A Unified Framework for Optimizing Locality, Parallelism, and Communication in Out-of-Core Computations

Mahmut Kandemir, *Member*, *IEEE*, Alok Choudhary, *Member*, *IEEE*,
J. Ramanujam, and Meenakshi A. Kandaswamy

Abstract—This paper presents a unified framework that optimizes out-of-core programs by exploiting locality and parallelism, and reducing communication overhead. For out-of-core problems where the data set sizes far exceed the size of the available in-core memory, it is particularly important to exploit the memory hierarchy by optimizing the I/O accesses. We present algorithms that consider both iteration space (loop) and data space (file layout) transformations in a unified framework. We show that the performance of an out-of-core loop nest containing references to out-of-core arrays can be improved by using a suitable combination of file layout choices and loop restructuring transformations. Our approach considers array references one-by-one and attempts to optimize each reference for parallelism and locality. When there are references for which parallelism optimizations do not work, communication is vectorized so that data transfer can be performed before the innermost loop. Results from hand-compiles on IBM SP-2 and Intel Paragon distributed-memory message-passing architectures show that this approach reduces the execution times and improves the overall speedups. In addition, we extend the base algorithm to work with file layout constraints and show how it is useful for optimizing programs that consist of multiple loop nests.

Index Terms—I/O-intensive codes, optimizing compilers, loop and data transformations, out-of-core computations, file layouts.

1 Introduction

It has been known for quite some time that the improvements in disk and memory speeds have not kept pace with improvements in processor speeds [16]. As a result, the issue of exploiting the memory hierarchy has emerged as one of the most important problems in efficiently using the available processing power. One way of handling this problem is to design algorithms that decompose the data sets into blocks and operate on blocks, maximizing their *reuse* before discarding them [13]. A computation that operates on disk-resident data sets is called *out-of-core* and an optimizing compiler for out-of-core computations is called an *out-of-core compiler* [7]. In contrast, a computation that operates on data sets in memory is called *in-core*. For out-of-core problems where the data set sizes far exceed the size of the available memory, it is particularly important to exploit the memory hierarchy as much as possible. Unfortunately, the task of optimizing out-of-core codes for I/O is very difficult and has not seen as much success as the efforts aimed primarily at optimizing parallelism and locality, such as [13]. We believe that an optimizing compiler for out-of-core programs faces several challenges:

- Since I/O accesses are generally at least an order of magnitude slower than interprocessor communication, I/O should be optimized as much as possible. The optimization of I/O may involve reducing both the number of I/O calls and the total number of elements (data items) transferred between the disks and main memory.
- As communication overhead is an important factor in performance, it should be performed only where necessary and through block transfers in order to amortize its high startup cost.
- The granularity of parallelism should be maximized as much as possible, as it is well-known that, having as many parallel outermost loop as possible reduces the synchronization and communication overhead and, as a result, the execution time on MIMD computers.

The difficulty here is that the issues of optimizing I/O, optimizing parallelism, and minimizing communication are interrelated. For example, for a given loop nest in which a number of out-of-core arrays are accessed, the I/O optimizations may imply a certain order for the loops, whereas the parallelism optimizations may suggest another. Bordawekar [7] shows that unoptimized I/O can also induce extra communication. Therefore, we believe that it is very important to develop compiler-based optimization strategies for I/O-intensive scientific codes that handle I/O,

- *M. Kandemir is with the Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA 16802. E-mail: kandemir@cse.psu.edu.*
- *A. Choudhary is with the Department of Electrical and Computer Engineering, Northwestern University, Evanston, IL 60208. E-mail: choudhar@ece.nwu.edu.*
- *J. Ramanujam is with the Department of Electrical and Computer Engineering, Louisiana State University, Baton Rouge, LA 70803. E-mail: jxr@ee.lsu.edu.*
- *M.A. Kandaswamy is with the Server Architecture Lab, Enterprise Server Group, Intel Corporation, Hillsboro, OR 97124. E-mail: meena.a.kandaswamy@intel.com.*

Manuscript received 20 Oct. 1997; revised 8 June 1999; accepted 16 Dec. 1999.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 105816.

communication and parallelism in a unified setting. In this paper, we offer such a *unified* strategy to optimize out-of-core programs for locality, parallelism, and communication. Specifically, our optimizations do the following: 1) reorganize data layouts in files and memory—matching the loop order with individual array layouts in files, which is key to obtaining high performance in out-of-core computations; 2) vectorize communication, i.e., perform the communication in large chunks of data; and 3) improve parallelism by transforming the loop nest such that the outermost loop(s) can run in parallel on a number of processors (this is not only good for reducing the communication requirements of parallel programs, but it also correlates with good memory/disk system behavior [40]).

While one may be tempted to view disks and files as another level in the memory hierarchy, there are two important issues that arise.

- Exploiting locality in out-of-core data is much more important than optimizing for cache locality since the disk access costs are 3 to 5 orders of magnitude higher than memory access costs, whereas memory access times are just an order of magnitude higher than cache accesses [16]; that is, in out-of-core codes, optimizing data transfer across the disk-main memory hierarchy is crucial.
- Cache optimizations are speculative since there is little software or programmer control. Virtual memory optimizations are similar. Unlike caches and virtual memory, explicit file I/O provides programmers/compilers with full control and, therefore, much greater potential for optimization, especially in regular scientific codes. For example, a compiler can generate code that uses explicit library calls to perform I/O at selected points of a given code.

In this paper, we present a compiler algorithm which, given an input program, generates an "optimized" out-of-core version of it with explicit I/O calls to a run-time library [38]. The resulting code is optimized in the sense that, compared to a naive (straightforward) out-of-core translation, it involves much less I/O, much less communication, and higher-granularity parallelism. In our approach, the compiler is in full control of the placement of I/O calls.

It is important to stress that our optimization domain is regular scientific codes. Currently, our approach works for only perfectly nested loops. If imperfect loops can be converted to perfect nests using loop fusion, fission, and code sinking [45], our approach can also handle them. Also, if a solution determined by our approach does not involve any loop transformations, we can also apply it to the imperfect nest being analyzed directly.

It is also important to note that we use the terms "optimizing" and "optimized" throughout this paper in the sense of "enhancing" or "improving." These are by no means a guarantee that the result of using our framework is an optimal solution (in the single or the multiple nest case) since many of these problems are NP-complete [29].

The remainder of this paper is organized as follows: In Section 2, we review the basic concepts in file layouts, tiling as well as general loop transformation theory. Sections 3 and 4 present automatic methods using the file locality, and the interprocessor communication which can be optimized, respectively. Section 5 presents a unified algorithm that 1) optimizes I/O, 2) minimizes communication, and 3) maximizes parallelism. Section 6 presents experimental results collected on an IBM SP-2 and an Intel Paragon multicomputer, which demonstrate the efficacy of the algorithm. In Section 7, we extend the base algorithm to handle the multiple-loop-nest case. Section 8 presents related work and, in Section 9, summary and conclusions are presented.

2 Preliminaries

2.1 Data Storage Model

We build our compiler optimizations upon a storage subsystem model, called the *Local Placement Model* (LPM) [38], [39], [7], which in principle can be implemented on any multicomputer. The main function of this subsystem is to isolate the peculiarities of the underlying I/O architecture and present a unified I/O interface to work with. Under this data storage subsystem, each *global out-of-core array* is divided into *local out-of-core arrays*. The local arrays of each processor are stored in separate files, called *local array files*, which in turn reside on a *logical local disk*. During the execution of an out-of-core program under the LPM, portions (blocks) of local out-of-core arrays, called **data tiles**, are fetched and stored in local memory. At any given time, computation is performed only on the data tiles in memory. The data sharing is performed through explicit message passing, so this system is a natural extension of the distributed-memory paradigm. So far, the LPM has been implemented on the IBM SP-2 and the Intel Paragon as a run-time library. Fig. 1 shows the local placement model; more details can be found in Bordawekar's thesis [7]. The run time library also allows users/compilers to define global and local out-of-core arrays, data tiles, and storage orders (e.g., row-major column-major) of data in files.

2.2 Reuse and Locality

We say a *temporal reuse* exists when two references access the same data element. *Spatial reuse,* on the other hand, occurs when two references access the same transfer unit such as a cache line, page, or data tile [45], [16]. When a loop nest exploits temporal (spatial) reuse, we say that the associated references exhibit temporal (spatial) locality. It should be noted that reuse does not necessarily mean locality [27]. To illustrate the concepts of reuse and locality, let us consider the following example.

$$
\begin{array}{l}
DO\ i = 1, n \\
\quad DO\ j = 1, n \\
\quad\quad A(i) = B(j) + C(i,j) + D(j,i) \\
\quad END\ DO \\
END\ DO.
\end{array}
$$

Assuming a *column-major* layout as the default, in this loop nest, array A has temporal reuse in the j loop and spatial reuse in the i loop. Array B has temporal reuse in the i loop and spatial reuse in the j loop. Arrays C and D, on the other hand, have only spatial reuses in the i and j loops, respectively; they do not exhibit any temporal reuse as

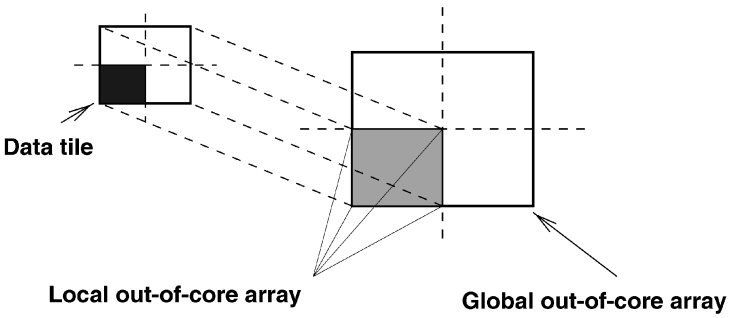


Fig. 1. The local placement model (LPM).

each element of C and D is accessed only once. Assuming that n is very large, only the reuses associated with the j loop exhibit locality. As a result, the *exploitable reuses* (that is, the reuses that can be converted into locality) for this nest are the temporal reuse for A, and the spatial reuses for B and D.

Note that if array C is stored in row-major order instead of the default column-major order, then spatial reuse for it can also be exploited in the innermost loop. In this paper, we mainly address the problem of selecting appropriate file layouts for out-of-core arrays, as well as suitable iteration space transformation for a given loop nest.

2.3 File Layouts

The file layout for an h-dimensional out-of-core array can be in one of $h!$ forms, each corresponding to the linear layout of data in file(s) by a nested traversal of the axes in some predetermined order. The innermost axis is called the *fastest-changing dimension*. As an example, for row-major file layout of a two-dimensional array, the second dimension is the fastest changing dimension. If we think of each element as a submatrix, our approach can also handle blocked file layouts. In other words, the method presented in this paper is applicable, with appropriate modifications, to the blocked layout case as well. That is, it can decide the storage order of blocks (tiles) for a given blocked out-of-core array. The order of elements within a block, on the other hand, can be determined using a cache locality optimization scheme (e.g., [27] and [44]). Since in out-of-core computations the arrays are very large, it might be difficult to exploit the locality beyond the fastest changing dimension. Consequently, our techniques try to determine the fastest changing dimension for a given array accurately. Even though the order of the other dimensions may impact the performance of an out-of-core program, their effect is of secondary importance. In this paper, we assume that the remaining dimensions can be ordered arbitrarily. However, our approach can be extended to determine a complete dimension order.

