row vector $g^T = (g_1, g_2, ..., g_n)$ to denote a hyperplane family (for different values of c).

To keep the discussion simple, we focus on two dimensional arrays; the results presented in this paper extend to higher dimensional arrays as well. In a two-dimensional data space, the hyperplanes are denoted by row vectors of the form (g_1, g_2). In that case, we can think of a hyperplane family as parallel lines for a fixed coefficient set (that is, the (g_1, g_2) vector) and different values of c. An important property of the hyperplanes is that two data points (array elements) (a, b) and (c, d) lie along the same hyperplane if $(g_1, g_2)\begin{pmatrix} a \\ b \end{pmatrix} = (g_1, g_2)\begin{pmatrix} c \\ d \end{pmatrix}$. For example, a hyperplane such as $(0, 1)$ indicates that two elements belong to the same hyperplane as long as they have the same value for the column index (i.e., the second dimension); the value for the row index does not matter.

It is important to note that a hyperplane family can be used to partially define the file layout of an out-of-core array. In the case of a two-dimensional array, the vector $(0, 1)$ is sufficient to indicate that the elements in a column of the array (i.e., the elements in a hyperplane with a specific c value) will be stored consecutively in file and will have *spatial locality*. The relative order of these columns in file is not as important provided that array size is large enough compared to the memory size which almost always holds true in out-of-core computations. In other words, the vector $(0, 1)$ can be used for representing column-major file layout. A few possible file layouts and their associated hyperplane vectors for two-dimensional arrays are shown in Figure 2. The last layout given is an example of blocked layouts where each dashed square constitutes a block. Our method currently does not handle blocked layouts, although as it is it can be used for determining optimal storage of blocks in file with respect to each other. It should be emphasized that these file layouts are only a handful of the set of all possible layouts, and there are many other hyperplanes which can define file layouts in two-dimensional space. For example, $(7, 4)$ also defines a hyperplane family and a file layout such that two array elements (a, b) and (c, d) lie along a same hyperplane (i.e., have spatial locality) if $7a + 4b = 7c + 4d$.

3.2.2 Loop transformations

Nest-level optimizations transform a loop nest to increase data locality. Essentially, the objective is to obtain either temporal locality or stride-one access of the arrays which is important. To understand the effect of a loop transformation let us represent a loop nest of depth k which consists of loops $i_1, i_2, \cdots, i_k$ as a polyhedron defined by the loop limits. We use a k-dimensional vector $\bar{I} = (\imath_1, \imath_2, \cdots, \imath_k)$ called the iteration vector to denote the execution of the body of this loop nest with $i_1 = \imath_1, i_2 = \imath_2, \cdots, i_k = \imath_k$.

Recall that we assume that the array subscript expressions and loop bounds are affine functions of enclosing loop indices and loop-index-independent variables. The class of iteration space transformations we are interested in can be represented using linear non-singular transformation matrices. For a loop nest of depth k, the iteration space transformation matrix T is of size $k \times k$. Such a transformation maps each iteration vector $\bar{I}$ of the original loop nest to an iteration $\bar{I}' = T\bar{I}$ of the transformed loop nest. Therefore, after the transformation, the new subscript function is $\mathcal{L}T^{-1}\bar{I}' + \bar{o}$. The problem investigated in papers such as [23] and [16] is to select a suitable T such that the locality of the reference is improved and all the data dependencies in the original nest are preserved. For example, in a loop nest of depth 2, loop interchange is represented by a *unimodular* transformation matrix

$$T = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}.$$

3.2.3 Combining loop and data layout transformations

The following claim gives us an important relation between a loop transformation, the file layout, and the access matrix in order for a given reference to have spatial locality in the innermost loop (see [13] for the proof).

