Improving the Performance of Out-of-Core Computations*

M. Kandemir† J. Ramanujam‡ A. Choudhary§

Abstract

The difficulty of handling out-of-core data limits the potential of parallel machines and high-end supercomputers. Since writing an efficient out-of-core version of a program is a difficult task and since virtual memory systems do not perform well on scientific computations, we believe that there is a clear need for compiler-directed explicit I/O approach for out-of-core computations. In this paper, we present a compiler algorithm to optimize locality of disk accesses in out-of-core codes by choosing a good combination of file layouts on disks and loop transformations. The transformations change the access order of array data. Experimental results obtained on IBM SP-2 and Intel Paragon provide encouraging evidence that our approach is successful at optimizing programs which depend on disk-resident data in distributed-memory machines.

1 Introduction

It is now widely acknowledged that due to increasing gap between speeds of high performance processors and memory subsystems, compiler optimizations aimed at improving the locality characteristics of programs are extremely important. Since the overall system performance is limited by the speed of the slowest unit, I/O subsystems of supercomputers constitute a bottleneck for applications that access huge amounts of data. A computation which operates on disk-resident arrays are called *out-of-core*, and an optimizing compiler for out-of-core computations is termed as *out-of-core compiler*. In contrast a computation which operates on data sets in memory is called *in-core*. Traditionally, in scientific computations, I/O is handled in two different ways: virtual memory (VM) and explicit file I/O. Although VM ensures correctness for programs whose data sizes far exceed the size of available memory, it has been observed that the performance of scientific applications that rely on VM is generally poor due to frequent paging in and out of data [5]. Our experiments confirm these results as for example shown in Figure 1 for the nest given in Figure 4:A. Figure 1 shows the performance improvement obtained by explicit I/O against VM on the Intel Paragon. Since the Intel Paragon has 32 MBytes node memory, three arrays each of which contains $1K \times 1K$ elements of size 8 bytes represent the largest amount of data per node in our input set that can fit into memory. (Note that all sizes of data sets in this paper are given on a per-node basis). So, as expected, for cases with 512×512 and $1K \times 1K$ arrays, VM performs better. But starting from $2K \times 2K$ arrays explicit I/O outperforms the VM and the performance improvement is highly significant. In the optimized approach, even for 512×512 and $1K \times 1K$ arrays, explicit I/O has been performed. This and similar experiments convince us that the VM is not a viable alternative for out-of-core computations.

Explicit file I/O, however, has it own problems. The task of programming I/O is tedious and error-prone. The low-level I/O decisions made by programmers also severely affect the portability of applications [11]. We show in this paper that a compiler approach based on explicit I/O can be very successful provided careful attention is paid to the layout of data in files. Note that the term layout is used to describe layout of data in files, e.g., column-major or row-major storage, and *does not* denote partitioning or decomposition of data among processor memories.

Since, due to high I/O startup costs, accessing data on disk is usually orders of magnitude slower than accessing data in memory, it is essential to reduce the number as well as the volume of the disk accesses. In this paper, we make the following contributions. First, we present a compiler algorithm based on explicit file I/O to reduce the time spent in disk I/O on distributed-memory message-passing machines. Our algorithm automatically transforms a given loop nest to exploit locality on disks, assigns appropriate disk layouts for out-of-core arrays, and partitions the available node memory across the out-of-core arrays, all in a unified framework. Secondly, we present performance results for several kernels on an IBM SP-2 and on an Intel Paragon. These results provide enough evidence that our algorithm can be very useful for compilation of out-of-core codes in distributed-memory machines. Thirdly, we discuss the impact of the algorithm on multiprogramming, and comment on the interaction between our algorithm and cache optimization techniques [8].

2 Compiling Out-of-Core Codes

In order to translate out-of-core programs, the compiler has to take into account the data distribution on disks, the number of disks used for storing data etc. The compilation of an out-of-core loop nest consists of two phases. In the first phase, called *in-core phase*, the out-of-core arrays in the source HPF program are partitioned according to the distribution information and bounds for local out-of-core arrays are computed. The second phase, called *out-of-core phase*, involves adding appropriate statements to perform I/O and communication. The local out-of-core arrays are first tiled according to the node memory available in each processor and the resulting tiles are analyzed for communication. The loops are then modified to insert necessary I/O calls. We identify three major issues to be exploited in order to generate efficient code for out-of-core computations: access pattern, storage layouts on disk, and memory allocation. The access pattern is generally a function of distribution directives and control constructs such as loops, conditional statements etc. Since in scientific computations most of the execution time is spent in loop nests, we will consider loops as the sole factor determining the access pattern along with the data distribution directives. On the other hand, the storage layout for an h-dimensional array can be in one of the $h!$ forms, each of which corresponds to layout of data on disk linearly by a nested traversal of the axes in

*Supported by NSF Young Investigator Award CCR-9357840 and by CCR-9509143. J. Ramanujam is supported in part by an NSF Young Investigator Award CCR-9457768 by CCR-9210422.

†CIS Dept., Syracuse University, Syracuse, NY 13244.

‡ECE Dept., Louisiana State University, Baton Rouge, LA 70803.

§Corresponding Author, ECE Dept., Northwestern University, Evanston, IL 60208-3118, e-mail: choudhar@ece.nwu.edu

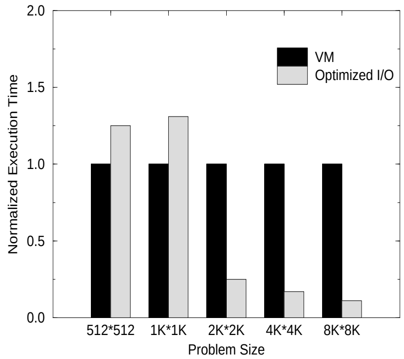

Figure 1: Optimized I/O vs. Virtual Memory (VM) on Intel Paragon.