2.4 Loop Transformation Theory

The algorithms presented in this paper rely on results from the general loop transformation theory [27], [45], [44]. We focus on loops where both array subscripts and loop bounds are *affine functions* of enclosing loop indices and loop-invariant constants. 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 the *array reference* (*access*) *matrix*, $\vec{b}$ is *offset* (*constant*) *vector* [44]; $\vec{I}$ is a column vector, called the *iteration vector*, whose elements written left to right represent the loop indices $i_1, i_2, \cdots, i_n$, starting from the outermost loop to the innermost in the loop nest. For the rest of the paper, the reference matrix for array X will be denoted by $\mathcal{L}^X$ whereas the ith row of $\mathcal{L}^X$ will be denoted by $\vec{\ell}_i^X$. To clarify these concepts, we consider the following loop nest.

```
DO i = 1, n
  DO j = 2, n/2
    DO k = 1, n/2
      A(i, j) = B(j + k, i, j − 1)
    END DO
  END DO
END DO
```

The reference matrices and offset vectors are as follows:

$$\mathcal{L}^A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix},$$

$$\vec{b}^A = \begin{pmatrix} 0 \\ 0 \end{pmatrix};$$

and

$$\mathcal{L}^B = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix},$$

$$\vec{b}^B = \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix}.$$

Linear mappings between iteration spaces of loop nests can be modeled by nonsingular matrices [27], [45]. 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 (resp. direction) vector, on applying T, $T\vec{d}$ is the new distance (resp. direction) vector [45]. Since $\mathcal{L}\vec{I} = \mathcal{L}T^{-1}\vec{J}$, after the transformation T, $\mathcal{L}T^{-1}$ is the new array reference matrix [45]. In this paper, we denote T^{-1} by Q for convenience. An important characteristic of our approach is that using the array reference matrices, the entries of Q are derived systematically. For instance, if we transform the program shown above by using

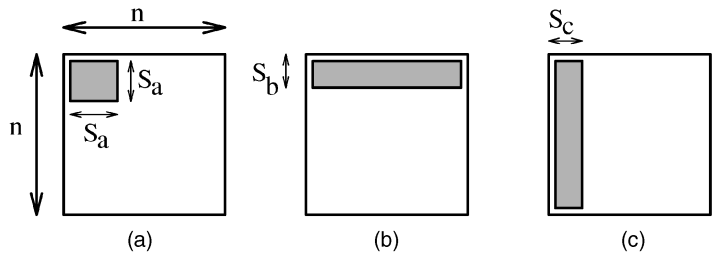


Fig. 2. (a) Unoptimized file access, (b) and (c) optimized file access.

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

as the transformation matrix, we obtain the following program.

$$\begin{array}{l} DO\ k = 1, n/2 \\ \quad DO\ j = 2, n/2 \\ \quad\quad DO\ i = 1, n \\ \quad\quad\quad A(i,j) = B(j+k, i, j-1) \\ \quad\quad END\ DO \\ \quad END\ DO \\ END\ DO \end{array}$$

2.5 Optimized I/O Accesses

We refer to an access to an array in a file as *optimized* if it can be performed such that all the required data along the fastest changing dimension will be read from the file incurring the least I/O cost. This concept can be best explained using an example. Consider the situation depicted in Fig. 2 for three different cases using a two-dimensional out-of-core array. The shaded portions denote data tiles (blocks). The case in Fig. 2a corresponds to the unoptimized case where the entire available memory is utilized for accessing a square tile from the corresponding file. In order to read an $S_a \times S_a$ data tile, S_a I/O calls should be issued no matter what the file layout is.[1] The cases shown in Fig. 2b and Fig. 2c, on the other hand, correspond to the optimized accesses for row-major and column-major file layouts, respectively. In Fig. 2b, it is possible to read $S_b \times n$ elements from the file using S_b I/O calls and, in Fig. 2c, $n \times S_c$ elements are read by issuing only S_c I/O calls (assuming for both these cases that "at most" n elements can be read by a single I/O call).

The following points should be noted: First, an optimized array access is only meaningful with a given corresponding file layout. For example, reading $S_b \times n$ elements from the array shown in Fig. 2b would cost n separate I/O calls if the array were stored in file as column-major. This observation indicates that the array layouts and I/O accesses should be compatible. Second, in order to have a fair comparison between programs, we *fix* the available memory size (M) no matter how the array layouts (and the I/O accesses) are optimized. As an example, for the cases shown in Figs. 2a, 2b, and 2c, assuming this is the only array referenced in the nest, the relation ${S_a}^2 = S_b n = n S_c = M$ should hold. And, finally, we should make a distinction between file and disk layouts. Depending on the storage style used by the underlying file system, a file can be striped across several disks [37]. Accordingly, an I/O call in the program can correspond to several system-level calls to disk(s). The technique described in the rest of the paper attempts to decide optimal file layouts, that is, the layouts that with accompanying loop transformation will incur reduced volume of I/O and reduced number of I/O calls. In general, the reduction in I/O calls to files leads to reduction in calls to disk; this relation, though, is system dependent and is not discussed in this paper.

1. We should note that, for blocked layouts, this type of access can be considered as optimized if all the elements inside the tile (block) are stored in the file consecutively.

2.6 Tiling for Out-of-Core Computations

Tiling, which is a combination of strip-mining and loop permutation, is a technique to improve locality and reduce synchronization overhead in in-core computations [31], [45], [23], [10], [18], [8]. When tiling is applied, it replaces an original loop with two new loops [45]: a *tiling loop* and an *element loop.* For example, the program shown first is transformed to the one shown second after the tiling. In the tiled program, σ refers to *tile size* or *blocking factor.* The tiling loops, *IT* and *JT*, iterate over the tiles, whereas the element loops, i and j, iterate over the elements of individual tiles. The compilers can use tiling to automatically create a blocked version of a given loop nest.

$$\begin{array}{l} DO\ i = 1, n \\ \quad DO\ j = 1, n \\ \quad\quad A(i,j) = C(j,i) \\ \quad END\ DO \\ END\ DO \end{array}$$

$$\begin{array}{l} DO\ IT = 1, n, \sigma \\ \quad DO\ JT = 1, n, \sigma \\ \quad\quad DO\ i = IT, min(IT+\sigma, n) \\ \quad\quad\quad DO\ j = JT, min(JT+\sigma, n) \\ \quad\quad\quad\quad A(i,j) = C(j,i) \\ \quad\quad\quad END\ DO \\ \quad\quad END\ DO \\ \quad END\ DO \\ END\ DO \end{array}$$

For in-core computations, tiling is usually an *optional* locality optimization technique applied to iteration spaces [44], [45], whereas, in an out-of-core compilation strategy based on *explicit file I/O*, tiling of out-of-core *data* into memory is *mandatory*; the compiler uses the results of dependence analysis [45] to determine whether or not tiling is legal (i.e., semantic-preserving). All necessary loop

transformations should be performed in order to ensure the legality of tiling.

A *naive* approach can extend the compilation methodology for in-core programs to out-of-core computations by assuming user-specified data decomposition (distribution) as follows: First, the out-of-core compiler applies the techniques used by in-core compilers [17] for message-passing machines to obtain the *node program*, that is, the program that will run on each node of the machine. After the node program is determined, the loops are tiled and appropriate I/O calls to the run-time library are inserted between the tiling loops. The available memory is then divided as evenly as possible among the arrays involved. There are several drawbacks to this straightforward approach [19]:

1. The program obtained by this method may not be able to exploit the available parallelism;
2. Assuming a fixed file layout such as row-major or column-major for all the arrays may adversely affect the performance of out-of-core codes [19], [20], [22]; and
3. When more than one array is involved in a computation, during memory allocation it might be more appropriate to favor (by giving more memory) the frequently accessed out-of-core arrays over the others.

Our optimizations address these problems within a unified framework. We give several examples and illustrate the following: *To achieve good performance in out-of-core computations, both data (file layout) and control (loop) transformations are necessary, and parallelism and locality should be handled in a unified way*. One point should also be discussed before we explain our approach in detail. The approach presented in this paper is based on explicit file I/O and is different from those presented in [1], [30], [32], [41] which target virtual memory (VM) based systems. While optimal file layouts and cache-friendly loop permutations are very useful for a VM-based frameworks as well, explicit file I/O through run-time calls gives the compiler full-control over the data transfers between disk and memory, as well as over the granularity of data tiles. In other words, the compiler can explicitly schedule I/O accesses and customize the tile sizes depending on the application being analyzed. In VM-based systems, on the other hand, these activities are controlled by the operating system in general using predefined page sizes.

3 Algorithm for Optimizing Locality in Files

In this section, we present an algorithm based on explicit I/O to reduce the time spent in I/O. Our algorithm automatically transforms a given loop nest to exploit spatial locality in files, assigns appropriate file layouts to out-of-core arrays, and partitions the available in-core memory among the data tiles of out-of-core arrays, all in a single unified framework.

3.1 Explanation of the Algorithm

The algorithm for optimizing file locality is shown in Fig. 3. Let $j_1, j_2, \cdots, j_n$ be the loop indices of the *transformed* nest (listed starting from the outermost position proceeding to the innermost). In the algorithm, C is the array reference on the left-hand-side (LHS), whereas A represents an array reference from the right-hand-side (RHS). The symbol δ stands for a *don't care* value. The algorithm works as follows.

3.1.1 Handling the LHS

The resultant loop transformation matrix should be such that the LHS array of the transformed loop nest should have the innermost index as the only element in one of the array dimensions and that index should not appear in any other dimension for this array.[2] In other words, after the transformation, the LHS array reference should be of the form $C(*, \cdots, *, j_n, *, \cdots, *)$, where j_n (the new innermost loop index) is in the rth dimension and $*$ indicates a term independent of j_n. This means that the rth row of the transformed reference matrix for C is $(0, 0, \cdots, 0, 1)$ and all the entries of the last column except the one in rth row are zero, that is, the transformed reference matrix is of the form

$$\begin{pmatrix} \cdots & \cdots & \cdots & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \vdots \\ \cdots & \cdots & \cdots & \cdots & 0 \\ 0 & 0 & \cdots & 0 & 1 \\ \cdots & \cdots & \cdots & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \vdots \\ \cdots & \cdots & \cdots & \cdots & 0 \end{pmatrix},$$

where the rth row is $(0, \cdots, 0, 1)$. After that process, the LHS array *can be* stored in the file such that the rth dimension is the fastest changing dimension in order to exploit spatial locality in the file. However, after this step, the out-of-core array is not stored in the file immediately according to the determined layout. Instead, the final layout for the LHS array is decided after considering all the alternatives.

3.1.2 Handling the RHSs

The algorithm then works on one reference from the RHS at a time. If a row s in the data reference matrix is identical to row r of the original reference matrix of the LHS array, it attempts to store this array on the file such that the sth dimension will be the fastest changing dimension. Note that the presence of such a row s does not guarantee that the array will be stored on the file with the sth dimension as the fastest changing dimension.

If the condition above does not hold for an RHS array A, then the algorithm tries to transform the reference to the form $A(*, \cdots, *, \mathcal{F}(j_{n-1}), *, \cdots, *)$, 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 in the second innermost loop. If no such transformation is possible, j_{n-2} is tried and so on. If there is no such transformation for any of the loop indices, then the remaining entries of Q are decided by taking into account the legality condition on the transformed data dependences and the nonsingularity requirement on the transformation matrix. Modified versions of the *dependence-sensitive*

2. This requirement is stricter than necessary, but suitable for the purposes of this paper.

Step 1 Initialize $i = 1$.

Step 2 Set $\vec{\ell_i}^C.Q = (0, 0, \cdots, 0, 1)$ and $\vec{\ell_k}^C.Q = (\delta, \delta, \cdots, \delta, 0)$ for each $k \neq i$ where δ denotes *don't-care*.

Step 3 Set the file 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 file 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 $\vec{\ell_j}^A.Q = (0, 0, \cdots, 0, 1, 0)$ and $\vec{\ell_k}^A.Q = (\delta, \delta, \cdots, \delta, 0, 0)$ 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 $\vec{\ell_j}^A.Q = (0, 0, \cdots, 1, 0, 0)$ and $\vec{\ell_k}^A.Q = (\delta, \delta, \cdots, \delta, 0, 0, 0)$ 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 *data dependences* and *non-singularity*.

Step 7 Repeat **Step 6** for all the reference matrices of a particular array A except those handled in **Step 4**. (Of course, all references for a particular A should have the same file layout; this algorithm is greedy and chooses the first possible layout)

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 file layouts, and choose the best alternative (see the explanation in Section 3).

Step 12 Determine the memory allocations for the all out-of-core arrays in the nest and obtain the *memory constraint*.

Step 13 Solve the memory constraint.