Claim 1 *Consider a reference $\mathcal{L}\bar{I} + \bar{o}$ to a 2-dimensional array in a loop nest of depth k where $\mathcal{L} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1k} \\ a_{21} & a_{22} & \cdots & a_{2k} \end{pmatrix}$ and let $Q = \begin{pmatrix} q_{11} & q_{12} & \cdots & q_{1k} \\ q_{21} & q_{22} & \cdots & q_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ q_{k1} & q_{k2} & \cdots & q_{kk} \end{pmatrix}$ be the inverse of the loop transformation matrix. In order to have spatial locality in the innermost loop, this array should have a file layout represented by hyperplane (g_1, g_2) such that $(g_1, g_2)\mathcal{L}(q_{1k}, q_{2k}, \cdots, q_{kk})^T = 0$.* □

Since both (g_1, g_2) and $(q_{1k}, q_{2k}, \cdots, q_{kk})^T$ are unknown, this formulation is non-linear. However, if either of them is known, the other can easily be found. If we know the last column of Q,

$$(g_1, g_2) \quad \in \quad Ker\left\{\mathcal{L}(q_{1k}, q_{2k}, \cdots, q_{kk})^T\right\}. \qquad (1)$$

Similarly, if we know (g_1, g_2), then

$$(q_{1k}, q_{2k}, \cdots, q_{kk})^T \quad \in \quad Ker\left\{(g_1, g_2)\mathcal{L}\right\}. \qquad (2)$$

Usually *Ker* sets may contain multiple vectors in which case we choose the one such that the gcd of its elements is minimum. Returning to the example given at the beginning of Section 3.1, we find the access matrices for nest 1 are $\mathcal{L}_U = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$, and $\mathcal{L}_{V_1} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$; and for nest 2 are $\mathcal{L}_{V_2} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$, and $\mathcal{L}_W = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$. As mentioned earlier, for the first nest we apply only data transformations; that is, $(q_{12}, q_{22})^T = (0, 1)$ (Q is identity matrix). Using Relation (1) given above, for array U,

$$(g_1, g_2) \in Ker\left\{\mathcal{L}_U \begin{pmatrix} 0 \\ 1 \end{pmatrix}\right\} \Longrightarrow (g_1, g_2) \in Ker\left\{\begin{pmatrix} 0 \\ 1 \end{pmatrix}\right\}.$$

A particular solution is $(g_1, g_2) = (1, 0)$; i.e., array U should be stored row-major. For V,

$$(g_1, g_2) \in Ker\left\{\mathcal{L}_{V_1} \begin{pmatrix} 0 \\ 1 \end{pmatrix}\right\} \Longrightarrow (g_1, g_2) \in Ker\left\{\begin{pmatrix} 1 \\ 0 \end{pmatrix}\right\}.$$

Selecting $(g_1, g_2) = (0, 1)$ results in column-major layout for array V.

Having fixed the layouts for these two arrays, we proceed with the second nest, assuming again that Q is the inverse of the loop transformation matrix for this nest. First, using Relation (2), we find the loop transformation which satisfies the reference to array V in this nest:

$$(q_{12}, q_{22})^T \in Ker\{(0, 1)\mathcal{L}_{V_2}\} \Longrightarrow (q_{12}, q_{22})^T \in Ker\{(0, 1)\}$$

A particular solution is $(q_{12}, q_{22})^T = (1, 0)^T$, which in turn can be completed (using the approach in [3]) as $Q = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$. Notice that this matrix corresponds to loop interchange [24]. The only remaining task is to determine the optimal file layout for array W. By taking into account the last column of Q and using Relation (1) once more,

$$(g_1, g_2) \in Ker\left\{\mathcal{L}_W \begin{pmatrix} 1 \\ 0 \end{pmatrix}\right\} \Longrightarrow (g_1, g_2) \in Ker\left\{\begin{pmatrix} 0 \\ 1 \end{pmatrix}\right\}$$

which means that array W should have a row-major file layout. The resulting program is as follows.

```
do u = 1, N
  do v = 1, N
    U(u,v) = V(v,u) + 1.0
  end do
end do

do u = 1, N
  do v = 1, N
    V(v,u) = W(u,v) + 2.0
  end do
end do
```