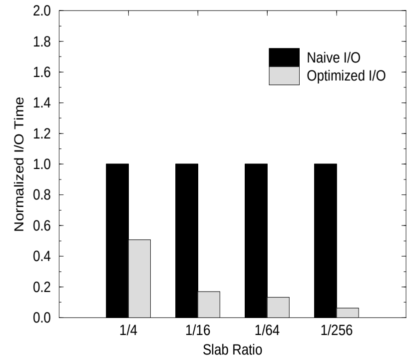

Figure 2: Optimized I/O vs. Unoptimized (naive) I/O on IBM SP-2.

some predetermined order. The innermost axis is called *the fastest changing dimension*; for example, for row-major storage layout of a two-dimensional array the second axis is the fastest changing dimension. And lastly, since node memory is a limited resource, it should be divided optimally among competing out-of-core arrays such that the total I/O time is minimized. A compiler for out-of-core codes should optimize the access pattern, storage layout and memory allocation together in order to exploit the locality as much as possible. In this paper, we offer an automatic method by which an optimizing compiler can achieve this goal.

Tiling (also known as blocking) is a technique to improve the locality and is a combination of strip-mining and loop permutation [17, 18]. It can be used to automatically create blocked versions of programs, and when it is applied it replaces the original loop with two new loops: a tiling loop and an element loop. The compilation methodology for in-core data-parallel programs can be naively extended for out-of-core computations as follows: after the node program is determined, the loops are tiled and appropriate I/O calls are inserted between tiling loops. If no optimization is performed, *data tiles* with sides of equal length in each dimension are fetched from disk, and the available memory is decomposed as evenly as possible across the arrays involved. We believe that the code produced by this naive method may not perform well due to the following reasons.

1. In this method, the order of tiling loops is the same as that of the original loops. As will be shown later, for many nests encountered in practice that order of tiling loops may not be the best to exploit the data locality on disks.

2. Assuming a fixed disk layout such as row-major or column-major may not be a good idea. In out-of-core computations, fixed layout strategy can lead to poor results on secondary storage. Since most of the known approaches like [7] and [17] are based on the fixed layout strategy, we believe their effect on optimizing I/O is limited.

3. When more than one array is involved in computation, during memory allocation it might be better to favor (by allocating more memory) the frequently accessed arrays over the others.

Our solution involves deriving the best loop transformation and disk layouts, and allocating the available node memory optimally.

2.1 Motivation for Optimized I/O

In order to illustrate the performance improvement that can be obtained by compiler optimizations over the naive explicit I/O approach described before, we consider the nest shown in Figure 4:A, assuming that arrays A, B and C are $n \times n$ out-of-core arrays. In this example, using HPF-like distribution directives, we assume that array A is distributed in row-block, the array B is distributed in column-block across the processors, and the array C is replicated. Also, with out-of-core computations, the compiler directives apply to data on disks. In the naive translation, after obtaining the node program, the compiler tiles all four loops and inserts I/O statements between tiling loops. A sketch of the resulting code is given in Figure 4:B, assuming n is an exact multiple of S, the tile size; and p is the number of processors. In the translated code the loops u, v, w and y are called *tiling loops* and the computation inside the tiling loops is performed on data tiles (sub-matrices) rather than individual array elements. In other words, there are four more loops called *element loops* (not shown for sake of clarity) that iterate over the individual elements of data tiles of A, B and C. It should be emphasized that a reference such as $A[u, v]$ denotes a data tile of size $S \times S$ from file coordinates (u, v) as upper-left corner to $(u + S - 1, v + S - 1)$ as lower-right corner. A reference like $A[u, 1 : n]$, on the other hand, denotes a data tile of size $S \times n$ from $(u, 1)$ to $(u + S - 1, n)$, i.e. a block of S consecutive rows of the out-of-core matrix A.

With Figure 4:B, during the execution, square tiles of size $S \times S$ (shown as shaded blocks in Figure 4:D) are read from disks. Note that this tile allocation scheme implies the memory constraint $3S^2 \leq M$ where M is the size of the node memory. However, by applying our approach which is described in the following sections, the compiler generates the code in Figure 4:C directly from the code in Figure 4:A. It decides on a row-major disk layout for A and C and a column-major disk layout with B; the optimized code shown in Figure 4:C is obtained after appropriate transformation of the tiling loops. The tiles of size $S \times n$ are allocated for A and C, and a tile of size $n \times S$ is allocated for B as shown in Figure 4:G, resulting in the memory constraint $3nS \leq M$. Since, for some dimensions, the tile size is equal to n (the array size), the tiling loops w and y disappear.

We ran the naive and optimized versions of this example nest on a single node of IBM SP-2 with $4K \times 4K$ double arrays, varying the amount of available node memory. Figure 2 shows the I/O times normalized with respect to the naive (unoptimized) version. It should be noted that the performance improvement is significant especially with the small values of *slab ratio* (SR), the ratio of

Step 1 Initialize $i = 1$. Set data tile size for each dimension of each array to S.

Step 2 Set $\vec{\ell_i}^{C}.Q = (0, 0, ..., 0, 1)$ and $\vec{\ell_k}^{C}.Q = (\delta, \delta, ..., \delta, 0)$ for each $k \neq i$, where δ denotes *dont-care.*

Step 3 Set the disk layout for C such that i^{th} index position will be the fastest changing position.

Step 4 For each array reference A on the RHS that has $\vec{\ell_l}^{A} = \vec{\ell_i}^{C}$ for some l, try to set the disk layout for A such that the l^{th} dimension will be the fastest changing dimension.

Step 5 Choose an array reference A for which the equality in **Step 4** does not hold. Initialize $j = 1$.

Step 6 Set $\vec{\ell_j}^{A}.Q = (0, 0, ..., 1, 0)$ and $\vec{\ell_k}^{A}.Q = (\delta, \delta, ..., \delta, 0, 0)$ for each $k \neq j$. If this step is consistent with the previous steps go to **Step 7**, otherwise increment j and go to the beginning of this step. If there exist inconsistencies for all j values, then initialize $j = 1$, and set $\vec{\ell_j}^{A}.Q = (0, 0, ..., 1, 0, 0)$ and $\vec{\ell_k}^{A}.Q = (\delta, \delta, ..., \delta, 0, 0, 0)$ for each $k \neq j$, and repeat **Step 6** and so on. If no T^{-1} is found then fill the remaining entries arbitrarily observing the *dependences* and *non-singularity*.

Step 7 Repeat **Step 6** for all reference matrices of a particular A (Of course, all reference matrices for a particular A should have the same distribution).

Step 8 Repeat **Step 6** for all distinct array references.

Step 9 Record the obtained transformation matrix. Also record, for each array, the loop index position which appears in the fastest changing position for that array.

Step 10 Increment i and go to **Step 2** (try a different layout for the LHS array C).

