A Unified Compiler Algorithm for Optimizing Locality, Parallelism and Communication in Out-of-Core Computations*

M. Kandemir[†] A. Choudhary[‡] J. Ramanujam[§] M. Kandaswamy[¶]

Abstract

This paper presents compiler algorithms to optimize out-of-core programs. These algorithms consider loop and data layout transformations in a unified framework. The performance of an out-of-core loop nest containing many references can be improved by a combination of restructuring the loops and file layouts. This 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 tiling loop. Preliminary results from hand-compiles on IBM SP-2 and Intel Paragon show that this approach reduces the execution time, improves the bandwidth speedup and overall speedup. In addition, we extend the base algorithm to work with file layout constraints and show how it can be used for optimizing programs consisting of multiple loop nests.

1 Introduction

In recent years, processor speed has become significantly higher than both memory and disk speeds. 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. A computation which 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*. In contrast a computation which operates on data sets in memory is called *in-core*. For out-of-core problems where the sizes far exceed the size of the available memory, it is particularly important. An optimizing compiler for parallel out-of-core programs faces several challenges: (1) the granularity of parallelism should be maximized to reduce synchronization and communication overhead; (2) communication should be optimized; and (3) since I/O accesses are generally several orders of magnitude slower than communication, I/O should be optimized.

The difficulty of optimizing out-of-core programs originates from the following factor: the issues of optimizing I/O, optimizing parallelism and minimizing communication are inter-related. For example, for a given loop nest where a number of out-of-core arrays are accessed, the I/O optimizations may imply a preferred order for the loops whereas the parallelism optimizations may suggest another. In this paper we offer a *unified* strategy to optimize out-of-core programs for locality, parallelism and communication. Specifically our optimizations (1) maximize the granularity of parallelism by transforming the loop nest such that the outermost loop can run 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 [26]); (2) vectorize communication, i.e., perform the communication in large chunks of data; and (3) 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.

The paper is organized as follows. In Section 2, we review the basic concepts in file layouts and tiling. In Sections 3 and 4, we present an automatic method by which file locality and communication can be optimized, respectively. Section 5 presents a unified algorithm which (1) maximizes granularity of parallelism, (2) minimizes communication, and (3) optimizes I/O. Section 6 presents experimental results which demonstrate the efficacy of the algorithm. Section 7 extends the base algorithm to handle the multiple-loop-nest case. Section 8 presents related work, and Section 9 concludes the paper.

2 Preliminaries

2.1 Data Storage Model

We build our compiler optimizations upon a storage subsystem model, called the *local placement model* (LPM) [6], which 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 platform to experiment with. Under the 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 LPM, portions of local out-of-core arrays, called data tiles, are fetched and stored in local memory. The data sharing is performed by explicit message passing, so this system is a natural extension of the distributed-memory

*A. Choudhary was supported in part by NSF Young Investigator Award CCR-9357840 and NSF CCR-9509143. The work of J. Ramanujam was supported in part by an NSF Young Investigator Award CCR-9457768.

[†]Department of Electrical Engineering and Computer Science, Syracuse University, Syracuse, NY 13244. mtk@ece.nwu.edu

[‡]Department of Electrical and Computer Engineering, Northwestern University, Evanston, IL 60208. choudhar@ece.nwu.edu

[§]Department of Electrical and Computer Engineering, Louisiana State University, Baton Rouge, LA 70803.jxr@ee.lsu.edu

[¶]Department of Electrical Engineering and Computer Science, Syracuse University, Syracuse, NY 13244. meena@ece.nwu.edu

paradigm.

2.2 File Layouts

The file layout for an h-dimensional out-of-core array can be in one of the $h!$ 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, the second dimension is the fastest changing dimension. If we think each element as a sub-matrix, that can also handle the blocked file layouts. In other words, the methods presented in this paper are applicable to blocked layout cases as well.