Fig. 3. Algorithm for optimizing file locality in out-of-core-computations.

completion algorithms found in [4] and [28] are used for filling the remaining entries such that all the data dependences in the original nest are maintained. Note that our approach determines only the fastest changing dimension of the layout.

3.1.3 Choosing the Best Alternative

After a transformation and corresponding file layouts are found, the next alternative for the LHS is tried and so on. Among the solutions recorded by **Step 9** of the algorithm, the *best* one (i.e., the one that results in the most locality) is chosen. Although several approaches can be used to select the best alternative, we chose the following scheme. Each loop in the nest is numbered with its level (depth), the outermost loop numbered 1. Then, for each reference in the nest, the level number of the loop whose index resides in the fastest changing dimension for this reference is checked. The number for all references in the nest are added and the alternative with the maximum sum is chosen. For example, if, for a two-deep nest with three references, a particular alternative exploits the locality for the first reference in the outer loop and for the other two references in the inner loop, the sum for this alternative is $1 + 2 + 2 = 5$. Intuitively, the alternative with the highest number will exploit the most spatial locality in the innermost loops. It is also possible to adopt a compile-time miss estimation technique (e.g., [14], (omitting conflict-miss estimation part) to obtain more accurate locality measurements.

3.1.4 Memory Allocation

The array references are divided into groups according to the layouts of the associated files (i.e., arrays with the same file layout are placed in the same group). The heuristic then handles the groups one by one. For each group, the algorithm considers *all* the fastest changing positions in turn. If a (tiling) loop index appears in the fastest changing position of a reference and does not appear in any other position (except the fastest changing) of any reference in that group, then the tile size for the fastest changing position is set to n (the array size and loop upper bound); otherwise, the tile size is set to S, a parameter whose value will be determined in the final step (we assume that $S \ll n$).

The tile sizes for the remaining dimensions are all set to *S*. After the tile sizes for all the dimensions of all array references are determined, the algorithm takes the size of the available node memory (*M*) into consideration and computes the actual value for *S*. For example, suppose that, in a four-deep nest in which four two-dimensional arrays are referenced, the previous steps have assigned row-major file layout for the arrays *A*, *B*, and *C*, and column-major file layout for the array *D*. Also, assume that *KT* is the innermost loop after the transformation and the references to the said arrays are *A[IT,KT]*, *B[JT,KT]*, *C[IT,JT]*, and *D[KT,LT]*. Our memory allocation scheme divides those references into two groups: *A[IT,KT]*, *B[JT,KT]*, *C[IT,JT]* in the row-major group and *D[KT,LT]* in the column-major group. Since *KT* appears in the fastest-changing positions of *A[IT,KT]* and *B[JT,KT]* and does not appear in any other position of any reference in this group, the tile sizes for *A* and *B* are determined as $S \times n$. Notice that *JT* also appears in the fastest-changing position. But, since it also appears in other positions of some other references (i.e., in the first dimension of *B[JT,KT]*) in this group, the algorithm determines the tile size for *C[IT,JT]* as $S \times S$. Then, it proceeds with the other group which contains the reference *D[KT,LT]* alone and allocates a data tile of size $n \times S$ for *D[KT,LT]*. After these allocations, the final *memory constraint* is determined as $3nS + S^2 \leq M$. Given a value for *M*, the value of *S* that utilizes all of the available memory can easily be determined by solving the second order equation $S^2 + 3nS - M = 0$ for positive values of *S*.

3.1.5 Important Observations

First, **Steps 2** and **6** of the algorithm given in Fig. 3 involve solving integer matrix equations. Second, the algorithm considers all possible fastest changing dimensions. For instance, for a three-dimensional out-of-core array, the algorithm may decide that the middle (second) dimension should be the fastest changing dimension; that is, it goes beyond the classical row-major and column-major array layouts found in conventional programming languages such as C and Fortran. Third, the algorithm first optimizes the LHS array. This is important from the I/O point of view because of the fact that the data tiles for this array *might* be both read and written, whereas the RHS arrays are only read. Finally, when the file layout of an out-of-core array is set to a specific form, the memory layouts of its data tiles in memory should also be set to the same form in order to reduce the transfer time between the disk and memory. Later on, cache locality optimization techniques [44], [27] can enhance the cache behavior taking the new memory layout into account.

3.2 Example

In this section, we give an example to illustrate the working of the algorithm shown in Fig. 3; this example will be used in the following two sections as well. Fig. 4a shows a matrix multiplication routine and Fig. 4b presents a straightforward out-of-core translation of it. In Fig. 4b, only the *tiling loops*—loops that iterate over the data tiles—are shown. Each reference corresponds to a data tile, the size and coordinates of which are determined by the relevant tiling loops. For example, in Fig. 4b, $C[u, v]$ denotes a data tile of size $S \times S$ from (u, v) to $(u + S - 1, v + S - 1)$ in file coordinates, whereas $C[w, u]$ in Fig. 4c corresponds to a data tile of size $n \times S$ (note the loop bounds and steps). Unless otherwise stated, the word *loop* refers to a tiling loop. In order to reduce clutter, all the element loops as well as the I/O calls between the tiling loops are omitted. The tile allocations for this naive translation are illustrated in Fig. 4e.

We now optimize the program shown in Fig. 4a using the locality algorithm shown in Fig. 3. Due to space limitations, we only show the successful trials. The reference matrices for the arrays are as follows:

$$L^C = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix},$$

$$L^A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix},$$

and

$$L^B = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}.$$

The algorithm works as follows:

First, it considers column-major file layout for *C*. Since

$$L^C.Q = \begin{pmatrix} 0 & 0 & 1 \\ \delta & \delta & 0 \end{pmatrix},$$

$q_{11} = q_{12} = q_{23} = 0$ and $q_{13} = 1$. Since

$$L^A.Q = \begin{pmatrix} 0 & 0 & 1 \\ \delta & \delta & 0 \end{pmatrix},$$

$q_{23} = 0$, which means there is no inconsistency[3] so far. However, the most optimized layouts create inconsistency for array *B*. Instead, since

$$L^B.Q = \begin{pmatrix} \delta & 1 & 0 \\ \delta & 0 & 0 \end{pmatrix},$$

we have $q_{22} = 0$ and $q_{32} = 1$. At this point,

$$T^{-1} = Q = \begin{pmatrix} 0 & 0 & 1 \\ q_{21} & 0 & 0 \\ q_{31} & 1 & 0 \end{pmatrix}.$$

By setting $q_{21} = 1$ and $q_{31} = 0$,

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

The resulting code is shown in Fig. 4c. The arrays *A*, *B*, and *C* have column-major file layouts. Tiles of size $n \times S$ are allocated for *C* and *A* and a tile of size $S \times S$ is allocated for *B* as shown in Fig. 4f. Since all the arrays have the same layout, during memory allocation there is only one group. The final memory constraint is $2nS + S^2 \leq M$.

Now, the algorithm tries the other file layout (row-major) for *C*. Since

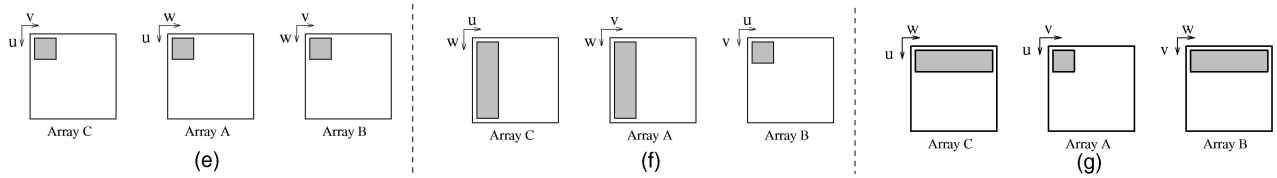
3. "Inconsistency" here means a situation where we happen to derive (from different equations) conflicting values for a given q_{ij} entry of Q.

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

```
DO u = 1, n, S
 DO v = 1, n, S
  DO w = 1, n, S
   C[u,v]+=A[u,w]*B[w,v]
  END DO
 END DO
END DO
```
(b)

```
DO u = 1, n, S
 DO v = 1, n, S
  DO w = 1, n, n
   C[w,u]+=A[w,v]*B[v,u]
  END DO
 END DO
END DO
```
(c)

```
DO u = 1, n, S
 DO v = 1, n, S
  DO w = 1, n, n
   C[u,w]+=A[u,v]*B[v,w]
  END DO
 END DO
END DO
```
(d)



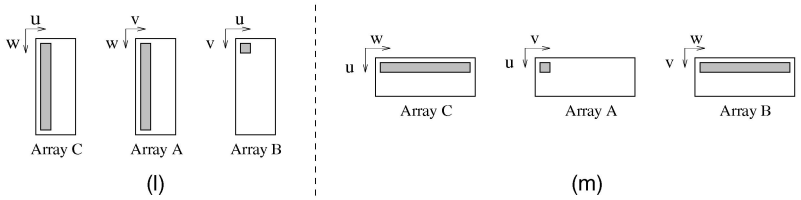
(e) (f) (g)

```
DO u = 1, n/p
 DO v = 1, n
  receive B(*,v)
  DO w = 1, n
   C(u,v)+=A(u,w)*B(w,v)
  END DO
 END DO
END DO
```
(h)

```
DO u = 1, n/p
 DO v = 1, n
  receive A(*,v)
  DO w = 1, n
   C(w,u)+=A(w,v)*B(v,u)
  END DO
 END DO
END DO
```
(i)

```
DO u = 1, n/p, S
 DO v = 1, n, S
  receive A[*,v]
  DO w = 1, n, n
   C[w,u]+=A[w,v]*B[v,u]
  END DO
 END DO
END DO
```
(j)

```
DO u = 1, n/p, S
 DO v = 1, n, S
  receive B[v,*]
  DO w = 1, n, n
   C[u,w]+=A[u,v]*B[v,w]
  END DO
 END DO
END DO
```
(k)



(l) (m)

Fig. 4. (a) Out-of-core matrix multiplication nest, (b) straightforward translation of (a) using square tiles, (c) I/O optimized translation of (a) with column-major file layouts, (d) I/O optimized translation of (a) with row-major file layouts, (e) tile allocations for (b), (f) tile allocations for (c), (g) tile allocations for (d), (h) parallelism optimized translation of (a) with communication calls for array *B*, (i) parallelism optimized translation of (a) with communication calls for array *A*, (j) parallelism and I/O optimized translation of (a) with column-major file layouts, (k) parallelism and I/O optimized translation of (a) with row-major file layouts, (l) tile allocations for (j), and (m) tile allocations for (k).

$$L^C.Q = \begin{pmatrix} \delta & \delta & 0 \\ 0 & 0 & 1 \end{pmatrix},$$

$q_{13} = q_{21} = q_{22} = 0$ and $q_{23} = 1$. Since

$$L^B.Q = \begin{pmatrix} \delta & \delta & 0 \\ 0 & 0 & 1 \end{pmatrix},$$

$q_{33} = 0$. Since

$$L^A.Q = \begin{pmatrix} \delta & 0 & 0 \\ \delta & 1 & 0 \end{pmatrix},$$

$q_{12} = 0$ and $q_{32} = 1$. At this point,

$$T^{-1} = Q = \begin{pmatrix} q_{11} & 0 & 0 \\ 0 & 0 & 1 \\ q_{31} & 1 & 0 \end{pmatrix}.$$

By setting $q_{11} = 1$ and $q_{31} = 0$,

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

The resulting code is shown in Fig. 4d. All the arrays have a row-major layout. Tiles of size $S \times n$ are allocated for *C* and *B*, and a tile of size $S \times S$ is allocated for A as shown in Fig. 4g. The final memory constraint is $2nS + S^2 \leq M$; it should be stressed that the parameter S in each optimized case is different and that, in general, its value depends on the memory constraint.

4 Algorithm for Maximizing Parallelism and Minimizing Communication

This section presents an algorithm that considers loop transformations to optimize parallelism and communication in message-passing machines. Specifically, the algorithm presented here transforms a loop nest such that 1) the transformed outermost loops are distributed over the

Step 1 Initialize $i = 1$.