Step 11 Compare all the recorded transformation matrices and their associated layouts, and choose the best alternative (see the explanation in Section 4).

Step 12 Divide the arrays into groups according to the selected disk layouts of the associated files. For each array, if a loop index appears in the fastest changing array dimension, and does not appear in any other dimension of any reference in that group, increase the tile size in that dimension to full array size for that reference.

Step 13 Re-evaluate the memory constraint to set the value of S.

Figure 3: Algorithm for optimizing locality in out-of-core computations.

available node memory size to total size of out-of-core arrays. Note that a smaller slab ratio indicates less available node memory. We believe that the Figures 1 and 2 indicate the need for compiler-directed optimized explicit I/O approach for out-of-core computations.

After briefly reviewing the loop transformation theory in Section 3, in Section 4 we present our optimization algorithm which contains the steps necessary to obtain the I/O optimized node programs from global name space programs. In Section 5, we present our experimental results. In Section 6, we show how our technique can be extended to handle the global I/O optimization problem and finally we conclude the paper in Section 7.

3 Review of Loop Transformations

We focus on loops where both array subscripts and loop bounds are affine functions of enclosing loop indices. A reference to an array X is represented by $X(\mathcal{L}\vec{I} + \vec{b})$ where $\mathcal{L}$ is a linear transformation matrix called *array reference matrix*, $\vec{b}$ is offset vector and $\vec{I}$ is a column vector representing the loop indices i_1, i_2,...,i_n starting from the outermost loop. Linear mappings between iteration spaces of loop nests can be modeled by nonsingular matrices [7]. Since the transformation matrices for general array references obtained by our approach are not necessarily unimodular, we need a general non-singular transformation scheme rather than a simpler unimodular transformation framework. If $\vec{I}$ is the original iteration vector, after applying linear transformation T, the new iteration vector is $\vec{J} = T\vec{I}$. Similarly if $\vec{d}$ is the distance/direction vector, on applying T, $T\vec{d}$ is the new distance/direction vector. A transformation is *legal* if and only if $T\vec{d}$ is lexicographically positive for every $\vec{d}$ [18]. On the other hand, since $\mathcal{L}\vec{I} = \mathcal{L}T^{-1}\vec{J}$, it is clear that $\mathcal{L}T^{-1}$ is the new array reference matrix after the transformation. We denote T^{-1} by Q. An important characteristic of our approach is that using the array reference matrices, the entries of Q are derived systematically. For the rest of the paper, the reference matrix for array X will be denoted by $\mathcal{L}^X$ whereas the i^{th} row of the reference matrix for array X will be denoted by $\vec{\ell_i}^{X}$. Unless otherwise stated, the word *loop* in this paper refers to *tiling loop*; and the element loops and communication statements will not be shown for sake of clarity.

4 Disk Locality Algorithm

Let i_1,i_2,...,i_n be loop indices of the original nest, and j_1,j_2,...,j_n be the loop indices of the transformed nest, starting from outermost position. An explanation of our algorithm for a single statement follows. (The modifications necessary for multiple LHSs and multiple nests are discussed in Section 6)

- Our transformation matrix should be such that the LHS array of the transformed loop should have the innermost index as the only element in one of the array dimensions and that in-

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

(A)

```
DO u = 1, n/p, S
  DO v = 1, n, S
    read data tile for A[u,v]
      DO w = 1, n, S
        read data tile for B[w,u]
        DO y = 1, n, S
          read data tile for C[y,w]
          A[u,v]+=B[w,u]+C[y,w]
        ENDDO y
      ENDDO w
      write data tile for A[u,v]
  ENDDO v
 ENDDO u
```

(B)

```
DO u = 1, n/p, S
 read data tile for A[u,1:n]
 read data tile for B[1:n,u]
 DO v = 1, n, S
   read data tile for C[v,1:n]
   A[u,1:n]+=B[1:n,u]+C[v,1:n]
 ENDDO v
 write data tile for A[u,1:n]
ENDDO u
```

(C)

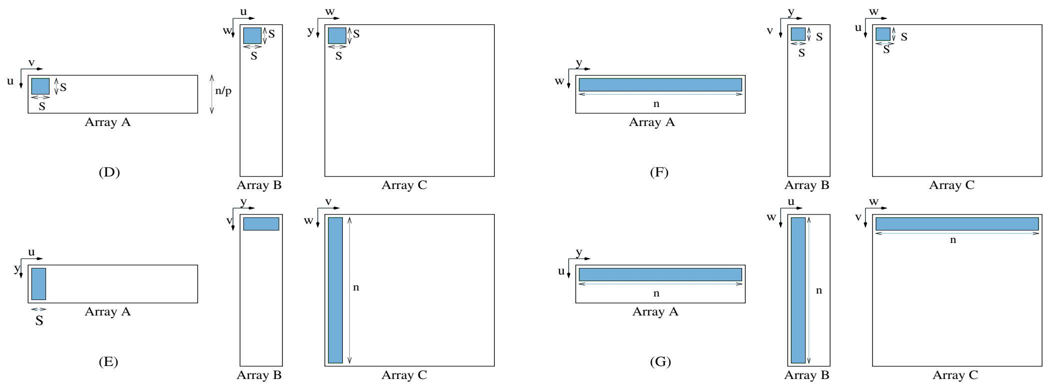

Figure 4: (A) An out-of-core loop nest. (B) Straightforward translation. (C) I/O optimized translation. (D)-(G) Different Tile Allocations.

dex should not appear in any other dimension for this array. In other words, after the transformation, the LHS array C should be of the form $C(*, *, ..., j_n, ..., *, *)$ where j_n (the new innermost loop index) is in the r^{th} dimension and $*$ indicates a term independent of j_n. This means that the r^{th} row of the transformed reference matrix for C is $(0, 0, ..., 0, 1)$ and all entries of the last column, except the one in r^{th} row, are zero. After that process the LHS array can be stored on disk such that the r^{th} dimension will be the fastest changing dimension. This exploits the spatial locality on disk, for this reference.