2.3 Optimized I/O Accesses and Tiling for Out-of-Core Computations

We refer to an array access as optimized if it can be performed such that all the data along a specific dimension will be read from a file incurring the least I/O cost. Consider the situation depicted in Figure 1 for three different cases using a two-dimensional array. The shaded portions denote data tiles. The case in Figure 1(a) 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 Figures 1(b) and Figures 1(c), on the other hand, correspond to the optimized accesses for row-major and column-major file layouts respectively. In Figure 1(b), with S_b I/O calls it is possible to read $S_b \times n$ elements from the file, and in Figure 1(c), $n \times S_c$ elements are read by issuing only S_c I/O calls (assuming for both the 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 the corresponding file layout. For example, reading $S_b \times n$ elements from the array shown in Figures 1(b) would cost n separate I/O calls if the array were stored in file as column-major. Second, in order to have a fair comparison we fix the available memory size (M) no matter how the array layouts are optimized. As an example, for the cases shown in Figure 1(a-c), assuming this is the only array referenced in the nest, the equality ${S_a}^2 = S_b n = n S_c = M$ 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. Accordingly, an I/O call in the program can correspond to several system calls to disk(s). The technique described in the rest of the paper attempts to decide optimal file layouts and to minimize the number of I/O calls to the files. Reduction in I/O calls to files lead, in general, to reduction in calls to disks. The relation, though, is system dependent and is not discussed in this paper.

Tiling is a technique to improve locality in in-core computations, and is a combination of strip-mining and loop permutation. When tiling is applied, it replaces the original loop with two new loops: a tiling loop and an element loop. The traditional tiling is an optional locality optimization technique and is applied to iteration spaces [29, 30]. In an out-of-core compilation strategy based on explicit file I/O, tiling of out-of-core *data* into memory is mandatory, and the compiler uses the results of dependence analysis [30] to determine whether or not tiling is legal. All necessary loop transformations should be performed in order to ensure the legality of tiling.

A naive approach can extend the compilation methodology of in-core programs for out-of-core computations by assuming user-defined data decomposition as follows: after the node program is determined, the loops are tiled and appropriate I/O calls are inserted between tiling loops. The available memory is divided evenly among the arrays involved. There are several drawbacks to this straightforward approach:

1. The program obtained by this method may not be able to exploit the largest granularity of parallelism;
2. Assuming a fixed file layout such as row-major or column-major for all arrays may adversely affect the performance [11]; 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 the best performance in out-of-core computations, both data (layout) and control (loop) transformations are necessary; and parallelism and locality should be handled in a unified way.* We note that our approach is based on explicit file I/O and is different from those presented in [1, 17, 19, 27].

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 for out-of-core arrays, and partitions the available memory among the data tiles of out-of-core arrays, all in a unified framework. Before discussing our algorithm, we present a quick overview of loop transformation theory.

3.1 Loop Transformation Theory

The algorithms presented in this paper rely on results from general loop transformation theory [15, 30]. We focus on loops where both array subscripts and loop bounds are affine functions of enclosing loop indices. A reference to an array X is represented by $X(\mathcal{L}\vec{I} + \vec{b})$ where $\mathcal{L}$ is a linear transformation matrix called the *array reference matrix*, $\vec{b}$ is the offset vector and $\vec{I}$ is a column vector representing the loop indices $i_1, i_2, \cdots, i_n$, starting from the outermost loop. For the rest of the paper, the reference matrix for array X will be denoted by $\mathcal{L}^X$ whereas the i^{th} row of $\mathcal{L}^X$ will be denoted by $\vec{\ell}_i^X$.

Linear mappings between iteration spaces of loop nests can be modeled by nonsingular matrices [15]. If $\vec{I}$ is the original iteration vector, after applying linear transformation T, the new iteration vector is $\vec{J} = T\vec{I}$. Similarly if $\vec{d}$ is the distance/direction vector, on applying T, $T\vec{d}$ is the new distance/direction vector. Since $\mathcal{L}\vec{I} = \mathcal{L}T^{-1}\vec{J}$, after the transformation $\mathcal{L}T^{-1}$ is the new array reference matrix. We