Step 2 Set $\vec{\ell}_i^C.Q = (1, 0, \cdots, 0, 0)$, i.e., distribute LHS out-of-core array across processors along dimension i.

Step 3 For all array references A on the RHS that have $\vec{\ell}_l^A = \vec{\ell}_i^C$ for some l, distribute array A along the dimension l.

Step 4 Choose an array reference A for which the equality in **Step 3** does not hold. Initialize $j = 1$.

Step 5 Set $\vec{\ell}_j^A.Q = (0, 0, \cdots, 0, 1)$ and $\vec{\ell}_k^A.Q = (\delta, \delta, \cdots, \delta, 0)$ for each $k \neq j$. If a valid Q is found, check the determinant of it. If non-zero block transfers are possible for that RHS array, go to **Step 6**. If there are no valid Q or the determinant of Q is zero for all j, block transfers are not possible on that array with the given distribution of the LHS array; increment j and go to **Step 5**.

Step 6 Repeat **Step 5** for all reference matrices of a particular A.

Step 7 Repeat **Step 5** for all distinct array references.

Step 8 Record the obtained transformation matrix. Also record the number of arrays for which there is no communication and the number of arrays for which block transfers are possible.

Step 9 Increment i and go to **Step 2** (try a different distribution for the LHS array).

Step 10 Compare all alternatives and choose the best one.

Fig. 5. Algorithm for data decomposition and parallelism.

processors, 2) data decomposition across processors is determined for each out-of-core array, and 3) communication is performed in large chunks and is optimized such that all nonlocal data are transferred to respective local disks before the execution of the innermost loop.

4.1 Explanation of the Algorithm

As before, let $j_1, j_2, \cdots, j_n$ be the loop indices of transformed loop. The algorithm works as follows:

4.1.1 Handling the LHS

The resultant loop transformation matrix should be such that the LHS array of the transformed loop should have the outermost index as the only element in one of array dimensions. In other words, the LHS array C should be of the form $C(*, \cdots, *, j_1, *, \cdots, *)$, where j_1 (the new outermost loop index) is in the rth dimension. This means that the rth row of the transformed reference matrix for C is $(1, 0, \cdots, 0, 0)$. Then, the LHS out-of-core array can be distributed along the dimension r across processors without any communication. Note that distributing an out-of-core array means creating a corresponding local array file on each (logical) local disk (see Fig. 1).

4.1.2 Handling the RHSs

The algorithm works on one reference from the RHS at a time. If a row s of data reference matrix for an RHS array A is identical to a row in the reference matrix for the LHS array, then it is always possible to distribute that array along sth dimension across processors without any communication.

If the condition above does not hold for an RHS reference for an array A, then the entries for Q should be chosen such that some dimension of that reference consists only of the innermost loop index and the other dimensions are independent of the innermost loop index. That is, the RHS transformed reference should be of the form $A(*, \cdots, *, j_n, *, \cdots, *)$, where $*$ indicates a term independent of j_n. If this condition is satisfied, the communication arising from that RHS reference can be moved out of the innermost loop, that is, it is vectorized [17].

4.1.3 Refining Communication

An aggressive approach may repeat the previous step several times to take the communication to the outermost loop possible, constrained only by the data dependences.

Of course, the transformation matrix should be nonsingular and must satisfy data dependences. This algorithm is presented in Fig. 5 and the details of a version of this algorithm for in-core computations can be found elsewhere [12], [35], [36]. As before, this algorithm may result in more than one choice of layouts. Currently, we favor the choice that contains the most number of communication-free references.

4.2 Example

We consider the matrix multiplication nest shown in Fig. 4a again. The algorithm works as follows:

$$\mathcal{L}^C.Q = \begin{pmatrix} 1 & 0 & 0 \\ \delta & \delta & \delta \end{pmatrix}.$$

Therefore, $q_{11} = 1$, $q_{12} = 0$, and $q_{13} = 0$. Since $\vec{\ell}_1^{\,A} = \vec{\ell}_1^{\,C}$, A can be distributed along the first dimension as well.

$$\mathcal{L}^B.Q = \begin{pmatrix} 0 & 0 & 1 \\ \delta & \delta & 0 \end{pmatrix}.$$

Therefore, $q_{31} = q_{32} = q_{23} = 0$ and $q_{33} = 1$. The remaining entries should be selected such that the rank of Q is 3 and no dependences are violated. In this case, the algorithm can set $q_{21} = 0$ and $q_{22} = 1$. This results in the identity matrix meaning that no transformation is needed. A and C are distributed by rows and B by columns. The resulting node program is shown in Fig. 4h. Note that the communication is performed outside the innermost loop.

Next, the algorithm tries to distribute C in the second dimension.

$$\mathcal{L}^C.Q = \begin{pmatrix} \delta & \delta & \delta \\ 1 & 0 & 0 \end{pmatrix}.$$

Therefore, $q_{21} = 1$, $q_{22} = 0$, and $q_{23} = 0$. Since $\vec{\ell}_2^{\,B} = \vec{\ell}_2^{\,C}$, B can be distributed along the second dimension as well.

$$\mathcal{L}^A.Q = \begin{pmatrix} 0 & 0 & 1 \\ \delta & \delta & 0 \end{pmatrix}.$$

Therefore, $q_{11} = q_{12} = q_{33} = 0$ and $q_{13} = 1$. The remaining entries should be selected such that the rank of Q is 3 and no dependences are violated. The algorithm sets $q_{31} = 0$ and $q_{32} = 1$. This results in

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

All arrays are distributed by columns. The resulting node program is shown in Fig. 4i.

5 Unified Algorithm

This section presents a unified algorithm that combines the characteristics of the algorithms presented in the previous two sections. It attempts to optimize for parallelism, communication, and locality in a given loop nest that accesses a number of out-of-core arrays. In general, it is hard to find an optimal solution in this case and, therefore, the approach presented here is a greedy heuristic. It is greedy in the sense that it handles array references one at a time and the decisions made for a reference may affect the optimization of the others as well.

5.1 Explanation of the Algorithm

We first define the following terms where μ represents the number of loops in the nest:

- An array is said to be **optimized for parallelism** if it can be distributed along an array dimension where only j_1 (the transformed outermost loop index) appears; thus, there is no communication.
- An array is said to be **degree α optimized for communication** if it cannot be optimized for parallelism, but communication for it can be performed before the αth loop, where $1 \leq \alpha \leq \mu$. An array optimized for parallelism is said to be degree 0 optimized for communication; that is, it needs no communication.
- An array is said to be **degree β optimized for locality** if it contains the loop index $j_{n-\beta+1}$ in an array dimension and it can be stored in a file such that this array dimension will be the fastest changing array dimension ($1 \leq \beta \leq \mu$).

Using these definitions, we can associate a tuple (α, β) with each array where α and β denote the degree of communication and the degree of locality, respectively. The tuple $(0, 1)$ is the best possible tuple for an array. Our algorithm tries to achieve this best possible tuple for all the references. For those arrays where this is not possible, the selection of the next tuple to be considered depends on whether parallelism is favored over locality or vice-versa. For example, for a three-deep nest in which two-dimensional out-of-core arrays are accessed, we follow the sequence $(0, 1)$, $(0, 2)$, $(3, 1)$; that is, if an array reference cannot be optimized for parallelism, we check only for the case where the communication can be taken out of the innermost transformed loop. If $(3, 1)$ is unsuccessful, we choose to apply communication or locality optimization alone.

Theoretically, if there are enough loop indices and array dimensions, an array reference C can be transformed to the form $C(*, \cdots, *, j_1, *, \cdots, *, j_n, *, \cdots, *)$, where $*$ denotes a subscript independent of j_n. If such a transformation is possible, then C can be distributed among the processors along the dimension where j_1 occurs alone and, at the same time, the local portions of it can be stored in local files such that the dimension where j_n occurs will be the fastest changing dimension. The problem here is that, in most of the nests found in scientific programs, the number of loops and the number of array dimensions are small values; thus, the number of entries in T^{-1} is small (e.g., 4, 9, etc.). Once the above form is obtained for one reference, since most of the entries of T^{-1} are already determined, the chance of optimizing the other references would be low. This is why our algorithm considers other degrees of communication and locality as well.

Optimizing an array for a tuple (α, β) can be formulated as the problem of finding a transformed reference matrix that is suitable for both α degree communication and β degree locality. For example, the reference matrices corresponding to some commonly used (α,β) tuples for a three-deep nest and a four-deep nest in which two-dimensional arrays are accessed are given in Table 1.

The combined algorithm is given in Fig. 6. $\mathcal{L}^i$ denotes the original reference matrix for the ith array in the nest with $i = 1$ corresponding to the LHS array. The jth possible transformed reference matrix for an (α, β) tuple is denoted by $\mathcal{R}^j_{(\alpha,\beta)}$. In fact, this algorithm is a generic form of the previous algorithms in the sense that by setting the $\mathcal{R}^j_{(\alpha,\beta)}$ matrices to appropriate values, both the previous algorithms can be implemented.

5.2 Example

Consider the matrix multiplication nest once again. Since

$$\mathcal{L}^C.Q = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$$

with $(\alpha, \beta) = (0, 1)$, $q_{11} = 0$, $q_{12} = 0$, $q_{13} = 1$, $q_{21} = 1$, $q_{22} = 0$, and $q_{23} = 0$. Since

TABLE 1
Array Reference Matrices for Commonly Used (α, β) Tuples for a Two-Dimensional Array Enclosed in a Three-Deep Loop Nest (Top) and in a Four-Deep Loop Nest (Bottom)

(α, β)		
(0,1)	(0,2)	(3,1)
$\begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$ $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 0 \\ \delta & 1 & 0 \end{pmatrix}$ $\begin{pmatrix} \delta & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$	$\begin{pmatrix} 0 & 0 & 1 \\ \delta & \delta & 0 \end{pmatrix}$ $\begin{pmatrix} \delta & \delta & 0 \\ 0 & 0 & 1 \end{pmatrix}$

(α, β)		
(0,1)	(0,2)	(3,2)
$\begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}$ $\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} \delta & \delta & 1 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$ $\begin{pmatrix} 1 & 0 & 0 & 0 \\ \delta & \delta & 1 & 0 \end{pmatrix}$	$\begin{pmatrix} \delta & \delta & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$ $\begin{pmatrix} 0 & 0 & 1 & 0 \\ \delta & \delta & 0 & 0 \end{pmatrix}$

$$\mathcal{L}^A.Q = \begin{pmatrix} 0 & 0 & 1 \\ \delta & \delta & 0 \end{pmatrix}$$

with $(\alpha, \beta) = (3, 1)$, $q_{33} = 0$. Since

$$\mathcal{L}^B.Q = \begin{pmatrix} \delta & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

with $(\alpha, \beta) = (0, 2)$, $q_{32} = 1$. By setting $q_{31} = 0$,

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

The resulting node program is shown in Fig. 4j. The three arrays are distributed column-wise among the processors. The arrays *C* and *B* are optimized for parallelism ($\alpha = 0$), whereas the array *A* is optimized for communication with $\alpha = 3$. The arrays *C* and *A* are optimized for locality in the innermost loop, whereas for array *B* the locality is exploited in the second loop. The tile allocations for local arrays are shown in Fig. 4l. All the local arrays are column-major.

Next, the algorithm considers the other alternative for the array *C*. Since

$$\mathcal{L}^C.Q = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

with $(\alpha, \beta) = (0, 1)$, $q_{11} = 1$, $q_{12} = 0$, $q_{13} = 0$, $q_{21} = 0$, $q_{22} = 0$ and $q_{23} = 1$. Since

$$\mathcal{L}^A.Q = \begin{pmatrix} 1 & 0 & 0 \\ \delta & 1 & 0 \end{pmatrix}$$

with $(\alpha, \beta) = (0, 2)$, $q_{32} = 1$ and $q_{33} = 0$. Since

$$\mathcal{L}^B.Q = \begin{pmatrix} \delta & \delta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

with $(\alpha, \beta) = (3, 1)$, $q_{32} = 1$. By setting $q_{31} = 0$,

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

The resulting node program is shown in Fig. 4k. All the arrays are row-wise decomposed across processors. The arrays *C* and *A* are optimized for parallelism ($\alpha = 0$), whereas the array *B* is optimized for communication with $\alpha = 3$. The arrays *C* and *B* are optimized for locality in the innermost loop, whereas for array *A* the locality is exploited in the second loop. The tile allocations for local arrays are shown in Fig. 4m.