- Then the algorithm works on one reference from the RHS at a time. If a row s in the data reference matrix is identical to r^{th} row of the original reference matrix of the LHS array, then this RHS array is considered to be stored on disk such that the s^{th} dimension will be the fastest changing dimension.[1]

- If the condition above does not hold for a RHS array A, in that case the algorithm attempts to transform the reference to $A(*, *, ..., \mathcal{F}(j_{n-1}), ..., *, *)$ where $\mathcal{F}(j_{n-1})$ is an affine function of j_{n-1} and other indices except j_n, and $*$ indicates a term independent of both j_{n-1} and j_n. This helps to exploit the spatial locality at the second innermost loop. If no such transformation is possible, the j_{n-2} is tried and so on. If all loop indices are tried unsuccessfully, then the remaining entries of Q are determined considering the data dependences and non-singularity. A modified version of the completion algorithms presented by Li [7] can be used for this purpose.

[1] Note that having such a row s does not guarantee that the array will be stored on the disk such that the s^{th} dimension will be the fastest changing dimension.

- After a transformation and corresponding disk layouts are found, the next alternative layout for the LHS is considered and so on. Among all feasible solutions, the best one is chosen. Although several approaches can be taken to select the best alternative, we found the following scheme both accurate and practical: Each loop in the nest is numbered with its level (depth), the outermost loop getting the number 1. Then, for each reference in the nest, the level number of the loop whose index sits in the fastest changing dimension for this reference is recorded. The number for all references in the nest are summed up, and the alternative with the maximum sum is chosen. As an example, if, for a two-deep nest with three references, an alternative exploits the locality for the first reference in the outer loop, and for the other references in the inner loop, the sum for this alternative is $1+2+2 = 5$.

- After choosing the best alternative, the following memory allocation scheme is applied: First, data tile size for each dimension of each array is set to S, a parameter whose value depends on the size of the available memory. For example, if a loop nest contains a one-dimensional array, two two-dimensional arrays and a three-dimensional array, the algorithm first allocates a tile of size S for the one-dimensional array, a tile of size $S \times S$ for each of the two-dimensional arrays, and a tile of size $S \times S \times S$ for the three-dimensional array. This allocation scheme implies the memory constraint $S^3 + 2S^2 + S \leq M$ where M is the size of the node memory. After that, the arrays are divided into groups according to disk layouts of the associated files (i.e. the arrays with the same disk layout are placed in the same group). The algorithm then handles the groups one by one. If a loop index appears in the fastest changing array dimension of the group, and does not appear in any other dimension of any reference in that group, the compiler increases the tile size in that dimension to full (local) array size for that reference. The memory constraint should be adjusted accordingly. Note that any inconsistency between the groups (due to a common loop index) should be resolved by not changing the original (initial) tile sizes.

The following points should be noted. First, the transformation matrix obtained by the above approach should be non-singular and must preserve the data dependence relations. Second, our algorithm considers all possible storage layouts, of which the row-major and column-major layouts are only two alternatives. Third, it should be noted that the algorithm first optimizes the LHS array as much as possible. This is important because of the fact that the data tiles for this array are both read and written.

The algorithm is shown in Figure 3. In the algorithm, C is the array reference on the LHS whereas A represents an array reference from RHS. The symbol $\times$ denotes the *don't care* condition.

4.1 Constraints

During the compilation of an out-of-core program either or both the following may be true: (a) the compiler does not have complete knowledge about the access pattern or storage layouts; (b) the compiler, due to data dependences or other constraints, is not able to change the access pattern or storage layout. Each unknown or unmodifiable information about access pattern or storage layouts constitutes a constraint for the compiler. The constraints mentioned can originate from different factors. For example, data dependence relations may render all but one (tiling) loop order illegal. The practical significance of this is that the compiler cannot change the loop order, but it can customize the disk layouts to optimize the locality. In another case, the storage layout of a specific array might not be set to a desired form, because the array has already been created on disk and changing the layout on disk would be too expensive.

We now focus on the problem of optimizing locality when some or all array layouts are fixed, as this case frequently occurs in practice. We note that each fixed layout requires that the innermost loop index should be in the appropriate array index position (dimension), depending on disk layout of the array. For example, suppose that the disk layout for a h-dimensional array is such that the dimension k_1 is the fastest changing dimension, the dimension k_2 is the second fastest changing dimension, k_3 is the third etc. The algorithm should first try to place the new innermost loop index j_n only to the ${k_1}^{th}$ dimension of this array. If this is not possible, then it should try to place j_n only to the ${k_2}^{th}$ dimension and so on. If all dimensions up to and including k_h are tried unsuccessfully, then j_{n-1} should be tried for the ${k_1}^{th}$ dimension and so on. As we will show later, this modified algorithm with the constrained layouts is very important for global I/O optimization.

4.2 Example

This section presents an example to show the working of the algorithm. Due to space concerns, we do not show the steps or the parts of the steps which lead to unsuccessful trials.

Consider the example shown in Figure 4:A. Figure 4:B presents a naive out-of-core translation for it. The tile allocations for this naive version are illustrated in Figure 4:D. The array reference matrices are as follows:

$L^A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$, $L^B = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$ and $L^C = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$. The algorithm works as follows:

$L^A.Q = \begin{bmatrix} 0 & 0 & 0 & 1 \\ \delta & \delta & \delta & 0 \end{bmatrix}$. Therefore $q_{11} = q_{12} = q_{13} = q_{24} = 0$ and $q_{14} = 1$.

$L^B.Q = \begin{bmatrix} \delta & \delta & \delta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$. Therefore $q_{34} = 0$.

$L^C.Q = \begin{bmatrix} \delta & \delta & 1 & 0 \\ \delta & \delta & 0 & 0 \end{bmatrix}$. Therefore $q_{33} = q_{44} = 0$ and $q_{43} = 1$. At this point $T^{-1} = Q = \begin{bmatrix} 0 & 0 & 0 & 1 \\ q_{21} & q_{22} & q_{23} & 0 \\ q_{31} & q_{32} & 0 & 0 \\ q_{41} & q_{42} & 1 & 0 \end{bmatrix}$. We set the unknowns to the following values: $q_{22} = q_{23} = q_{31} = q_{41} = q_{42} = 0$ and $q_{21} = q_{32} = 1$, and obtain $T^{-1} = Q = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$.

The resulting code is as follows.

```
DO u = 1, n/p, S
 DO v = 1, n, S
  DO w = 1, n, n
   DO y = 1, n/p, n/p
    A[y,u]+=B[v,y]+C[w,v]
   ENDDO y
  ENDDO w
 ENDDO v
ENDDO u
```

Arrays A and C are column-major whereas the array B is row-major. By using our memory allocation scheme explained earlier, a tile of size $n/p \times S$ is allocated for A, of size $n \times S$ for C, and of size $S \times n/p$ for B. The final memory constraint is $2nS/p + nS \leq M$. Tile allocations are shown in Figure 4:E.