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

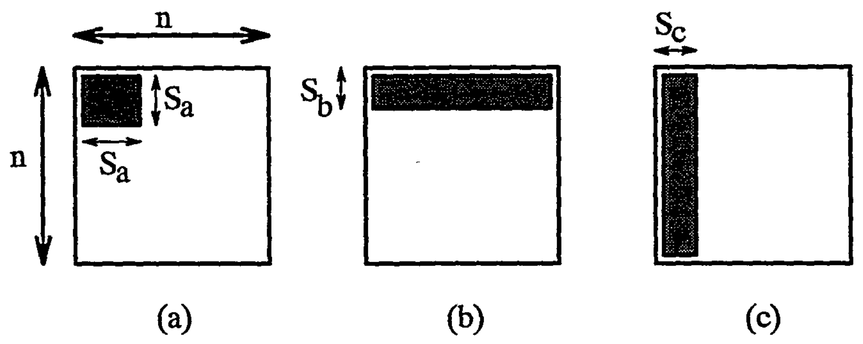


Figure 1: (a) Unoptimized access; (b)-(c) Optimized accesses.

denote T^{-1} by Q. An important characteristic of our approach is that using the array reference matrices, the entries of Q are derived systematically.

3.2 Explanation of the Algorithm

The algorithm for optimizing file locality is shown in Figure 2. Let $i_1, i_2, \cdots, i_n$ be the loop indices of the original nest, and $j_1, j_2, \cdots, j_n$ be the loop indices of the transformed nest, starting from the outermost loop. In the algorithm, C is the array reference on the LHS whereas A represents an array reference from the RHS. The symbol δ denotes the *don't care* condition. The algorithm works as follows.

Handling the LHS. The 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. In other words, after the transformation, the LHS array C should be of the form $C(*, \cdots, *, j_n, *, \cdots, *)$ where j_n (the new innermost loop index) is in the r^{th} dimension and $*$ indicates a term independent of j_n. This means that the r^{th} row of the transformed reference matrix for C is $(0, 0, \cdots, 0, 1)$ and all entries of the last column except the one in r^{th} row are zero. After that process the LHS array can be stored in the file such that the r^{th} dimension will be the fastest changing dimension to exploit the spatial locality in the file[2].

Handling the RHS's. The algorithm 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 tries to store this array on the file that the s^{th} dimension will be the fastest changing dimension.[3] 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 all loop indices are tried unsuccessfully, then the remaining entries of Q are set considering the data dependences and non-singularity. A modified version of the completion algorithm found in [16, 22, 23] is used for filling the remaining entries. Notice that our approach determines only the fastest changing dimension of the layout.

Choosing the best alternative. After a transformation and corresponding disk layouts are found, the next alternative for the LHS is tried and so on. Among all feasible solutions, the best one is chosen. Although several approaches can be taken 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 an alternative exploits the locality for the first reference in the outer loop and for the other references in the inner loop, the sum for this alternative is $1 + 2 + 2 = 5$.

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* 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 it sets the tile size for the fastest changing position to n (the array size and loop upper bound); otherwise it sets the tile size to S, a parameter whose value will be determined in the final step ($S \ll n$). The tile sizes for the remaining dimensions are also set to S. After all the tile sizes for all 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 the references to those 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 ap-

[2]Notice that 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 alternatives.

[3]Note that the presence of such a row s does not guarantee that the array will be stored on the file with the s^{th} dimension as the fastest changing dimension.

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 *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.

Figure 2: Algorithm for optimizing file locality in out-of-core computations.

pears 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 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 $3 \times n \times S + S \times S \leq M$. 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 the positive S values. Note that any inconsistency between the groups should be resolved by setting the tile size for the conflicting dimension(s) to S.

Important observations. First, **Steps 2** and **6** of Figure 2 involve solving integer matrix equations. Second, the algorithm considers all possible file layouts, of which the row-major and column-major layouts are only two alternatives. Third, the algorithm first optimizes the LHS array. This is important because of the fact that the data tiles for this array are both read and written. Finally, when the file layout of an out-of-core array is set to a specific form, memory layouts of its data tiles should also be set to the same form.

3.3 Example application of the algorithm

In this section we give an example to illustrate the algorithm shown in Figure 2. The same example will be used in the following two sections as well. Figure 3(a) shows a matrix-multiplication routine and Figure 3(b) presents a straightforward out-of-core translation for it. In Figure 3(b) only *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 Figure 3(b), $C[u, v]$ 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 Figure 3(c) corresponds to a data tile of size $n \times S$ (notice the loop bounds and steps). Unless stated otherwise, the word *loop* refers to tiling loop. For clarity all I/O statements between the tiling loops are omitted. The tile allocations for this naive translation are illustrated in Figure 3(e).