6 Experimental Results

The experiments were performed on an IBM SP-2 and an Intel Paragon, for different values of *Slab Ratio* (*SR*), the ratio of available node memory to the total size of all the out-of-core local arrays. *SR* is an appropriate parameter as it gives us the opportunity to investigate the behavior of an out-of-core program under tight memory constraints. The practical values for the *SR* parameter is between $\frac{1}{4}$ and $\frac{1}{256}$ [7]. The transformations to the original programs were applied manually following the algorithms by using the PASSION run-time library [7], which can associate different file layouts with different out-of-core arrays in a given nest. All the reported times are in *seconds* and obtained by instrumenting the code being analyzed.

Step 1 Initialize $i = 1$. Initialize $(\alpha, \beta) \leftarrow (0, 1)$ (try the best possible optimization).

Step 2 Initialize $j = 1$. (try the first transformed reference matrix for this (α, β) tuple).

Step 3 Set $\mathcal{L}^i.Q = \mathcal{R}^j{}_{(\alpha,\beta)}$. If there is no inconsistency, then go to **Step 4**; else increment j (try the next possible transformed reference matrix for this (α, β) tuple) and repeat this step. If there are inconsistencies for every value of j, then increment (α, β) tuple (try the next tuple on the trial sequence) and repeat this step. If there are inconsistencies for all (α, β) tuples, then apply pure communication or pure locality optimization for this reference.

Step 4 Increment i and go to **Step 2** (optimize the next array reference).

Step 5 When a Q is found, record it. Also record the associated (α, β) tuples for each array reference.

Step 6 When all solutions are obtained, choose the best alternative by comparing (α, β) values, and apply the memory allocation scheme.

Fig. 6. Unified algorithm for optimizing parallelism, communication, and locality.

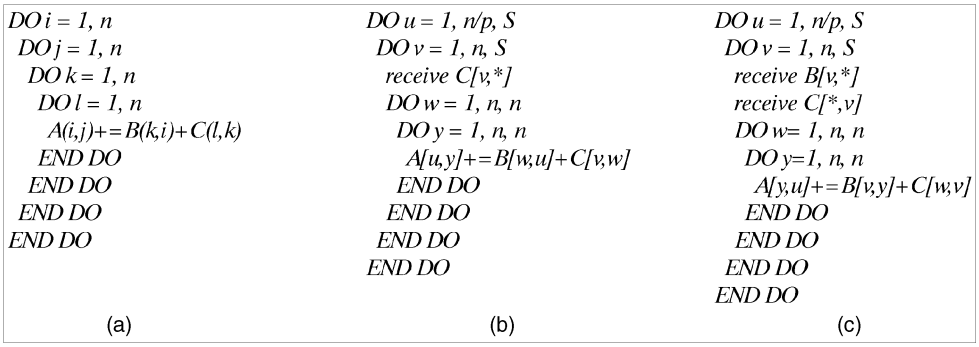


Fig. 7. (a) An example out-of-core loop nest, (b-c) optimized versions of (a).

TABLE 2
I/O Times (in Seconds) for the Example Shown in Fig. 7a on the SP-2 and the Paragon

	IBM SP-2							
	2K×2K double arrays				$4K \times 4K$ double arrays			
SR	Ori	Col	Row	Opt	Ori	Col	Row	Opt
1/4	152	138	131	77	363	311	281	199
1/16	623	577	524	105	1,441	1,108	1,074	441
1/64	4,224	3,111	2,462	563	8,384	6,449	5,912	1,422
1/256	25,087	21,627	19,298	1,566	48,880	35,759	30,055	4,584

	Intel Paragon							
	2K×2K double arrays				4K×4K double arrays			
SR	Ori	Col	Row	Opt	Ori	Col	Row	Opt
1/4	1,159	1,050	988	208	5,172	4,431	4,001	1,273
1/16	2,320	2,141	1,951	252	7,360	5,668	5,412	1,344
1/64	19,201	14,141	11,760	576	28,864	22,002	20,257	2,304
1/256	99,583	83,848	76,245	3,028	192,512	141,370	117,270	8,193

6.1 Experimental Platforms and Run-time Library

We use a 512-node Intel Paragon with i860 processors for the experiments discussed in this section. The Parallel File System (PFS) is part of the OSF.R1.4.3 release of the operating system. There are two different parallel file system configurations that can be used, namely a 12 I/O node X 2 GB partition on original Maxtor RAID 3 level disks and a 16 I/O node X 4 GB partition on individual Seagate disks. In the experiments, we used the former configuration. In both the partitions, the stripe factor is equal to the number of I/O nodes. The default striping unit size of both the I/O partitions is 64 KB.

The IBM SP-2 is a distributed-memory machine that uses Ethernet and switches to connect the nodes. It uses RS/6000 processors. The nodes are divided into two groups: thin nodes and wide nodes. In the experiments, we use thin nodes, each of which runs at 120MHz, has 256 MBytes of memory, and 4.5 GBytes of local disk space. The PIOFS, IBM's parallel file system running under AIX operating system, supports shared file accesses.

The PASSION run-time library [38], [39] uses PFS and PIOFS on Paragon and SP-2, respectively. It provides a high-level interface that makes it easy for the compiler to specify which portion of the out-of-core data structure needs to be read from or written to the file and the internal routines perform all the necessary I/O efficiently using the interface to the underlying parallel file system.

6.2 Performance Numbers for an Out-of-Core Loop Nest

Our experimental suite consists of several loop nest from benchmarks and math libraries. But, we first present detailed performance numbers on a small loop nest that can benefit from unified (loop+data) locality improvements.

6.2.1 Optimization

Fig. 7a shows a four-deep loop nest that can benefit from file layout optimizations. The application of our algorithm results in two optimized node programs, as shown in Fig. 7b and Fig. 7c, respectively. In Fig. 7b, the reference *A* is optimized with $(0,1)$, the reference *B* is optimized with $(0,2)$, and the reference *C* is optimized with $(3,2)$. Before the w-loop, communication is performed for *C*. On the other hand, in Fig. 7c, the reference *A* is optimized with $(0,1)$ and the reference *C* is optimized with $(3,2)$, incurring communication before the w-loop. The reference *B* could only be optimized for locality and communication is needed for it before the w-loop. In our experiments, the code in Fig. 7b outperformed the one in Fig. 7c, so, in the following, we consider only the former.

6.2.2 I/O Times

Table 2 shows the I/O times on a single processor and in Fig. 8 and Fig. 9, we present the normalized I/O times on

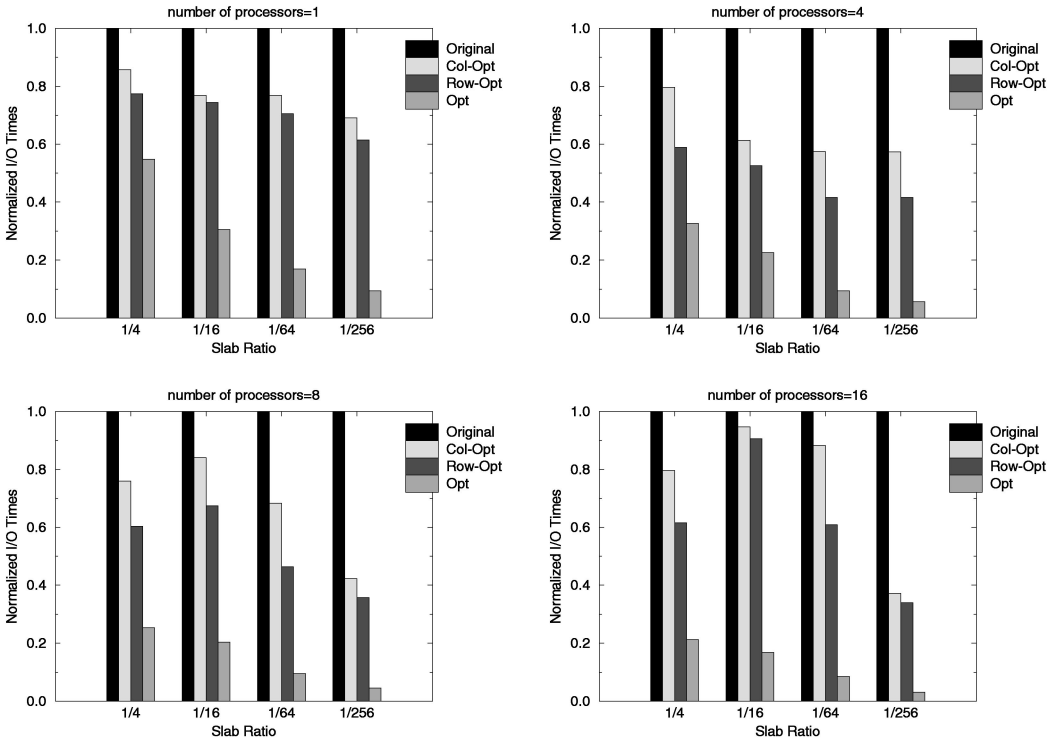


Fig. 8. Normalized I/O times with $4\mathrm{K} \times 4\mathrm{K}$ (128 MByte) double arrays on the SP-2 (example shown in Fig. 7a).

the SP-2 and the Paragon, respectively, for *four* different versions of this example using $4K \times 4K$ out-of-core arrays:

1. `Original` or `Ori`: The original program in Fig. 7a is modified manually for parallelism and the data distributions are applied to files holding the out-of-core arrays. A fixed column-major file layout is used for all out-of-core arrays and square tiles are read/written.
2. `Col-Opt`: The *optimized* program using our approach under *fixed column-major layouts* for all arrays. This is similar to the loop-level locality optimization techniques used by commercial compilers. In that case, our algorithm allocates data tiles of size $S \times S$, $S \times S$, and $n \times S$ for the A, B, and C, respectively, resulting in the memory constraint $nS + 2S^2 \leq M$, where M is the size of the node memory.
3. `Row-Opt`: The *optimized* program under *fixed row-major layouts* for all arrays. The algorithm allocates data tiles of size $S \times n$, $S \times S$, and $S \times S$ for the A, B, and C, respectively, resulting in the memory constraint $nS + 2S^2 \leq M$.
4. `Opt`: The program *optimized by our approach* assuming no fixed file layouts (Fig. 7b). A and C are row-major, while B is column-major.

The algorithm allocates data tiles of sizes $S \times n$, $n \times S$, and $S \times n$ for the A, B, and C, respectively, resulting in the memory constraint $3nS \leq M$. Notice that, in this case, all the three array accesses are optimized. Notice that, for all the versions, the final code is tiled because tiling is mandatory for out-of-core computations based on explicit file I/O as mentioned earlier.

Table 3 shows the number of bytes read and number of I/O calls issued for each of the four versions. It is easy to see that the `Opt` version minimizes both the number of I/O calls in the program and the number of bytes transferred from the files resulting in a corresponding reduction in the overall I/O time.

6.2.3 Speedups

Fig. 10 shows the speedups for the `Original` and `Opt` versions on the SP-2 using $4K \times 4K$ array. Note that each speedup is relative to the sequential version of the same program (Original or Opt). Speedup curves for the Paragon exhibit a similar trend and, therefore, they are omitted.

6.2.4 I/O Bandwidth

The *I/O bandwidth* (also called the *aggregate read bandwidth*) of an out-of-core program is computed as the total number of bytes read by all processors divided by the total time to read. The optimized programs have better bandwidth speedups than their unoptimized counterparts. Fig. 11 shows two different cases for the example given in Fig. 7a: 1) $4\mathrm{K} \times 4\mathrm{K}$ double arrays with a slab ratio of $\frac{1}{64}$ and 2) $2\mathrm{K} \times 2\mathrm{K}$ double arrays with a slab ratio of $\frac{1}{256}$. Note that each bandwidth speedup is relative to the I/O bandwidth of the same version on a single processor. Therefore, the bandwidth speedups for the original and optimized cases start from the same point when $p = 1$, even though the actual values are different for each version. The I/O bandwidth of the optimized program when $p = 1$ is 6.23 MB/sec, whereas, when $p = 16$, the bandwidth is 55.3 MB/sec. Recall that the `Opt` version is the one shown in Fig. 7b.