Next the algorithm considers the other alternative layout (row-major) for A.

$L^A.Q = \begin{bmatrix} \delta & \delta & \delta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$. Therefore $q_{14} = q_{21} = q_{22} = q_{23} = 0$ and $q_{24} = 1$.

$L^B.Q = \begin{bmatrix} \delta & \delta & 1 & 0 \\ \delta & \delta & 0 & 0 \end{bmatrix}$. Therefore $q_{13} = q_{34} = 0$ and $q_{33} = 1$.

$L^C.Q = \begin{bmatrix} \delta & \delta & 0 & 0 \\ \delta & \delta & 1 & 0 \end{bmatrix}$. Therefore $q_{43} = q_{44} = 0$. At this point $T^{-1} = Q = \begin{bmatrix} q_{11} & q_{12} & 0 & 0 \\ 0 & 0 & 0 & 1 \\ q_{31} & q_{32} & 1 & 0 \\ q_{41} & q_{42} & 0 & 0 \end{bmatrix}$. By setting $q_{12} = q_{31} = q_{32} = q_{41} = 0$ and $q_{11} = q_{42} = 1$, $T^{-1} = Q = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$.

The resulting code is as follows.

```
DO u = 1, n/p, S
 DO v = 1, n, S
  DO w = 1, n ,n
   DO y = 1, n, n
    A[u,y]+=B[w,u]+C[v,w]
   ENDDO y
  ENDDO w
 ENDDO v
ENDDO u
```

Arrays A and C are row-major whereas the array B is column-major. Tiles of size $S \times n$ are allocated for A and C, and a tile of size $n \times S$ is allocated for B. The final memory constraint is $3 \times n \times S \leq M$. Tile allocations are shown in Figure 4:G. Notice that since tile sizes are equal to array sizes for some array dimensions, the loops w and y disappear and the optimized code given earlier in Figure 4:C is obtained. Also note that although we could have chosen one of these two alternatives as the best alternative by using our criterion based on loop levels as explained earlier, we have presented here the tile allocations for both the alternatives for illustrative purposes.

We now revisit example shown in Figure 4, this time assuming fixed row-major disk layouts for all arrays. Due to lack of space, we do not show the formulation; but after our constraint-based algorithm is run, the tiles of size $S \times S$ are allocated for B and C, and a tile of size $S \times n$ is allocated for A. The final memory constraint is $nS + 2S^2 \leq M$. Tile allocations for this constrained version are shown in Figure 4:F. Note that the two array references in this constrained version are not optimized due to fixed layout requirement. This is the main reason that the locality optimizations adopted by out-of-core compilers should take into account both disk layouts and loop transformations.

4.3 Analytical Approach

In this subsection, using an analytical model, we evaluate the effectiveness of the algorithm presented in this paper at reducing the number of I/O calls. In order to get a clear formulation, we assume that at most n elements can be accessed in a single I/O call, where n being the array size in one dimension and the loop upper bound. Let C_f, t_f and M be the startup cost[2] for a file read (write), cost of reading (writing) an element from (into) file, and size of available memory respectively. If we assume that I/O cost of reading (writing) l consecutive array elements from (into) file can be approximated by $C_f + l \times t_f$, then the total I/O cost of the naive out-of-core translation shown in Figure 4:B (*excluding write costs*) is

$$T_{overall} = \underbrace{\frac{n^2 C_f}{S} + n^2 t_f}_{T_A} + \underbrace{\frac{n^3 C_f}{S^2} + \frac{n^3 t_f}{S}}_{T_B} + \underbrace{\frac{n^4 C_f}{S^3} + \frac{n^4 t_f}{S^2}}_{T_C}$$

under the memory constraint $3S^2 \leq M$. The T_A, T_B and T_C represent the I/O costs for the arrays A, B and C respectively. The overall I/O cost of the I/O optimized version (Figure 4:C), on the other hand, is

$$T_{overall} = \underbrace{nC_f + n^2 t_f}_{T_A} + \underbrace{nC_f + n^2 t_f}_{T_B} + \underbrace{\frac{n^2 C_f}{S} + \frac{n^3 t_f}{S}}_{T_C}$$

under the memory constraint $3nS \leq M$. The overall I/O costs for the versions with fixed layouts are computed similarly. These formulae clearly show that, for reasonable values of M, our algorithm is very effective at reducing the number as well as the volume of disk accesses. We computed the number of I/O calls for different input sizes for the example in Figure 4. Figures 5:A and B illustrate the curves representing the number of I/O calls (coefficient of C_f) for $2K \times 2K$ and $4K \times 4K$ double arrays respectively on logarithmic scales. We consider four different versions: unoptimized (naive) version (Ori), optimized version assuming fixed column-major layout for all arrays (Col), optimized version assuming fixed row-major layout for all arrays (Row), and the version that is optimized by our approach (Opt). It should be emphasized that the curves for the optimized versions are overestimates, because of the fact that we assume at most n elements can be read in a single I/O call. The effectiveness of our approach at reducing the number of I/O calls is clearly reflected on these curves. For example, for $4K \times 4K$ double arrays with a memory size of 10^3 elements , the number of I/O calls required for Ori, Col, Row and Opt versions are approximately 5×10^5, 2×10^5, 1.5×10^5 and 1.5×10^4 respectively.

4.4 Other Levels of Memory Hierarchy

Our algorithm is applicable to optimize the data traffic on memory-cache hierarchy as well. It is interesting to note that for some nests, the optimized loop order is the same for both caches and main memories. As an example, under column-major layouts for all arrays, for matrix-multiplication, our algorithm obtains a nest which is the same nest obtained by [7] and [8] for cache memories. The other optimized nest obtained by our algorithm for the matrix-multiplication is the same nest used in [6], for fixed row-major layouts. Since, even if the disk layouts are fixed, our approach considers all possible loop transformations; in the worst case it replicates the results based on the loop transformations alone to optimize the locality, if the temporal locality is not taken into account.