We now optimize the program shown in Figure 3(b) using the locality algorithm. Due to space limitations, we only show the successful trials. See [12] for details. 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 $L^C.Q = \begin{pmatrix} 0 & 0 & 1 \\ \delta & \delta & 0 \end{pmatrix}$, $q_{11} = q_{12} =$

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

(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]
  ENDDO w
 ENDDO v
 ENDDO u
```

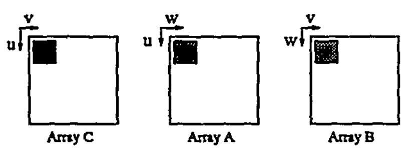
(b)

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

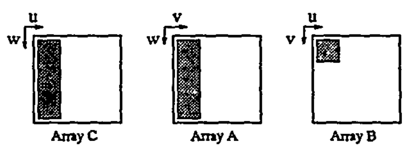
(c)

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

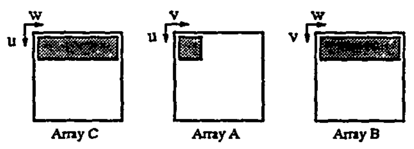
(d)



(e)



(f)



(g)

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

(h)

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

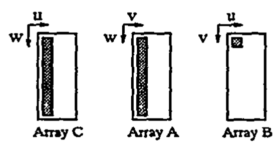
(i)

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

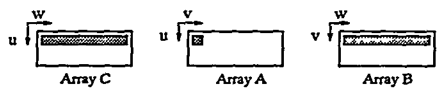
(j)

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

(k)



(l)



(m)

Figure 3: (a) Out-of-core matrix-multiplication nest; (b) Straightforward translation of (a); (c) I/O optimized translation of (a); (d) I/O optimized translation of (a); (e) Tile allocations for (b); (f) Tile allocations for (c); (g) Tile allocations for (d); (h) Parallelism optimized translation of (a); (i) Parallelism optimized translation of (a); (j) Parallelism and I/O optimized translation of (a); (k) Parallelism and I/O optimized translation of (a); (l) Tile allocations for (j); (m) Tile allocations for (k).

$q_{23} = 0$ 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 so far. Since $L^B.Q = \begin{pmatrix} \delta & 1 & 0 \\ \delta & 0 & 0 \end{pmatrix}$, $q_{22} = 0$ 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 Figure 3(c). All arrays 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 Figure 3(f). 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 $L^C.Q = \begin{pmatrix} \delta & \delta & 0 \\ 0 & 0 & 1 \end{pmatrix}$, $q_{13} = q_{21} = q_{22} = 0$ 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 Figure 3(d). All arrays are row-major. 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 Figure 3(g). The final memory constraint is $2nS + S^2 \leq M$, as before.[4]

4 Algorithm for Maximizing Parallelism and Minimizing Communication

This section presents an algorithm which 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 outermost transformed loop is distributed over the processors, (2) data decomposition across processors is determined for

[4]It should be emphasized that the parameter S in each optimized case is different, and that its value depends on the memory constraint.

each out-of-core array, and (3) communication is performed in large chunks and is optimized such that all non-local data are transferred to respective local disks before the execution of the innermost loop.

4.1 Explanation of Algorithm

As before, let $i_1, i_2, \cdots, i_n$ be the loop indices of the original loop and $j_1, j_2, \cdots, j_n$ be the loop indices of transformed loop. The following is the explanation of the algorithm:

Handling the LHS. The 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 r^{th} dimension. This means that the r^{th} 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 occurring. Note that distributing an out-of-core array means creating a corresponding local array file on each (logical) local disk.

Handling the RHSs. The algorithm works on one reference from RHS at a time. If a row s of data reference matrix for a 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 s^{th} dimension across processors without any communication.