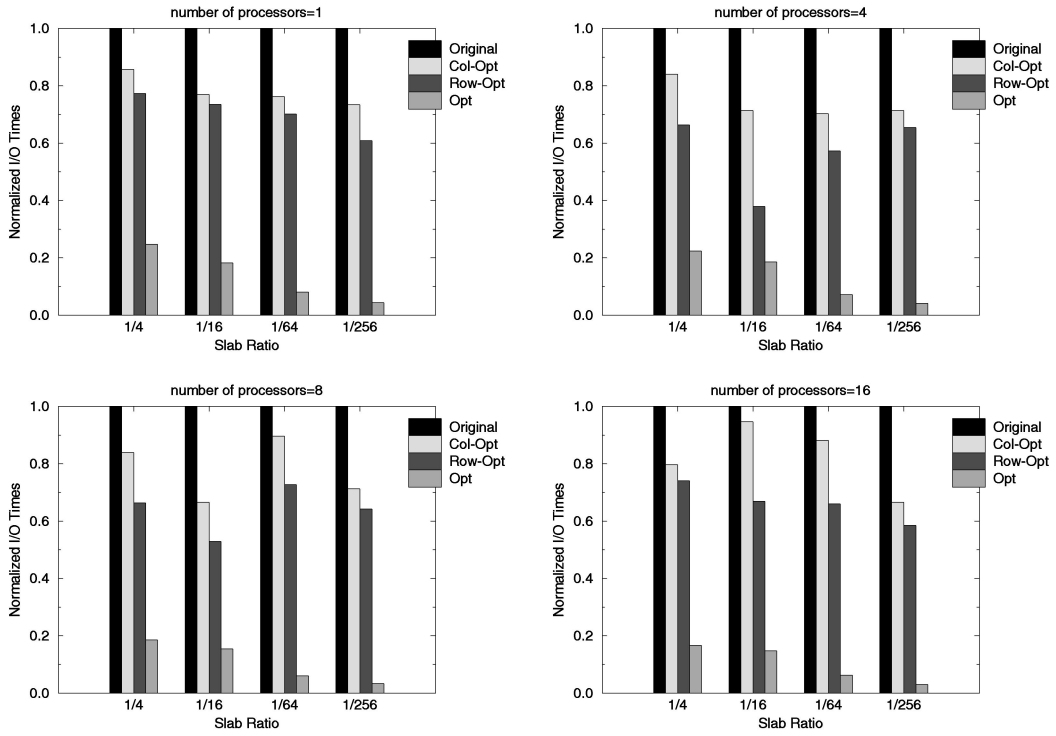


Fig. 9. Normalized I/O times with $4\mathrm{K} \times 4\mathrm{K}$ (128 Mbyte) double arrays on the Paragon (example shown in Fig. 7a).

TABLE 3
Number of Mbytes Read and Number of I/O Calls Issued (Example Shown in Fig. 7a)

SR=1/4	2K×2K double arrays				4K×4K double arrays			
	Ori	Col	Row	Opt	Ori	Col	Row	Opt
Number of Mbytes read	235	235	134	134	940	940	537	537
Number of I/O calls issued	28,672	20,480	14,336	8,192	57,344	40,960	28,672	16,384

6.2.5 Observations

From these results we observe the following:

- The `Opt` version performs better than the other versions for this out-of-core nest.
- For a fixed slab ratio (*SR*), when the number of processors is increased, the effectiveness of our approach (`Opt`) increases (see Fig. 8 and Fig. 9). This is because of the fact that more processors are now working on the out-of-core local array(s) I/O-optimally.
- For a fixed number of processors, when the slab ratio (*SR*) is decreased, the effectiveness of our approach increases (see Fig. 8 and Fig. 9). As the amount of node memory is reduced, the `Original` version performs many number of small I/O requests, and this degrades the performance dramatically.
- As shown in Fig. 10, the `Opt` version also scales better than `Original` for all slab ratios.
- As shown in Fig. 11, the `Opt` version also has a better I/O bandwidth speedup than `Original`.
- Demonstrations on two different platforms with varying compile-time and run-time parameters, such as the number of processors, available in-core node memory, array sizes, etc., prove that the algorithm is robust.
- The results presented here are conservative in the sense that the unoptimized code is also modified such that the outermost loops are parallelized. Since this may not always be the case, the performance improvement obtained by our approach will be higher in general.

6.3 Performance Numbers for Several Programs

In this subsection, we present experimental results obtained on the Intel Paragon. In the experiments, we applied the following methodology: We took 10 loop nests from several benchmarks and math libraries. The salient features of these nests are shown in Table 4. Then, we parallelized these nests for execution on the Paragon such that interprocessor communication is eliminated. This allowed us to focus solely on the I/O performance of the nests and the scalability of the I/O subsystem.

After this parallelization and data allocation, we hand-coded *five* different out-of-core versions of each nest using the PASSION runtime library [38]. The `Col` and `Row` are the original (unoptimized) codes. The `L-Opt` version is the optimized code obtained using loop transformations alone. For this version, we used the best of the resulting nests generated by [27] and [44] (it corresponds to the `Col-Opt` version discussed earlier and does not use any file layout transformation). The `D-Opt` version is the layout-optimized

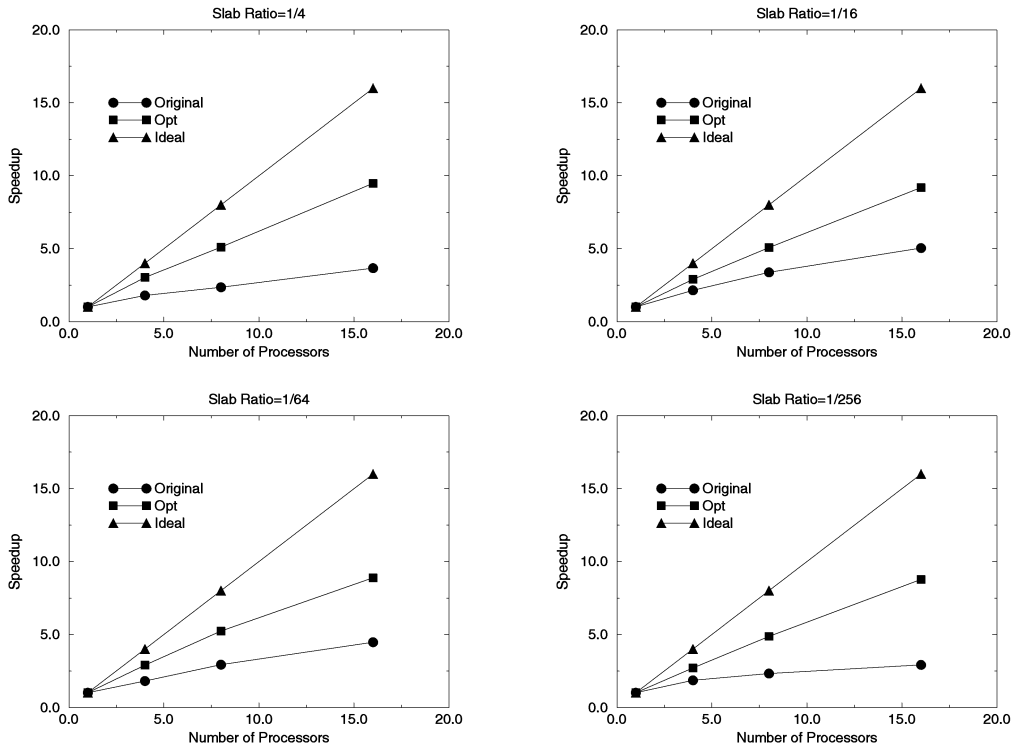


Fig. 10. Speedups for unoptimized and optimized versions of the example shown in Fig. 7 with $4\mathrm{K} \times 4\mathrm{K}$ double arrays on the SP-2.

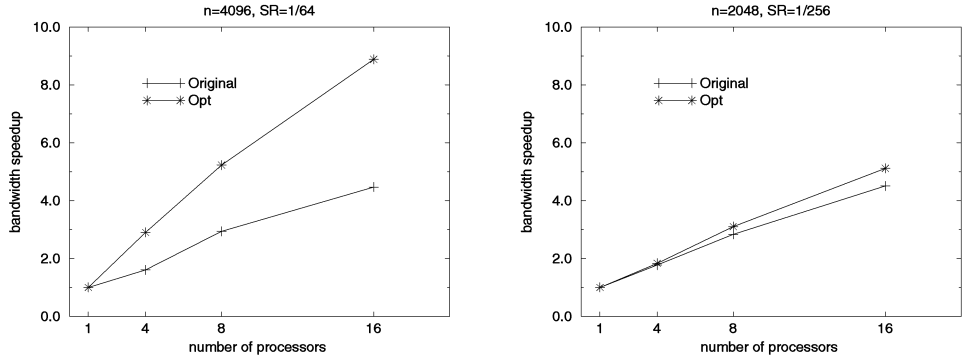


Fig. 11. Bandwidth speedups for the example shown in Fig. 7 on the SP-2.

version without any loop transformation. For this, we used the best of the resulting nests generated by [33] and [26]. The `Opt` (compiler optimized) version is the one obtained using the approach discussed in this paper. `H-Opt` is a hand-optimized version. 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 `Opt`, all the loops carrying some form of reuse are tiled. As usual, we use all versions to employ tiling to bring the right amount of data into memory.

For each nest, we set the slab ratio (*SR*) to $\frac{1}{128}$. 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 5 shows the results on 16 processors of the Paragon. For each data set, the `Col` column gives the total execution time of the out-of-core nest in *seconds*. The other columns, on the other hand, give the respective execution times for the other versions as a fraction of that of `Col`. As an example, the execution time of the `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 the average, `L-Opt` brings only a 16 percent improvement over `Col`. The approach based on pure data transformations performs much better. Our integrated approach explained in this paper, however, results in a 40 percent reduction in the execution times with respect to `Col`. Using a hand optimized version (`H-Opt`) brings an additional 8 percent reduction over `Opt`, which encourages us to incorporate array chunking and interleaving into our technique.

Table 6, 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 on the Paragon. It should be stressed that, in obtaining these speedups, we used the

TABLE 4
Programs Used in Our Experiment

Nest	Source	iter	Arrays
mat.2	-	2	three 2-D
mxm.2	spec92	3	three 2-D
adi.2	livermore	5	three 1-D, three 3-D
vpenta.6	spec92	3	seven 2-D, two 3-D
btrix.4	spec92	2	twenty-five 1-D, four 4-D
emit.3	spec92	2	ten 1-D, three 3-D
syr2k.2	blas	2	three 2-D
htribk.2	eispack	3	five 2-D
gfunp.4	hompack	3	one 1-D, five 2-D
trans.2	nwchem	3	two 2-D

The `iter` column for each code shows the number of iterations of the outermost timing loop. The number at the end of the program name denotes the loop nest used. The `arrays` gives the number and the dimensionality of the arrays accessed by the program from which the nest is extracted. `mat.2` is the main loop nest in the classical matrix-multiply nest.