During the experiments it was observed that the I/O time constitutes at most 41% of the total execution time for our experimental suite. That is, in order to obtain the best performance in out-of-core computations both tiling loops and element loops should be optimized. After the disk layouts and the tiling loops are optimized by the algorithm presented in this paper, the element loops can be optimized by using the methods offered by McKinley *et al.* [8], Li [7] or Wolf and Lam [17].

5 Experimental Results

The experiments were performed by hand using PASSION [16], a run-time library for parallel I/O. PASSION routines can be called from C and Fortran, and different out-of-core arrays can be associated with different disk layouts. All the reported times are in seconds. The experiments were performed for different values of *slab ratio* (SR), the ratio of available node memory to the size of out-of-core local arrays combined. Notice that the SR is both an abstract variable and a good indicator of the behavior of the out-of-core programs under different memory constraints. Table 1 shows the I/O times of four different versions (Ori, Col, Row and Opt as explained above) of the example shown in Figure 4 on single nodes of IBM SP-2 and Intel Paragon. Figure 6 presents the normalized I/O times of these four versions with $4K \times 4K$ (128 MByte) double arrays on varying number of processors on SP-2. Figure 7 gives the speedups for Ori (original, unoptimized) and Opt versions.

We also applied our optimizations to a number of common kernels. The experiments were conducted on *four* nodes of SP-2 with $4K \times 4K$ two-dimensional and $4K$ one-dimensional double arrays, and the results are shown in Figure 8 as normalized I/O times. Figure 8:(A) gives the performance improvement obtained on a matrix-transpose loop that contains the statement $B(i,j) = A(j,i)$. Our algorithm assigns column-major layout for array A and row-major layout for array B. It then allocates a tile of size nS to A, and a tile of size Sn to B. The performance improvement on the innermost statement of the iterative-solver is given in Figure 8:(B). For this example, our heuristic associates column-major layout for all arrays. Figure 8:(C) shows the results of our optimizations on

[2] Startup cost can be thought as sum of the average values for seek time and rotational latency. Although the authors understand that a fixed startup cost is not very realistic; it is not clear, in their opinion, how a varying startup cost can be incorporated in a compilation framework.

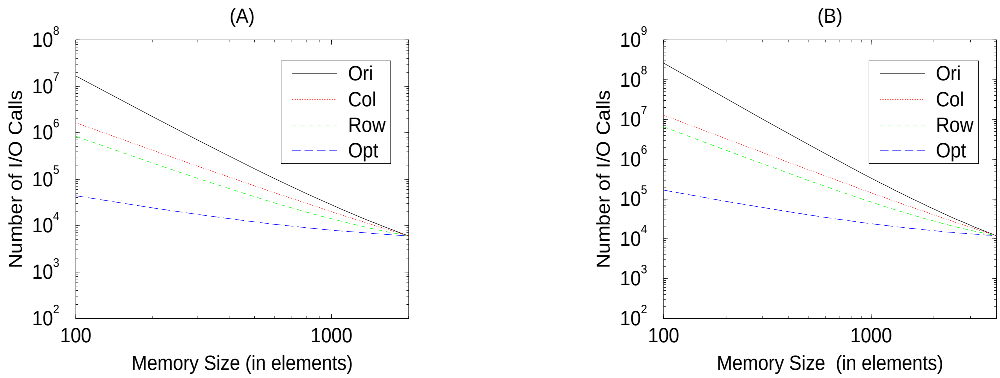

Figure 5: (A) Number of I/O calls for the example shown in Figure 4 for $2K \times 2K$ double arrays. (B) Number of I/O calls for the example shown in Figure 4 for $4K \times 4K$ double arrays.

Table 1: I/O times (in seconds) for the example shown in Figure 4 on SP-2 and Paragon.

	IBM SP-2								Intel Paragon							
	$2K \times 2K$ arrays				$4K \times 4K$ arrays				$2K \times 2K$ arrays				$4K \times 4K$ arrays			
SR	Ori	Col	Row	Opt	Ori	Col	Row	Opt	Ori	Col	Row	Opt	Ori	Col	Row	Opt
1/4	152	138	131	77	363	311	281	199	1159	1050	988	208	5172	4431	4001	1273
1/16	623	577	524	105	1441	1108	1074	441	2320	2141	1951	252	7360	5668	5412	1344
1/64	4224	3111	2462	563	8384	6449	5912	1422	19201	14141	11760	576	28864	22002	20257	2304
1/256	25087	21627	19298	1566	48880	35759	30055	4584	99583	83848	76245	3028	192512	141370	117270	8193

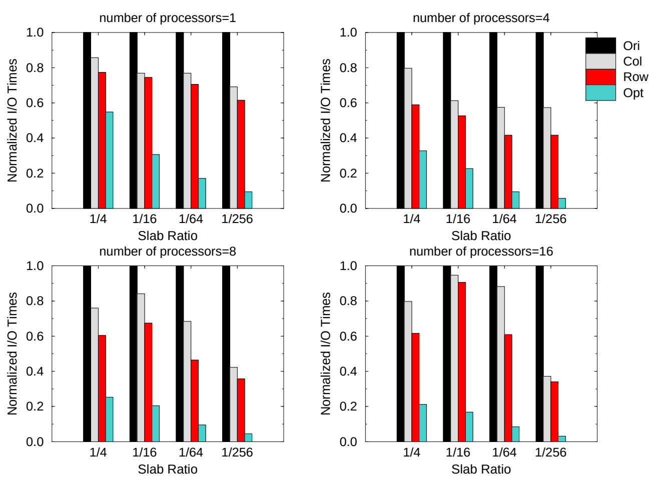

Figure 6: Normalized I/O times for our example with $4K \times 4K$ (128 MByte) double arrays on SP-2.