If the condition above does not hold for a 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.

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.

As before, the transformation matrix should be nonsingular, and must satisfy data dependences. The algorithm is presented in Figure 4 and its details can be found in [24, 25].

4.2 Example

We reconsider the matrix-multiplication nest shown in Figure 3(b). 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 Figure 3(h). 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 Figure 3(i).

5 Unified Algorithm

This section presents a unified greedy algorithm which combines the characteristics of the algorithms presented in the previous two sections. It first attempts to optimize for parallelism, then for communication, and finally for locality.

5.1 Explanation of the Algorithm

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

- An array reference is said to be **optimized for parallelism** if the array can be distributed along an array dimension where only j_1 (the transformed outermost loop index) appears; thus, there is no communication.
- An array reference 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 α^{th} loop, where $1 \leq \alpha \leq \mu$. A reference optimized for parallelism is said to be degree 0 optimized for communication.
- An array reference 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) for each array reference where α and β denote the degree of communication and locality, respectively. The tuple $(0, 1)$ is the best possible tuple for a reference. Our algorithm tries to achieve this best possible tuple for all references. For those references 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 3-deep nest in which 2-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 across processors along 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.

Figure 4: Algorithm for data decomposition and parallelism.

Table 1: Array reference matrices for commonly used (α, β) tuples for a two-dimensional array enclosed in a three-deep loop nest (left) and in a four-deep loop nest (right).

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

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

dimension where j_1 occurs alone, and at the same time 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 is that in most of the nests, 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.

For an array reference, optimization for a tuple (α,β) can be formulated as a problem of finding a transformed reference matrix which 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 Figure 5. $\mathcal{L}^i$ denotes the original reference matrix for the i^{th} array in the nest, $i = 1$ corresponding to the LHS array. The j^{th} possible transformed reference matrix for an (α,β) tuple is denoted by $\mathcal{R}^j{}_{(\alpha,\beta)}$. This algorithm is a generic form of the previous algorithms in the sense that by setting $\mathcal{R}^j{}_{(\alpha,\beta)}$ matrices to appropriate values both of 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 $\mathcal{L}^A.Q = \begin{pmatrix} 0 & 0 & 1 \\ \delta & \delta & 0 \end{pmatrix}$ 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 Figure 3(j). All arrays are columnwise decomposed across 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 Figure 3(l). 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

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.

Figure 5: Unified algorithm for optimizing parallelism, communication and locality.

$\mathcal{L}^A.Q = \begin{pmatrix} 1 & 0 & 0 \\ \delta & 1 & 0 \end{pmatrix}$ 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 Figure 3(k). All 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 Figure 3(m).