3.3 Tiling of Out-of-Core Arrays

As mentioned earlier in the paper, tiling is mandatory for out-of-core computations. Several transformations [24, 23] may need to be performed prior to tiling to ensure its legality. In our running example no such a transformation is necessary. A traditional tiling approach can derive the following code. Here B is the *tile size* and all the loops are tiled with this tile size. The loops UT and VT are the tile loops whereas the loops u' and v' are the element loops. Notice that tile loops are placed in the outer positions in each nest.

```
do UT = 1, N, B
  do VT = 1, N, B
  < read data tiles for arrays U and V from files >
    do u' = UT, min(UT+B,N)
      do v' = VT, min(VT+B,N)
        U(u',v') = V(v',u') + 1.0
      end do
    end do
  < write data tile for array U to its file >
  end do
end do

do UT = 1, N, B
  do VT = 1, N, B
  < read data tiles for arrays V and W from files >
    do u' = UT, min(UT+B,N)
      do v' = VT, min(VT+B,N)
        V(v',u') = W(u',v') + 2.0
      end do
    end do
  < write data tile for array W to its file >
  end do
end do
```

Although such a tiling strategy allows data reuse for the data tiles in memory, its I/O performance might be unexpectedly poor. The reason for this can be seen when we consider the tile access pattern shown in Figure 3(a). In this figure each circle corresponds to an array element and the arrows connecting the circles indicates file layouts (horizontal arrows for row-major and vertical arrows for column-major). The top two arrays are U and V in the first nest and the bottom two arrays are V and W in the second nest. Assuming that we have a main memory size of 32 elements (for illustration purposes), we can allocate this memory evenly across the arrays in a nest. Traditional tiling causes the tile access pattern shown in Figure 3(a). Let us focus on array V in the first nest. In order to read a 4×4 data tile from the file we need to issue 4 I/O calls. Notice that the alternative of reading the entire array and sieving out the unwanted array elements may not be applicable in out-of-core computations. For array V, we are able to read 4 elements per I/O call. The same situation also occurs with other array accesses.

Now consider the tile access pattern shown in Figure 3(b). Focusing on array V in the first nest, we see that in order to read 16 elements from the file we need to issue only 2 I/O calls (assuming that in a single I/O call *at most* 8 elements can be read or written). Notice that in both cases (Figure 3(a) and Figure 3(b)) we are using the *same amount* of in-core memory. This small example shows that by being a bit more careful about how to read the array elements from file into memory (i.e., how to tile the loop nests) we might be able to save a number of I/O calls.

The important point is to see that we can achieve this optimized tile access pattern by *not* tiling the innermost loop. This is in contrast with the traditional tiling strategy in which the innermost loop in the nest is almost always tiled (as long as it is legal to do so). Unfortunately, tiling the innermost loop in out-of-core computations (where disk accesses are very costly) can lead to excessive number of I/O calls as this loop (after linear transformations) exhibits spatial locality. Therefore, as a rule after applying the loop and data transformations to improve locality, tiling should be applied to all but the innermost loop in the nest. The tiled program corresponding to the access pattern shown in Figure 3(b) is as follows.

```
do UT = 1, N, B
  < read data tiles for arrays U and V from files >
    do u' = UT, min(UT+B,N)
      do v' = 1, N
        U(u',v') = V(v',u') + 1.0
      end do
```