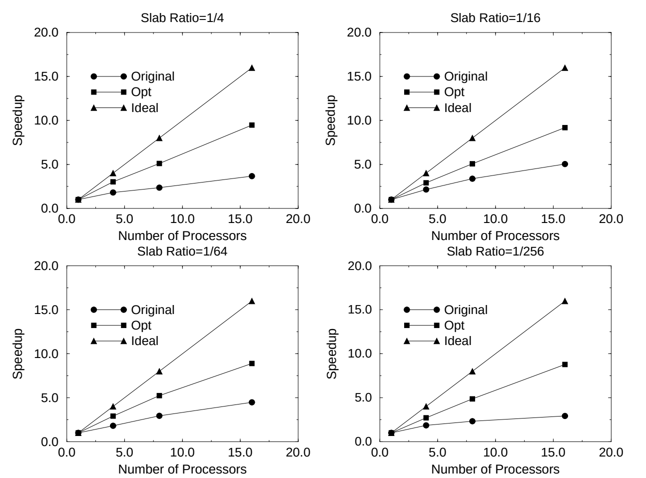

Figure 7: Speedups for unoptimized and optimized versions of our example with $4K \times 4K$ double arrays on SP-2.

matrix-smoothening (four-point relaxation) nest. Our algorithm associates row-major layout for all arrays. Finally Figure 8:(D) illustrate the performance improvement on matrix-vector multiplication (Y=AX). It is interesting to note that in this example all layout combinations are equally good as far as the I/O cost is concerned. Going with column-major layout for all arrays, our compiler allocates a data tile of size nS for A, a tile of size n for Y, and a tile of size S for X. From these results we conclude the following:

1. For all examples the Opt (I/O optimized) version performs much better than the Ori (naive) version. The reason for this result is that in the Opt version, array accesses are optimized as much as possible. For example as shown in Figure 4:G, by using the I/O optimization technique presented in this paper all three array accesses are optimized.

2. When the slab ratio is decreased, the effectiveness of our approach (Opt) increases (see Figure 6). As the amount of node memory is reduced, the Ori performs many number of small I/O requests, and that in turn degrades the performance significantly. The Opt version, on the other hand, continues with the optimized I/O no matter how small the node memory is.

3. As shown in Figure 7, the Opt version also scales better than the Ori (original, unoptimized) for all slab ratios.

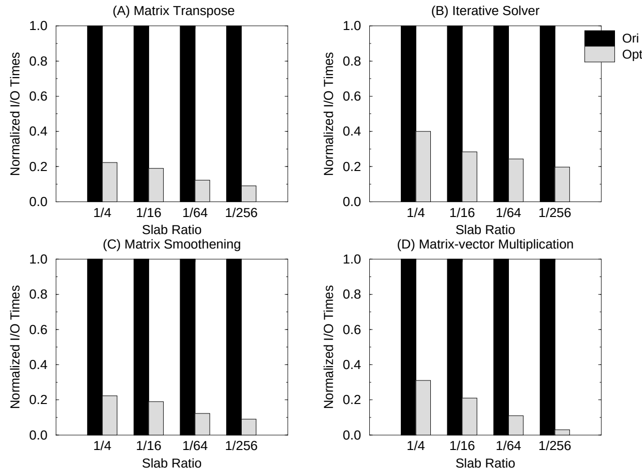

Figure 8: Normalized I/O times for different nests. (A) Matrix transpose. (B) Iterative solver. (C) Matrix smoothening. (D) Matrix-vector multiplication.

4. When the slab ratio is very small, the optimized versions with fixed layouts for all files also perform much better than the Ori (see Figure 6).

5. Experiments on two different platforms (Paragon and SP-2) with varying compile-time/run-time parameters such as available node memory, array sizes, number of processors etc., demonstrate that our algorithm is quite robust.

6 Global I/O Optimization

In this section, we show how our algorithm can be extended to work on multiple nests. Notice that a single loop nest with multiple LHSs (i.e. multiple statements) can be handled in a similar manner if we think the nest as distributed [18] over the individual statements.

Since a number of arrays can be accessed by a number of nests and these nests may require different layouts for a specific out-of-core array, the algorithm should find a layout for that array that satisfies the majority of the nests. In the following we present sketch of a simple heuristic. Our approach is based on the concept of *most costly nest*, the nest which takes the most I/O time. The programmer can use *compiler directives* to give hints about this nest, or we can use a metric such as multiplication of the number of loops and the number of arrays referenced in the nest. The nest which has the largest resulting value can be marked as the most costly nest. Then the algorithm proceeds as follows. First, the most costly nest is fully optimized by using the algorithm presented in this paper. After this step, disk layouts for some of the arrays will be determined. Then each of the remaining nests can be optimized using the approach presented for the constrained layout case in Section 4.1. After each nest is optimized, new layout constraints will be obtained, and these will be propagated for optimization of the next nest.

Figure 9 shows an algorithm for global I/O optimization. The functions Unconstrained() and Constrained() implement the algorithms for the unconstrained and constrained layout cases respectively. constraints refers to a set that holds the array layout constraints and updated after each nest is processed. Before the main loop, estimated cost of each nest i is computed using a metric and the nests are sorted according to non-increasing values of Cost(i) into Nest_List (e.g. Nest_List(1) is more costly than Nest_List(2) etc.). Then the most

```
for (each loop nest i) compute Cost(i);
endfor;
sort nests according to non-increasing values
  of Cost(i) into Nest_List;
Unconstrained(Nest_List(1),&new_constraints);
constraints = new_constraints;
while (there is a nest in the Nest_List)
  Current_Nest = next nest in the Nest_List;
  Constrained(Current_Nest,constraints,
              &new_constraints);
  constraints = constraints ∪ new_constraints;
endwhile;
```

Figure 9: A global I/O optimization algorithm

costly nest Nest_List(1) is optimized using Unconstrained() whereas the others are optimized inside the **while** loop using Constrained(). Notice that besides returning new_constraints both Unconstrained() and Constrained() also compute the loop transformation for each nest (not shown for clarity).

We note that the global I/O optimization problem is very similar to the problem of determining the appropriate data decomposition across the processors for multiple nests [9]. We are currently investigating whether or not the techniques developed for the data decomposition problem can be applied for the disk layout optimization for multiple nests.

7 Related Work

Iteration space tiling has been used for optimizing the cache locality in several papers [7, 17]. McKinley *et al.* [8] proposes an optimization technique consisting of loop permutation, loop fusion and loop distribution. They do decide individual file layouts but assume a column-major storage for all arrays. The assumption of a fixed layout strategy prevents some array references getting optimized as shown earlier in this paper, and that in turn may cause a substantial performance loss. But as indicated earlier, after our approach is applied, we strongly recommend the use of an in-core locality optimization technique for data tiles in memory.