6 Experimental Results

The experiments were performed on IBM SP-2 and Intel Paragon, for different values of *Slab Ratio* (SR), the ratio of available node memory to the total size of all out-of-core local arrays. The transformations to the original programs were applied manually following the algorithms.

6.1 Experimental Suite

Our experimental suite consists of a simple benchmark, four common kernels and an out-of-core 2-D FFT program.

6.1.1 A Simple Benchmark

Figure 8(a) shows a four-deep loop nest. Application of our algorithm results in two optimized node programs as shown in Figures 8(b) and (c) respectively. In Figure 8(b), 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 Figure 8(c) 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.

I/O Times Table 2 shows the I/O times on a single processor and in Figures 6 and 7 we present the normalized I/O times on SP-2 and Paragon respectively for four different versions of this example: (1) Original: The original program in Figure 8(a) is parallelized manually for maximum granularity and the data distributions are applied to files holding the out-of-core arrays. Fixed column-major layout is used and square tiles are read/written. (2) Col-Opt: The optimized program by using our approach under *fixed column-major layouts* for all arrays. 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 (Figure 8(b)). 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 three array accesses are optimized.

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.

Speedups Figure 9 shows the speedups for the Original and Opt versions on SP-2. Notice that each speedup is relative to the sequential version of the same program (Original or Opt).

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. Figure 10 shows two different cases for the example given in Figure 8(a): (1) 4K × 4K double arrays with a slab ratio of 1/64, and (2) 2K × 2K double arrays with a slab ratio of 1/256. Notice that each bandwidth speedup is relative to the I/O bandwidth of the same version on a single processor. Therefore 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

Table 2: I/O times (in seconds) for the example shown in Figure 8(a) on SP-2 and Paragon.

	IBM SP-2							
	$2K \times 2K$ 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	1441	1108	1074	441
1/64	4224	3111	2462	563	8384	6449	5912	1422
1/256	25087	21627	19298	1566	48880	35759	30055	4584

	Intel Paragon							
	$2K \times 2K$ double arrays				$4K \times 4K$ double arrays			
SR	Ori	Col	Row	Opt	Ori	Col	Row	Opt
1/4	1159	1050	988	208	5172	4431	4001	1273
1/16	2320	2141	1951	252	7360	5668	5412	1344
1/64	19201	14141	11760	576	28864	22002	20257	2304
1/256	99583	83848	76245	3028	192512	141370	117270	8193

Table 3: Number of Mbytes read and number of I/O calls issued (example shown in Figure 8(a)).

SR=1/4	$2K \times 2K$ double arrays				$4K \times 4K$ 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	28672	20480	14336	8192	57344	40960	28672	16384

is 6.23 MB/sec, whereas when p=16 the bandwidth is 55.3 MB/sec.

6.1.2 Common Kernels

We also applied our optimizations to a number of common kernels. The experiments were conducted on *four* nodes of SP-2 with $4K \times 4K$ two-dimensional and $4K$ one-dimensional double arrays and the results are shown in Figure 11 as normalized I/O times. Figure 11(a) gives the performance improvement obtained on a matrix-transpose loop that contains the statement $B(i,j) = A(j,i)$. The algorithm assigns column-major layout for array A and row-major layout for array B. It then allocates a tile of size nS to A, and a tile of size Sn to B. Notice that no fixed file layout for all arrays (as in C or Fortran) can obtain that performance. The