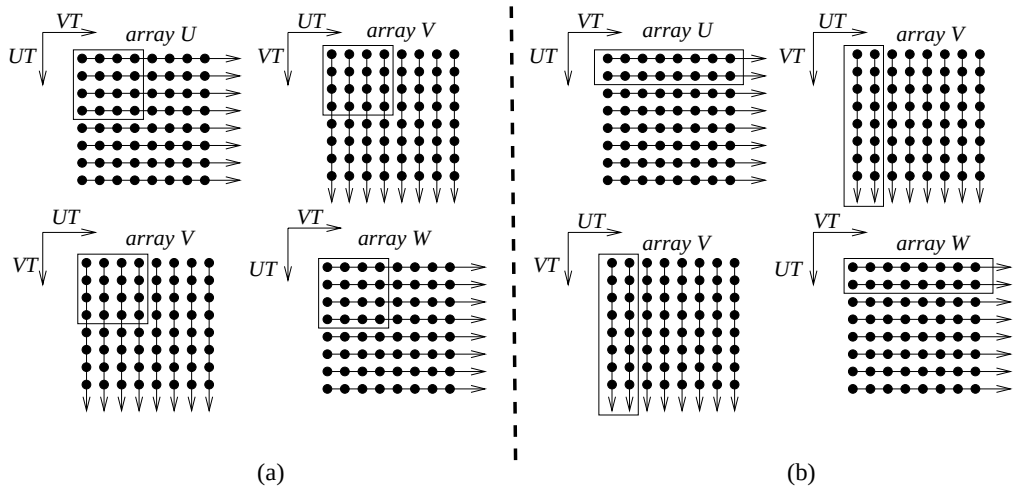

Figure 3: Different tile access patterns.

```
    end do
  < write data tile for array U to its file >
end do

do UT = 1, N, B
  < read data tiles for arrays V and W from files >
    do u' = UT, min(UT+B,N)
      do v' = 1, N
        V(v',u') = W(u',v') + 2.0
      end do
    end do
  < write data tile for array W to its file >
end do
```

3.4 Reducing the Extra Storage Requirements

When data transformations other than dimension re-indexing are applied, there might be an increase in the size of the array in question. The reason is that in conventional languages such as Fortran and C the arrays need be declared as rectilinear.

Consider the access matrix $\begin{pmatrix} a & b \\ c & 0 \end{pmatrix}$ for an array U after the locality has been optimized and the transformed loop indices are u and v from outermost. Assume that $a, b, c > 0$ and $a \geq c$. Further assume that $1 \leq u \leq N'$ and $1 \leq v \leq M'$. Thus, the transformed reference $U(au + bv, cu)$ is good from the locality point of view assuming column-major layout. However considering the bounds for u and v the array region accessed by this reference consists of $(N' + M' - 1)(a + b) \times (N' - 1)c$ elements. The transformed array should be declared as $U[a + b : aN' + bM', c : (N' - 1)c]$.

Consider now applying a data transformation represented by the transformation matrix $\begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix}$ to this access matrix. The new access matrix is $\begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} a & b \\ c & 0 \end{pmatrix} = \begin{pmatrix} a-c & b \\ c & 0 \end{pmatrix}$. Now this array needs to be declared as $U[a-c+b : (a-c)N' + bM', c : (N'-1)c]$ covering a region of $(a-c+b)(N'+M'-1) \times (N'-1)c$ elements. Depending on the actual values for a and c we may obtain a huge amount of reduction in the layout requirements. If $a < c$ then we can use the following data transformation matrix: $\begin{pmatrix} -1 & 1 \\ 0 & 1 \end{pmatrix}$.

The data transformation matrix chosen should have two important properties. First, it should not distort the good locality obtained by the previous data transformations. In our example, the 0 element in the access matrix should stay as 0. Second, it should result in a reduction in the size requirements. Although not unique nor the best, the transformation matrix given above has the desired properties. The determination of the data transformation which minimizes the space requirements is an issue that we will re-visit in the future.

4 Experimental Results

In this section we present experimental results obtained on the Intel Paragon at Caltech. Paragon uses a parallel file system called PFS which stripes files across 64 I/O nodes with 64KB stripe units. In the experiments we applied the following methodology. We took ten codes from several benchmarks and math libraries. The salient features of these codes are shown in Table 1. Then we parallelized these codes for execution on the Paragon such that the inter-processor communication is eliminated. This allowed us to focus solely on the I/O performance of the codes and the scalability of the I/O subsystem. After this parallelization and data allocation, we generated five different out-of-core versions of each code using the PASSION runtime library [22]:

- `col`: fixed column-major file layout for every out-of-core array
- `row`: fixed row-major file layout for every out-of-core array
- `l-opt`: loop-optimized version: no file layout transformations
- `d-opt`: file layout-optimized version: no loop transformations
- `c-opt`: integrated loop and file layout transformations
- `h-opt`: hand optimized version using blocking and interleaving

The `col` and `row` are the original (unoptimized) programs. For the `l-opt` version, we used the best of the resulting codes generated by [16], [17], [23]. For the `d-opt` version, we used the best of the resulting codes generated by [18], [15], and [12]. `c-opt` (compiler optimized) version is the one obtained using the approach discussed in this paper. In obtaining `h-opt` we used chunking and interleaving in order to further reduce the number of I/O calls. For all the versions except `c-opt` all the loops carrying some form of reuse are tiled. For the `c-opt` version we used the tiling strategy explained in Section 3.3.