In [3], a unified approach to locality optimization which employs both control and data transformations is presented for in-core problems in distributed shared-memory machines. This model can be adapted to out-of-core computations as well. But we believe our approach is better than that of [3] because of the following reasons: (1) The approach given in [3] depends on a *stride vector* whose value should be guessed by the compiler beforehand; our approach does not have such a requirement; (2) Our approach is more accurate, as it does not restrict the search space of possible loop transformations whereas the approach in [3] does; (and) (3) Our extension to multiple nests (global optimization) is also simpler than the one offered by [3] for global optimization, because we use the same algorithm for each nest. In the future, we intend to implement the algorithm in [3], enhance it by a suitable memory allocation scheme for out-of-core computations and compare it with our approach on different programs.

Another compiler-directed optimization, prefetching, is used by [10] for caches and by [11] for main memories. We believe that the compiler-directed prefetch is complimentary to our work in the sense that once the I/O time is reduced by our optimization, the remaining I/O time can be hidden by prefetching.

There has been a few papers on out-of-core compilation. The approaches can be divided into two groups: The first group considers optimizing the performance of virtual memory (VM). The most notable work is from Abu-Sufah *et al.* [1], which deals with optimizations to enhance the locality properties of programs in a VM environment. Among the program transformations used are loop fusion, loop distribution and tiling (page indexing). In principle, our disk layout determination scheme can be applied for optimizing the performance of the VM as well (by changing tile sizes to take the page size into account). But we believe that the impact of the I/O optimizations based on VM will be limited compared to those based on explicit file I/O due to the following reasons.

1. The fixed page sizes present a problem. Even if the computation requires a small portion of a data tile, a full page containing the data is brought into memory. Or, conversely, even if there is enough bandwidth for fetching a number of pages, the VMs generally bring one or two pages after every page fault, wasting the bandwidth.

2. The performance of the VM depends mostly on the page replacement policy of the operating system, which in turn, is out of control of the compiler. In particular, even if a chunk of data will not be used any more, the replacement policy can keep that chunk in memory for a long time [11].

In [4], the functionality of a ViC*, a compiler-like preprocessor for out-of-core C* is described. Several compiler methods for out-of-core HPF programs are presented in [15] and [2]. In [12], compiler techniques to choreograph I/O for applications based on high-level programmer annotations are investigated. The techniques in [15] and [12] are specifically designed for parallel machines whereas our approach can also be used on uniprocessors. To our knowledge, non of the previous work on out-of-core computations considers the compiler-directed disk layout transformations.

8 Summary and Conclusions

The difficulty of efficiently handling out-of-core data limits the performance of supercomputers as well as the enormous potential of parallel machines. Since coding out-of-core version of a problem might be a very onerous task and virtual memory does not perform well in scientific programs, we believe that there is a need for compiler-directed explicit I/O approach for parallel architectures. Unfortunately, fixed layout strategies adopted by current popular compilers prevent the potential spatial locality on disks from being exploited.

In this paper we presented a compiler algorithm to optimize the locality on disks by changing the access pattern and disk layouts. Our technique can easily be employed as a part of an out-of-core compilation framework like [4], [12] or [15]. An important characteristic of the approach is that it also optimizes the nests with arrays whose layouts are constrained. We show that this is important for global I/O optimization as well. An area of future work is integrating this technique with the techniques designed to eliminate I/O costs originating from communication requirements of out-of-core parallel programs [2]. We also intend to employ the algorithm presented in this paper in in-core compilers, and investigate its effectiveness at determining memory layouts and in improving cache performance.

References

[1] W. Abu-Sufah. On the Performance Enhancement of Paging Systems Through Program Analysis and Transformations, *IEEE Transactions on Computers*, C-30(5):341–355, 1981.

[2] R. Bordawekar, A. Choudhary and J. Ramanujam. Compilation and Communication Strategies for Out-of-core programs on Distributed Memory Machines. In *Journal of Parallel and Distributed Computing*, 38(2):277-288, Nov. 1996.

[3] M. Cierniak and W. Li. Unifying Data and Control Transformations for Distributed Shared Memory Machines. Technical Report 542, Dept. of Computer Science, University of Rochester, NY, Nov. 1994.

[4] T. H. Cormen and A. Colvin. ViC*: A Preprocessor for Virtual-Memory C*. Dartmouth College Computer Science Technical Report PCS-TR94-243, Nov. 1994.

[5] J. del Rosario and A. Choudhary. High Performance I/O for Parallel Computers: Problems and Prospects. *IEEE Computer*, Mar. 1994.

[6] M. S. Lam, E. Rothberg and M. E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *Proc. Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Apr. 1991.

[7] W. Li. Compiling for NUMA Parallel Machines, Ph.D. Thesis, Cornell University, Ithaca, NY, 1993.

[8] K. McKinley, S. Carr, and C.W. Tseng. Improving Data Locality with Loop Transformations. *ACM Transactions on Programming Languages and Systems,* 1996.

[9] U. Kremer. Automatic Data Layout for Distributed Memory Machines. Ph.D. thesis, Technical Report CRPC-TR95559-S, Center for Research on Parallel Computation, Oct. 1995.

[10] T. C. Mowry, M. S. Lam and A. Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proc. Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1992.

[11] T. C. Mowry, A. K. Demke and O. Krieger. Automatic Compiler-Inserted I/O Prefetching for Out-of-Core Applications. *Proc. Second Symposium on Operating Systems Design and Implementations*, Seattle, WA, Oct. 1996.

[12] M. Paleczny, K. Kennedy and C. Koelbel. Compiler Support for Out-of-Core Arrays on Parallel Machines. CRPC Technical Report 94509-S, Rice University, Houston, TX, Dec. 1994.

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

[14] J. Ramanujam and A. Narayan. Automatic data mapping and program transformations. In *Proc. Workshop on Automatic Data Layout & Performance Prediction*, Apr. 1995.

[15] R. Thakur, R. Bordawekar and A. Choudhary. Compiler and Run-time Support for Out-of-Core HPF Programs. In *Proc. 1994 ACM International Conference on Supercomputing*, pp. 382–391, Manchester, UK, Jul. 1994.

[16] R. Thakur, A. Choudhary, R. Bordawekar, S. More and K. Sivaramkrishna. PASSIONate Approach to High-Performance Parallel I/O, In *IEEE Computer*, Jun. 1996.

[17] M. Wolf and M. Lam. A Data Locality Optimizing Algorithm. In *Proc. ACM SIGPLAN 91 Conf. Programming Language Design and Implementation*, pp. 30–44, Jun. 1991.

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