TABLE 5
Experimental Results on 16 Nodes of an Intel Paragon

Nest	Col	Row	L-Opt	D-Opt	Opt	H-Opt
mat.2	257.20	93.3	65.1	56.8	60.8	54.3
mxm.2	220.01	181.5	100.0	112.6	79.8	67.0
adi.2	144.12	134.9	22.8	46.5	22.8	22.8
vpenta.6	135.00	47.1	100.0	47.1	47.1	29.9
btrix.4	91.45	66.6	100.0	61.3	61.3	42.3
emit.3	88.64	176.5	100.0	100.0	100.0	100.0
syr2k.2	215.34	86.3	52.0	77.4	52.0	47.6
htribk.2	248.61	110.8	127.2	81.1	81.1	72.6
gfunp.4	86.05	128.4	73.3	68.0	46.9	34.0
trans.2	181.90	100.0	100.0	48.2	48.2	48.2
average:		112.5	84.0	69.9	60.0	51.9

single node result of the respective versions. For example, the speedup for the `Opt` version of `emit.3` was computed for $p \in 16, 32, 64, 128$ as

$$\frac{\text{Execution Time of the } \texttt{Opt} \text{ version of } \texttt{emit.3} \text{ on 1 node}}{\text{Execution Time of the } \texttt{Opt} \text{ version of } \texttt{emit.3} \text{ on } p \text{ nodes}}.$$

Since the execution times of the parallelized codes on single nodes may not be as good as that of 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. Nevertheless, these results demonstrate that our approach performs very well in practice.

7 Global I/O Optimization

We have shown that data transformations are very effective for out-of-core computations, especially when they are accompanied by iteration space transformations. However, unlike the impact of iteration space transformations whose scope is a single loop nest, the effect of a data transformation is *global;* that is, when an array layout is determined, it should be used for all the references to the same array in all the loop nests. Since dynamic redistribution of out-of-core data at run-time can be very costly, it should be considered only in exceptional cases. In this section, we show how our algorithm can be extended to work for multiple nests. Since a number of out-of-core arrays can be accessed by a number of nests and each of these nests may, in principle, require a different file layout for a specific array, the algorithm should determine a single static file layout for that array that satisfies the majority of the nests, taking their respective costs into account. Essentially, our global layout determination algorithm attempts to resolve the global impact of data transformations locally by using iteration space transformations. For a different and more comprehensive treatment of the global layout determination problem, we refer the reader to [21].

7.1 Optimization with Constrained Layouts

We first focus on the problem of optimizing locality when some or all the file layouts are fixed. We note that each fixed file layout requires the innermost loop index to be in the appropriate array index position (dimension), depending on the file layout of the array. For example, suppose that the file layout for an 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_1th dimension of this array. If this is not possible, then it should try to place j_n only to the k_2th dimension and so on. If all the dimensions up to and including k_h are tried unsuccessfully,

TABLE 6
Results on Scalabilty of Optimized Versions on the `large` Data Set

Nest	Version	No. 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
	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
	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
	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
	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
	Opt	13.9	25.1	46.2	98.1
	H-opt	13.1	24.6	44.3	93.1

Nest	Version	No. 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
	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
	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
	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
	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
	Opt	15.4	30.9	60.2	113.0
	H-Opt	15.4	30.9	60.2	113.0

then j_{n-1} should be tried for the k_1th dimension and so on. As we will show shortly, this *constrained layout* algorithm is very important for global optimization.

7.2 A Heuristic for Global Optimization Problem

Our approach to the global optimization problem is based on the concept of the *most costly nest*. Intuitively, this is the nest that would take the most I/O time and should be optimized. A programmer can use *compiler directives* to give hints about this nest. We can also use a metric such as the product of the number of loops and the number of arrays referenced in the nest. The nest that has the largest resulting value can be marked as the most costly nest. Currently, we use profile information to order the nests according to a cost criterion. The rest of the algorithm is independent of how the most costly nest is determined. The algorithm proceeds as follows: First, the most costly nest is optimized by using the algorithm presented in Fig. 6. After this step, the file layouts for some of the out-of-core arrays will be determined. Then, each of the remaining nests can be optimized using the approach presented for the *constrained layout* case in Section 7.1. After each nest is optimized, new file layout constraints will be obtained and these will be propagated for the optimization of the remaining nests.

As an example that illustrates the working of the algorithm, we consider the program shown below.

$$
\begin{array}{l}
DOi = 1, n \\
\quad DOj = 1, n \\
\quad\quad A(i,j) = B(j,i) + C(i,j) \\
\quad END\ DO \\
END\ DO \\
DOi = 1, n \\
\quad DOj = 1, n \\
\quad\quad C(j,i) = D(j,i) + A(j,i) \\
\quad END\ DO \\
END\ DO \\
DOi = 1, n \\
\quad DOj = 1, n \\
\quad\quad D(j,i) = E(i,j) + F(i,j) \\
\quad END\ DO \\
END\ DO
\end{array}
$$

This code accesses six out-of-core arrays using different access patterns. An informal description of how our approach works follows: Without loss of generality, we assume that the first nest is the most costly nest and the last nest the least costly. We also assume that we optimize this code for the single processor case.[4] For the first nest, we use

4. In the multiple processor case, we need to propagate the distribution information (in addition to the layout information) across the nests. In fact, the automatic file layout determination problem is very similar to the automatic data decomposition problem [24], [15] and we believe that it can be attacked using similar techniques.

TABLE 7
Experimental Results on 16 Nodes of an Intel Paragon

Program	Col	Row	L-Opt	D-Opt	Opt	H-Opt
vpenta	683.12	54.4	100.0	54.4	54.4	33.6
btrix	495.05	69.0	100.0	66.2	66.2	47.0

TABLE 8
Experimental Results on 16 Nodes of an IBM SP-2

Program	Col	Row	L-Opt	D-Opt	Opt	H-Opt
vpenta	128.31	57.0	100.0	63.7	55.0	40.5
btrix	99.15	82.4	100.0	78.2	61.0	48.1

our approach (without any constraints) and decide that no loop transformation is necessary and that the file layouts of arrays *A*, *B*, and *C* should be row-major, column-major, and row-major, respectively. Having optimized the first nest, we move to the second one and, in optimizing this nest, we take the layouts determined so far into account. Since layouts of *A* and *C* are already set to row-major, in order to satisfy these constraints, our approach applies loop interchange to the nest. This new loop order, in turn, leads to a row-major file layout for array *D*. Finally, we come to the third nest and apply loop interchange again to satisfy the layout requirements of *D*. Finally, this new loop order causes our algorithm to assign column-major file layouts for arrays *E* and *F*.

The results of this global optimization algorithm on a 2-D out-of-core FFT code (that uses out-of-core transpose loops) demonstrates a 17-26 percent reduction in I/O times using our heuristic [20]. In general, the global layout optimization improves the scalability of the program as well as the execution time.

In order to perform an evaluation of this global optimization technique on the whole programs, we hand-applied it using two representative codes from our experimental suite. Table 7 presents the results on 16 processors of the Paragon and Table 8 presents the results on 16 processors of the SP-2. As before, for each data set, the `Col` column gives the total execution time of the out-of-core program in *seconds*, whereas the other columns show the respective execution times for the other versions as a fraction of that of `Col`. These results are encouraging and indicate that our global optimization scheme has good potential.

8 Related Work

In this section, we briefly discuss the related work on out-of-core compilation and locality optimization techniques.

8.1 Related Work on Out-of-Core Compilation

Several compiler methods for out-of-core HPF programs are presented in [6] and [5]. In Bordawekar et al. [6], a technique for optimizing communication for out-of-core stencil problems is discussed. They show that the extra file I/O originating from communication requirements can completely be removed for some special cases.

The group at Rice University [34] incorporates out-of-core compilation techniques with Fortran D. The main philosophy behind their approach is to choreograph the I/O from disks along with the corresponding computation. They group computations into "deferred routines" which are computations to data that fall into elements that are close physically and, hence, can be computed upon when the I/O for that contiguous chunk is performed. Their compilation phases are: program analysis, I/O insertion and optimization, and parallelization and communication optimization. Our file locality optimization algorithms discussed in this paper are general in the sense that they can be embedded in any out-of-core compilation framework such as [6], [5], or [34].

Abu-Sufah et al. [1] dealt with optimizations to enhance the locality properties of programs in a virtual memory environment. They especially evaluated the gains obtained from tiling and loop fusion. In contrast, we consider dimension-wise layout transformations and use explicit file I/O.

ViC* (Virtual C*) [11] is a preprocessor which transforms a C* program that uses out-of-core data structures into a program with in-core data structures with appropriate library calls from ViC* library that read/write data from/to disks into the in-core data structures. It uses virtual memory to implement the accesses to out-of-core structures. In principle, our file 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 and by omitting the memory allocation part). None of the previous work on out-of-core compilation, to our knowledge, considered the compiler-directed file layout transformations.

Another compiler-directed optimization, prefetching, is used by Mowry et al. [32] for optimizing out-of-core programs. We believe that the compiler-directed prefetch is complementary 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.

8.2 Related Work on Locality Optimizations

Iteration space tiling has been used for optimizing cache locality in several papers [27], [44]. McKinley et al. [31] propose an optimization technique consisting of loop permutation, loop fusion, and loop distribution. The assumption of a fixed layout strategy prevents some array references from getting optimized, as shown earlier in this paper, and that in turn may cause a substantial performance loss. This is the main reason that we believe that, in optimizing out-of-core computations, the compiler must consider both data space and iteration space transformations. After the I/O is optimized by our techniques, however, we strongly recommend the use of an in-core locality optimization technique for data tiles in memory.

In Cierniak and Li [9], a unified approach to locality optimization that 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 [9] because of the following reasons:

- The approach given in [9] depends on a *stride vector* whose value should be guessed by the compiler beforehand. Our approach does not have such a requirement.
- Our approach does not restrict the search space of possible loop transformations, whereas the approach in [9] does.
- Our extension to multiple nests (global I/O optimization—Section 7) is simpler than the one offered by [9] for global optimization.

Also, their paper does not present any heuristic for choosing the stride vector (which determines the layout).

Anderson et al. [3] propose a data transformation technique for distributed shared memory machines. By using two types of data transformations (strip-mining and permutation), they try to make the data accessed by the same processor contiguous in the shared address space. Their algorithm inherits parallelism decisions made by a previous phase of the SUIF compiler [42]; so, in a sense, is not directly comparable to our approach which attempts to derive a transformation matrix suitable for both locality and parallelism.

O'Boyle and Knijnenburg [33] consider a unifying framework for nonsingular data transformations in order to optimize in-core computations. They present validity conditions and describe constructive algorithms to generate desired transformations. Unlike our approach, their technique is based on the transformation of the data space alone. Since, in out-of-core computations, the main data structures reside on files stored on disks, their approach, which involves skewing of data, is too costly for an out-of-core compilation framework.

Finally, previous work on parallelism has concentrated, among other topics, on compilation techniques for multicomputers [17], automatic discovery of parallelism [43], [15], [24], and data layout reorganizations [9], [3], [2].

9 Summary and Conclusions

In this paper, we proposed algorithms for 1) optimizing locality, 2) optimizing parallelism and communication, and 3) optimizing locality, parallelism, and communication together. Our techniques can reduce the execution times by as much as an order of magnitude on the IBM SP-2 and the Intel Paragon. The results, however, should not be interpreted as a general comparison of the two machines as they are dependent on the parallel file systems, our parallel I/O library that we used, and the I/O access pattern of the loop nests in our experiment suite. We believe that our work is unique in the sense that it combines data transformations (file layout optimizations) and control transformations (loop permutations) in a unified framework for optimizing *out-of-core programs* on distributed-memory message-passing machines. We have shown in this paper that the combination of these two transformations leads to optimized communication and optimized file layouts and that, in turn, minimizes the overall execution time.

Acknowledgments

This work was supported in part by U.S. National Science Foundation (NSF) Young Investigator Award CCR-9357840, NSF grant CCR-9509143, and NSF Young Investigator Award CCR-9457768 and NSF grant CCR-9210422.

References

[1] W. Abu-Sufah, D. Kuck, and D. Lawrie, "On the Performance Enhancement of Paging Systems Through Program Analysis and Transformations," *IEEE Trans. Computers,* vol. 30, no. 5, pp. 341–355, May 1981.

[2] J. Anderson and M. Lam, "Global Optimizations for Parallelism and Locality on Scalable Parallel Machines," *Proc. SIGPLAN '93 Conf. Programming Language Design and Implementation,* June 1993.

[3] J. Anderson, S. Amarasinghe, and M. Lam, "Data and Computation Transformations for Multiprocessors," *Proc. Fifth ACM SIGPLAN Symp. Principles and Practice of Parallel Programming,* July 1995.