For each code we set the memory size allocated for the computation to $1/128$th of the sum of the sizes of the out-of-core arrays

Table 1: Programs used in our experiments. The `iter` column for each code shows the number of iterations of the outermost timing loop. The `arrays` gives the number and the dimensionality of the arrays accessed by the code.

program	source	iter	arrays
mat	-	2	three 2-D
mxm	Spec92	3	three 2-D
adi	Livermore	5	three 1-D, three 3-D
vpenta	Spec92	3	seven 2-D, two 3-D
btrix	Spec92	2	twenty-five 1-D, four 4-D
emit	Spec92	2	ten 1-D, three 3-D
syr2k	BLAS	2	three 2-D
htribk	Eispack	3	five 2-D
gfunp	Hompack	3	one 1-D, five 2-D
trans	Nwchem	3	two 2-D

Table 2: Experimental results on 16 nodes.

program	col	row	l-opt	d-opt	c-opt	h-opt
mat	257.20	93.3	65.1	56.8	60.8	54.3
mxm	220.01	181.5	100.0	112.6	79.8	67.0
adi	144.12	134.9	22.8	46.5	22.8	22.8
vpenta	135.00	47.1	100.0	47.1	47.1	29.9
btrix	91.45	66.6	100.0	61.3	61.3	42.3
emit	88.64	176.5	100.0	100.0	100.0	100.0
syr2k	215.34	86.3	52.0	77.4	52.0	47.6
htribk	248.61	110.8	127.2	81.1	81.1	72.6
gfunp	86.05	128.4	73.3	68.0	46.9	34.0
trans	181.90	100.0	100.0	48.2	48.2	48.2
average:		112.5	84.0	69.9	60.0	51.9

accessed in the code. Each dimension of each array used in the computation is set to $4,096$ double precision elements. However, some array dimensions with very small hard-coded dimension sizes were not modified as modifying them correctly would necessitate full understanding of the program in question.

Table 2 shows the results on 16 processors. For each data set, the `col` column gives the total execution time in *seconds*. The other columns, on the other hand, give the respective execution times as a fraction of that of `col`. As an example, the execution time of `c-opt` version of `gfunp.4` is 46.9 percent of that of `col`. From these results we infer the following. First, the classical locality optimization schemes based on loop transformations alone may not work well for out-of-core computations. On average `l-opt` brings only a 16% improvement over `col`. The approaches based on data transformations perform much better. Our integrated approach explained in this paper, however, results in a 40% reduction in the execution times with respect to `col`. Using a hand optimized version (`h-opt`) brings an additional 8% reduction over `c-opt`, which encourages us to incorporate array chunking and interleaving into our technique.

Table 3, on the other hand, shows the speedups obtained by different versions for processor sizes of 16, 32, 64, and 128 using all 64 I/O nodes. It should be stressed that in obtaining these speedups we used the single node result of the respective versions. For example, the speedup for the `c-opt` version of `emit.3` was computed for $p \in 16, 32, 64, 128$ as

$$\frac{\textit{Execution Time of the } \texttt{c-opt} \textit{ version of } \texttt{emit.3} \textit{ on } 1 \textit{ node}}{\textit{Execution Time of the } \texttt{c-opt} \textit{ version of } \texttt{emit.3} \textit{ on } p \textit{ nodes}}$$

Since the execution times of the parallelized codes on single nodes may not be as good as the best sequential version, these results are higher than we expected. Also, since the codes were parallelized such that there is no interprocessor communication, the scalability was limited only by the number of I/O nodes and the I/O subsystem bandwidth.

5 Conclusions

The increasing disparity between the speeds of disk subsystems and the speeds of other components (such as processors, memories, and interconnection networks) has rendered the problem of improving the performance of out-of-core programs (i.e., programs that access very large amounts of disk-resident data) very important and difficult. Programmers usually have to embed code for staging in and out of data between memory and I/O devices explicitly in the program. Often this results in non-portable and error-prone code. This paper presents a technique that an optimizing compiler can use to transform the in-core programs to derive I/O-efficient out-of-core versions. In doing this, the approach uses loop (iteration space) and file layout (data space) transformations. Specifically, this paper uses linear algebra techniques to derive good file layouts along with the accompanying loop transformations. Preliminary results show that our technique substantially reduces the time spent in performing I/O. Currently we are working on extending our approach across procedure boundaries. We are also working on the problem of determining optimal file layouts using techniques from integer linear programming.