performance improvement on an iterative-solver nest is given in Figure 11(b). The innermost loop contains the statement $Y(k) = Y(k) - U(k,j) * X(j)$. For this imperfectly nested example our algorithm associates column-major layout for all arrays. Figure 11(c) shows the results of the optimizations on a matrix-smoothing nest. The nest contains an outermost serial loop. Our algorithm associates row-major layout for all arrays (a fixed column-major layout for all arrays is also equally acceptable). Figure 11(d) illustrate the performance improvement on matrix-vector multiplication (Y=AX). It is interesting to note that in this example all layout combinations are equally good as far as I/O cost is concerned. Going with column-major layout for all arrays, the algorithm allocates a data tile of size nS for A, a tile of size n for Y, and a tile of size S for X.

6.1.3 2-D Out-of-Core FFT

Fast Fourier transform (FFT) is widely used in many areas such as digital signal processing, partial differential equation solutions and various other scientific and engineering problems. We implemented 2-D out-of-core FFT on the Intel Paragon. The 2-D out-of-core FFT consists of three steps : 1) 1-D out-of-core FFT, 2) Out-of-core transpose and 3) 1-D out-of-core FFT. The 1-D FFT steps consist of reading data from the two-dimensional out-of-core array and applying 1-D FFT on each of the columns. After this, the processed columns are written to file. In the transpose step, the out-of-core array is staged into memory, transposed and written to file. Our optimizing algorithm was applied to the original program (Orig) and the results are presented in Figure 12 for three different cases. For all cases, the I/O times constitute the bulk of the execution times. The bar charts in the figure show that there is a $17-26\%$ reduction in the overall I/O time when the program is optimized. The overhead time (Ovhd) includes times for file open and close operations, buffer copying and other times for communication initializations.

6.2 Observations

From these results we observe the following:

- The Opt version performs much better than all other versions.
- When the number of processors is increased, the effectiveness of our approach (Opt) increases (see Figures 6 and 7). This is because of the fact that more processors are now working on the out-of-core local array(s) I/O-optimally.

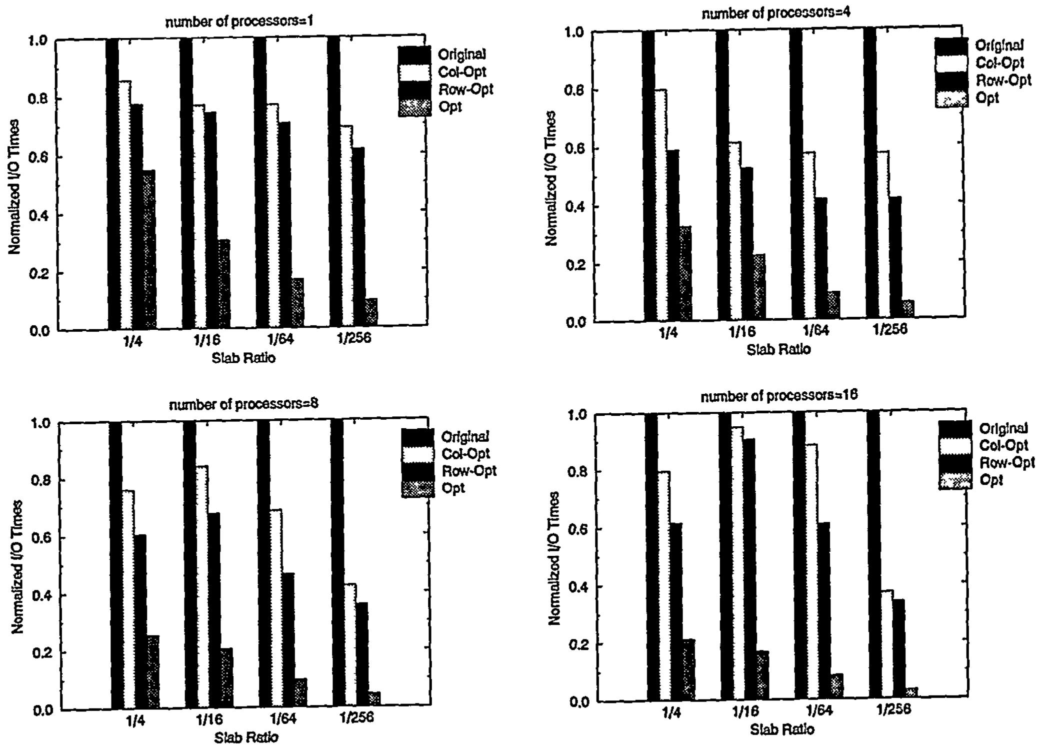


Figure 6: Normalized I/O times with $4K \times 4K$ (128 MByte) double arrays on SP-2.

- When the slab ratio is decreased, the effectiveness of our approach increases (see Figures 6, 7 and 11). As the amount of node memory is reduced, the Original version performs many number of small I/O requests, and that in turn degrades the performance dramatically.
- As shown in Figure 9, the Opt version also scales better than the Original for all slab ratios.
- As shown in Figure 10, the Opt version also has a better I/O bandwidth speedup than the Original.
- Demonstrations on two different platforms with varying compile-time and run-time parameters, such as number of processors, available memory, array sizes etc., prove that the algorithm is quite robust.
- The results presented here are conservative in the sense that the unoptimized programs are also parallelized such that the maximum granularity is obtained. Since this may not always be the case, the performance improvement obtained by our approach will be higher in general.

7 Global I/O Optimization

In this section we show how our algorithm can be extended to work on multiple nests. Since a number of out-of-core arrays can be accessed by a number of nests and each of these nests may require a different file layout for a specific array, the algorithm should determine a file layout for that array that satisfies the majority of the nests.