[4] A.J.C. Bik and H.A.G. Wijshoff, "On a Completion Method for Unimodular Matrices," Technical Report 94-14, Dept. of Computer Science, Leiden Univ., The Netherlands, 1994.

[5] R. Bordawekar, A. Choudhary, K. Kennedy, C. Koelbel, and M. Paleczny, "A Model and Compilation Strategy for Out-of-Core Data-Parallel Programs," *Proc. Fifth ACM Symp. Principles and Practice of Parallel Programming,* July 1995.

[6] R. Bordawekar, A. Choudhary, and J. Ramanujam, "Automatic Optimization of Communication in Out-of-Core Stencil Codes," *Proc. 10th ACM Int'l Conf. Supercomputing,* pp. 366–373, May 1996.

[7] R. Bordawekar, "Techniques for Compiling I/O Intensive Parallel Programs," PhD thesis, Dept. of Electrical and Computer Eng., Syracuse Univ., Apr. 1996.

[8] L. Carter, J. Ferrante, S.F. Hummel, B. Alpern, and K. Gatlin, "Hierarchical Tiling: A Methodology for High Performance," Technical Report CS96–508, Univ. of California at San Diego, Nov. 1996.

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

[10] S. Coleman and K. McKinley, "Tile Size Selection Using Cache Organization and Data Layout," *Proc. SIGPLAN '95 Conf. Programming Language Design and Implementation,* June 1995.

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

[12] I. Couvertier-Reyes, "Automatic Data and Computation Mapping for Distributed Memory Machines," PhD thesis, Dept. of Electrical and Computer Eng., Louisiana State Univ., May 1996.

[13] J. Dongarra, J. DuCroz, I. Duff, and S. Hammerling, "A Set of Level 3 Basic Linear Algebra Subprograms," *ACM Trans. Math. Software,* vol. 16, pp. 1–17, 1990.

[14] J. Ferrante, V. Sarkar, and W. Thrash, "On Estimating and Enhancing Cache Effectiveness," *Proc. Languages and Compilers for Parallel Computing (LCPC '91),* pp. 328–343, 1991.

[15] M. Gupta and P. Banerjee, "Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compilers on Multicomputers," *IEEE Trans. Parallel and Distributed Systems,* vol. 3, no. 2, pp. 179–193, Mar. 1992.

[16] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach.* San Mateo, Cailf.: Morgan Kaufmann, 1990.

[17] S. Hiranandani, K. Kennedy, and C.-W. Tseng, "Compiling Fortran D for MIMD Distributed Memory Machines," *Comm. ACM,* vol. 35, no. 8, pp. 66–88, Aug. 1992.

[18] F. Irigoin and R. Triolet, "Supernode Partitioning," *Proc. 15th Ann. ACM Symp. Principles of Programming Languages,* pp. 319–329, Jan. 1988.

[19] M. Kandemir, R. Bordawekar, and A. Choudhary, "Data Access Reorganizations in Compiling Out-of-Core Data Parallel Programs on Distributed Memory Machines," *Proc. Int'l Parallel Processing Symp.,* pp. 559–564, Apr. 1997.

[20] M. Kandemir, A. Choudhary, J. Ramanujam, and M. Kandaswamy, "A Unified Compiler Algorithm for Optimizing Locality, Parallelism and Communication in Out-of-Core Computations," *Proc. Workshop I/O in Parallel and Distributed Systems,* Nov. 1997.

[21] M. Kandemir, M. Kandaswamy, and A. Choudhary, "Global I/O Optimizations for Out-of-Core Computations," *Proc. High-Performance Computing Conf.,* Dec. 1997.
[22] M. Kandemir, J. Ramanujam, and A. Choudhary, "Improving the Performance of Out-of-Core Computations," *Proc. 1997 Int'l Conf. Parallel Processing,* pp. 128–136, Aug. 1997.
[23] I. Kodukula, N. Ahmed, and K. Pingali, "Data-Centric Multilevel Blocking," *Proc. Programming Language Design and Implementation,* June 1997.
[24] U. Kremer, "Automatic Data Layout for Distributed Memory Machines," PhD thesis, Dept. of Computer Science, Rice Univ., Houston, Tex., 1995.
[25] M. Lam, E. Rothberg, and M. Wolf, "The Cache Performance of Blocked Algorithms," *Proc. Fourth Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* Apr. 1991.
[26] S. Leung and J. Zahorjan, "Optimizing Data Locality by Array Restructuring," Technical Report 95-09-01, Computer Science and Engineering Dept., Univ., of Washington, Sept. 1995.
[27] W. Li, "Compiling for NUMA Parallel Machines," PhD thesis, Cornell Univ. 1993.
[28] W. Li and K. Pingali, "Access Normalization: Loop Restructuring for NUMA Compilers," *ACM Trans. Computer Systems,* Nov. 1993.
[29] M. Mace, *Memory Storage Patterns in Parallel Processing.* Boston: Kluwer Academic, 1987.
[30] A. McKellar, E. Coffman, "The Organization of Matrices and Matrix Operations in a Paged Multiprogramming Environment," *Comm. ACM,* vol. 12, no. 3, pp. 153–165, Mar. 1969.
[31] K. McKinley, S. Carr, and C.W. Tseng, "Improving Data Locality with Loop Transformations," *ACM Trans. Programming Languages and Systems,* July 1996.
[32] T. Mowry, A. Demke, and O. Krieger, "Automatic Compiler-Inserted I/O Prefetching for Out-of-Core Applications," *Proc. Second Symp. Operating Systems Design and Implementations,* pp. 3–17, Oct. 1996.
[33] M.F.P. O'Boyle and P.M.W. Knijnenburg, "Non-Singular Data Transformations: Definition, Validity, Applications," *Proc. Sixth Workshop Compilers for Parallel Computers,* 1996.
[34] M. Paleczny, K. Kennedy, and C. Koelbel, "Compiler Support for Out-of-Core Arrays on Parallel Machines," CRPC Technical Report 94509-S, Rice Univ., Houston, Tex., Dec. 1994.
[35] J. Ramanujam and A. Narayan, "Integrating Data Distribution and Loop Transformations for Distributed Memory Machines," *Proc. Seventh SIAM Conf. Parallel Processing for Scientific Computing,* D. Bailey et al., eds., pp. 668–673, Feb. 1995.
[36] J. Ramanujam and A. Narayan, "Automatic Data Mapping and Program Transformations," *Proc. Workshop Automatic Data Layout and Performance Prediction,* Apr. 1995.
[37] B. Rullman, *Paragon Parallel File System,* External Product Specification, Intel Supercomputer Systems Division.
[38] R. Thakur, R. Bordawekar, A. Choudhary, R. Ponnusamy, and T. Singh, "PASSION Runtime Library for Parallel I/O," *Proc. Scalable Parallel Libraries Conf.,* pp. 119–128, Oct. 1994.
[39] R. Thakur, R. Bordawekar, and A. Choudhary, "Compilation of Out-of-Core Data Parallel Programs for Distributed Memory Machines," *Proc. Workshop I/O in Parallel Computer Systems at IPPS '94,* Apr. 1994.
[40] E. Torrie, C-W. Tseng, M. Martonosi, and M. Hall, "Evaluating the Impact of Advanced Memory Systems on Compiler-Parallelized Codes," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques,* June 1995.
[41] K. Trivedi, "On the Paging Performance of Array Algorithms," *IEEE Trans. Computers,* vol. 26, no. 10, pp. 938–947, Oct. 1977.
[42] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S.K. Tjiang, S.-W. Liao, C.-W. Tseng, M. Hall, M. Lam, and J. Hennessy, "SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers," *ACM SIGPLAN Notices,* vol. 29, no. 12, pp. 31–37, Dec. 1994.
[43] M. Wolf and M. Lam, "A Loop Transformation Theory and an Algorithm to Maximize Parallelism," *IEEE Trans. Parallel and Distributed Systems,* vol. 2, no. 4, pp. 452–471, Oct. 1991.
[44] M. Wolf and M. Lam, "A Data Locality Optimizing Algorithm," *Proc. ACM SIGPLAN '91 Conf. Programming Language Design and Implementation,* pp. 30–44, June 1991.
[45] M. Wolfe, *High Performance Compilers for Parallel Computers.* Addison-Wesley, 1996.

Mahmut Kandemir received the BSc and MSc degree in control and computer engineering from Istanbul Technical University, Istanbul, Turkey, in 1988 and 1992, respectively. He received the PhD degree from Syracuse University, Syracuse, New York, in electrical engineering and computer science, in 1999. He has been an assistant professor in the Computer Science and Engineering Department at the Pennsylvania State University since August 1999. His main research interests are optimizing compilers, I/O intensive applications, and power-aware computing. He is a member of the IEEE.

Alok Choudhary received the BE (Hons.) degree from the Birla Institute of Technology and Science, Pilani, India, in 1982. He received the MS degree from the University of Massachusetts, Amherst, in 1986, and the PhD degree from the University of Illinois, Urbana-Champaign, in electrical and computer engineering in 1989. He has been an associate professor in the Electrical and Computer Engineering Department at Northwestern University since September, 1996. From 1993 to 1996, he was an associate professor in the Electrical and Computer Engineering Department at Syracuse University and, from 1989 to 1993, he was an assistant professor in the same department. Prior to 1984, he worked in industry as a computer consultant.

Dr. Choudhary received the U.S. National Science Foundation's Young Investigator Award in 1993 (1993-1999). He has also received an IEEE Engineering Foundation award, an IBM Faculty Development award, and an Intel Research Council award. His main research interests are in high-performance computing and communication systems and their applications in many domains including multimedia systems, information processing, and scientific computing. In particular, his interests lie in the design and evaluation of architectures and software systems (from system software such as runtime systems, compilers, and programming languages to applications), high-performance servers, high-performance databases, and input-output. He has published more than 130 papers in various journals and conferences in the these areas. He has also written a book and several book chapters on the these topics. His research has been sponsored by (past and present) DARPA, the National Science Foundation, NASA, AFOSR, ONR, DOE, Intel, IBM, and TI.

Dr. Choudhary served as the conference cochair for the International Conference on Parallel Processing and is currently the chair of the International Workshop on I/O Systems in Parallel and Distributed Systems. He is an editor for the *Journal of Parallel and Distributed Computing* and has served as a guest editor for *Computer* and *IEEE Parallel and Distributed Technology*. He serves (or has served) on the program committees of many international conferences in architectures, parallel computing, multimedia systems, performance evaluation, distributed computing, etc. He also serves on the High-Performance Fortran Forum, a forum of academia, industry, and government labs working on standardizing programming languages for portable programming on parallel computers. He is a member of the IEEE, the IEEE Computer Society, and the ACM.

J. Ramanujam received the BTech degree in electrical engineering from the Indian Institute of Technology, Madras, in 1983, and the MS and PhD degrees in computer science from Ohio State University, Columbus, in 1987 and 1990, respectively. He is currently an associate professor of electrical and computer engineering at Louisiana State University, Baton Rouge. His research interests are in compilers for high-performance computer systems, program transformations, embedded systems, high-level hardware synthesis, parallel input/output systems, parallel architectures, and algorithms. He has published more than 80 papers in journals and conferences in these areas.

Dr. Ramanujam received the U.S. National Science Foundation's Young Investigator Award in 1994. He has served on the program committees of several conferences such as the International Conference on Parallel Architectures and Compilation Techniques (PACT 2000), the International Symposium on High Performance Computing (HiPC 99), and the 1997 International Conference on Parallel Processing. He has taught tutorials on compilers for high-performance computers at several conferences such as the International Conference on Parallel Processing (1998, 1996), Supercomputing 94, Scalable High-Performance Computing Conference (SHPCC 94), and the International Symposium on Computer Architecture (1993 and 1994).

Meenakshi A. Kandaswamy received the BE degree (Honors) from the Regional Engineering College, Trichy, India, in 1989, the MS degree in computer science, and the PhD degree in computer science from Syracuse University in 1995 and 1998, respectively. She is a senior systems engineer at the Server Architecture Lab of Intel Corporation, Hillsboro, Oregon. Her research interests include high performance I/O, parallel applications, server performance analysis, multiprocessor file systems, and memory hierarchies.