Acknowledgments Mahmut Kandemir and Alok Choudhary are supported in part by NSF Young Investigator Award CCR-9357840 and NSF CCR-9509143. J. Ramanujam is supported in part by an NSF Young Investigator Award CCR-9457768.

References

[1] W. Abu-Sufah, D. Kuck, and D. Lawrie. On the performance enhancement of paging systems through program analysis & transformations, *IEEE Trans. Comp.*, C-30(5):341–355, 1981.

[2] J. Anderson, S. Amarasinghe, and M. Lam. Data and computation transformations for multiprocessors. In *Proc. 5th ACM SIGPLAN Symp. Prin. & Prac. Par. Prog.*, July 1995.

[3] A. Bik, and H. Wijshoff. On a completion method for unimodular matrices. Technical Report 94–14, Dept. of Computer Science, Leiden University, 1994.

[4] R. Bordawekar, A. Choudhary, K. Kennedy, C. Koelbel, and M. Paleczny. A model and compilation strategy for out-of-core data-parallel programs. In *Proc. SIGPLAN Symp. Prin. & Prac. Par. Pro.*, July 1995.

[5] R. Bordawekar, A. Choudhary, and J. Ramanujam. Automatic optimization of communication in out-of-core stencil codes. In *Proc. 10th ACM Int. Conf. Supercomp.*, pp. 366–373, 1996.

[6] P. Brezany, T. Muck, and E. Schikuta. Language, compiler and parallel database support for I/O intensive applications, In *Proc. High Performance Computing & Networking*, 1995.

[7] M. Cierniak, and W. Li. Unifying data and control transformations for distributed shared memory machines. Technical Report 542, CS Dept., University of Rochester, November 1994.

[8] P. Corbett, D. Feitelson, S. Fineberg, Y. Hsu, B. Nitzberg, J. Prost, M. Snir, B. Traversat, and P. Wong. Overview of the MPI-IO parallel I/O interface, *Proc. 3rd Workshop I/O in Par. & Dist. Sys.*, Apr. 1995.

Table 3: Results on scalability of different versions.

program	version	number of processors			
		16	32	64	128
mat.2	col	10.9	20.6	34.8	64.3
	row	11.0	20.9	35.6	66.0
	l-opt	13.9	27.6	53.8	100.4
	d-opt	14.5	28.1	55.0	104.2
	c-opt	14.0	27.7	54.8	102.7
	h-opt	15.2	30.9	60.9	115.6
mxm.2	col	11.1	21.2	37.6	70.0
	row	8.2	15.4	30.0	52.6
	l-opt	11.1	21.2	37.6	70.0
	d-opt	9.7	17.0	32.1	56.4
	c-opt	13.7	24.8	56.4	106.6
	h-opt	13.7	24.8	56.1	107.2
adi.2	col	12.0	22.2	51.2	70.9
	row	6.89	10.9	18.6	31.4
	l-opt	15.3	28.2	61.4	107.5
	d-opt	13.8	24.0	55.5	74.9
	c-opt	15.3	28.2	61.4	107.5
	h-opt	15.3	28.2	61.4	107.5
vpenta.6	col	10.0	24.2	51.3	78.9
	row	14.5	28.0	60.9	109.8
	l-opt	10.0	24.2	51.3	78.9
	d-opt	14.5	28.0	60.9	109.8
	c-opt	14.5	28.0	60.9	109.8
	h-opt	14.7	29.0	62.4	108.2
btrix.4	col	10.0	18.1	27.0	42.7
	row	12.9	23.9	45.8	87.1
	l-opt	10.0	18.1	27.0	42.7
	d-opt	13.9	25.1	46.2	98.1
	c-opt	13.9	25.1	46.2	98.1
	h-opt	13.1	24.6	44.3	93.1

program	version	number of processors			
		16	32	64	128
emit.3	col	12.7	23.1	45.0	89.9
	row	6.8	11.0	18.5	33.9
	l-opt	12.7	23.1	45.0	89.9
	d-opt	12.7	23.1	45.0	89.9
	c-opt	12.7	23.1	45.0	89.9
	h-opt	12.7	32.1	45.0	89.9
syr2k.2	col	10.3	20.0	36.5	71.5
	row	11.7	22.0	38.9	78.0
	l-opt	13.8	26.8	51.0	95.1
	d-opt	12.5	24.1	45.6	87.4
	c-opt	13.8	26.8	51.0	95.1
	h-opt	14.1	26.0	51.0	95.3
htribk.2	col	11.7	20.3	37.7	76.6
	row	9.5	16.9	30.0	55.4
	l-opt	8.8	15.0	24.3	44.0
	d-opt	11.9	21.5	37.9	76.9
	c-opt	11.9	21.5	37.9	76.9
	h-opt	12.1	21.6	40.1	76.9
gfunp.4	col	10.9	20.4	38.4	70.8
	row	9.5	17.0	32.6	60.6
	l-opt	8.1	15.7	28.2	52.2
	d-opt	14.0	25.0	56.0	102.3
	c-opt	14.0	25.0	56.0	102.3
	h-opt	14.5	24.7	57.0	105.7
trans.2	col	13.0	22.7	31.6	67.7
	row	13.0	22.7	31.6	67.7
	l-opt	13.0	22.7	31.6	67.7
	d-opt	15.4	30.9	60.2	113.0
	c-opt	15.4	30.9	60.2	113.0
	h-opt	15.4	30.9	60.2	113.0

[9] T. H. Cormen, and A. Colvin. ViC*: A preprocessor for virtual-memory C*. Dartmouth College Computer Science Technical Report PCS-TR94-243, November 1994.

[10] M. Kandemir, R. Bordawekar, and A. Choudhary. Data access reorganizations in compiling out-of-core data parallel programs on distributed memory machines. In *Proc. IPPS 97*, pp. 559–564, April 1997.

[11] M. Kandemir, J. Ramanujam, and A. Choudhary. A compiler algorithm for optimizing locality in loop nests. In Proc. *11th ACM Int. Conf. Supercomp.*, pp. 269-278, July 1997.

[12] M. Kandemir, A. Choudhary, N. Shenoy, P. Banerjee, and J. Ramanujam. A hyperplane based approach for optimizing spatial locality in loop nests. In *Proc. 1998 ACM Int. Conf. Supercomp.*, July 1998.

[13] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. A matrix-based approach to the global locality optimization problem. In *Proc. PACT'98*, October 1998.

[14] M. Kandemir, M. Kandaswamy, and A.Choudhary. Global I/O optimizations for out-of-core computations. In *Proc. High-Performance Computing Conference (HiPC)*, 1997.

[15] S. Leung, and J. Zahorjan. Optimizing data locality by array restructuring. Technical Report, CSE Dept., University of Washington, TR 95-09-01, Sep. 1995.

[16] W. Li. Compiling for NUMA parallel machines. Ph.D. dissertation, Cornell University, 1993.

[17] K. McKinley, S. Carr, and C.W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 1996.

[18] M. O'Boyle, and P. Knijnenburg. Non-singular data transformations: Definition, validity, applications. In *Proc. 6th Workshop on Compilers for Par. Comp.*, pp. 287–297, 1996.

[19] M. Paleczny, K. Kennedy, and C. Koelbel. Compiler support for out-of-core arrays on parallel machines. CRPC Technical Report 94509-S, Rice University, Dec. 1994.

[20] J. Ramanujam. Non-unimodular transformations of nested loops. In *Proc. Supercomputing 92*, pp. 214–223, 1992.

[21] J. Ramanujam, and P. Sadayappan. Compile-time techniques for data distribution in distributed memory machines. *IEEE Trans. Par. & Dist. Sys.*, 2(4):472–482, Oct. 1991.

[22] R. Thakur, A. Choudhary, R. Bordawekar, S. More, and S. Kuditipudi. PASSION: Optimized I/O for parallel applications, *IEEE Computer*, (29)6:70–78, June 1996.

[23] M. Wolf, and M. Lam. A data locality optimizing algorithm. In *Proc. ACM SIGPLAN 91 Conf. Prog. Lang. Des. & Impl.*, pp. 30–44, June 1991.

[24] M. Wolfe. *High Performance Compilers for Parallel Computing*, Addison-Wesley, 1996.