7.1 Constrained Layouts

We first focus on the problem of optimizing locality when some or all 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 form of the array. For example, suppose that the file layout for a h-dimensional array is such that the dimension k_1 is the fastest changing dimension, the dimension k_2 is the second fastest changing dimension, k_3 is the third etc. The algorithm should first try to place the new innermost loop index j_n only to the ${k_1}^{th}$ dimension of this array. If this is not possible, then it should try to place j_n only to the ${k_2}^{th}$ dimension and so on. If all dimensions up to and including k_h are tried unsuccessfully, then j_{n-1} should be tried for the ${k_1}^{th}$ dimension and so on. As we will show shortly this *constrained layout* algorithm is very important for global optimization.

7.2 A Simple Heuristic for Global Optimization

In the following we present sketch of a simple heuristic. Due to space limitations, the details are omitted (See [12] for details). Our approach is based on the concept of *most costly nest*. Intuitively, this is the nest which takes 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 multiplication of the number of loops and the number of arrays referenced in the nest. The nest which has the largest resulting value can be marked as the most costly nest. Then the algorithm proceeds as follows: First, the most costly nest is optimized by using the algorithm presented in Figure 5. After this step, file layouts for some of the out-of-core arrays will be determined. Then each of

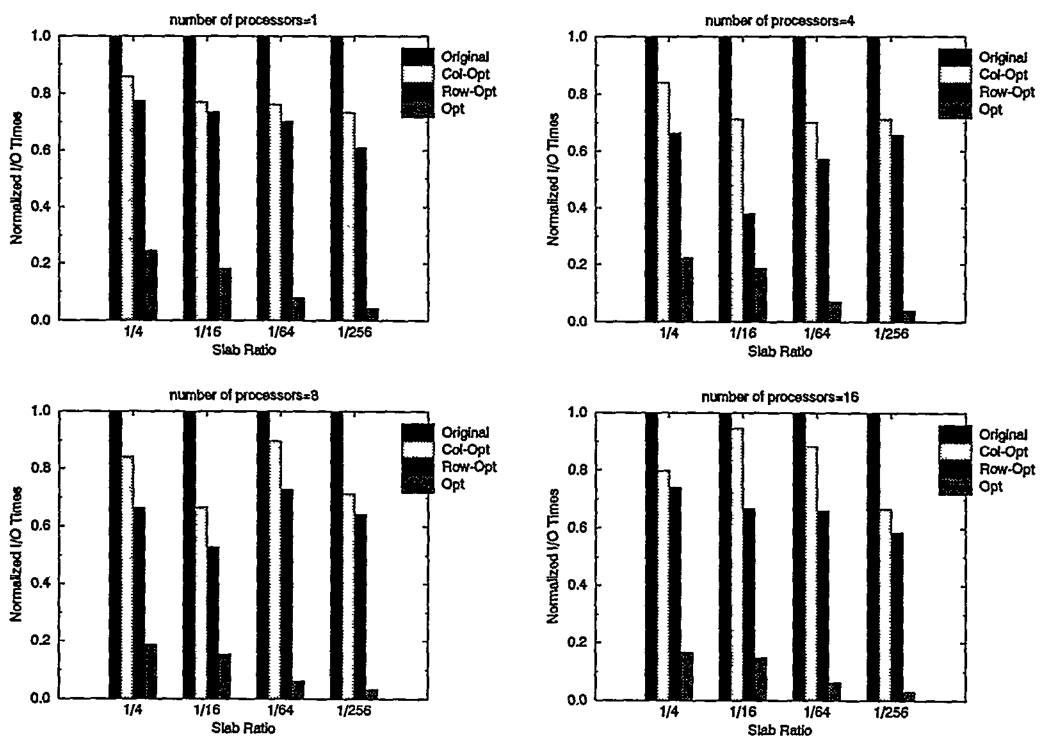


Figure 7: Normalized I/O times with $4K \times 4K$ (128 MByte) double arrays on Paragon.

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 optimization of the next nest.

Results on a 2-D out-of-core FFT code demonstrate a $17-26\%$ reduction in I/O times using our heuristic. Note that due to lack of space, we just summarize the results in this case. The 2-D out-of-core FFT consists of three steps: (a) 1-D out-of-core FFT,(b) out-of-core transpose, and (c) 1-D out-of-core FFT. The 1-D FFT steps consist of reading data from the two-dimensional out-of-core array and applying 1-D FFT on each of the columns. After this, the processed columns are written to the file. In the transpose step, the out-of-core array is staged into memory, transposed and written to the file. We also note that the effectiveness of the approach increases with increasing number of processors, decreasing slab ratio and increasing problem size. That is, the global layout optimization improves the scalability of the program as well as the execution time.

8 Related Work

Previous work on compiler optimizations to improve locality has concentrated on iteration space tiling. In [29] and [15], iteration space tiling is used for optimizing cache performance of in-core programs. In [18], a simple model for optimizing cache locality is developed. In [9], *uniformly generated sets* are introduced, and *windows* are used to capture data reuse. Previous work on parallelism has concentrated, among other topics, on compilation techniques for multicomputers [10], automatic discovery of parallelism [28] and data layout reorganizations [2, 3, 7].

In [8], the functionality of ViC*, a compiler-like preprocessor for out-of-core C* is described. Output of ViC* is a standard C* program with the appropriate I/O and library calls added for efficient access to out-of-core parallel variables. In [20], the compiler support for handling out-of-core arrays on parallel architectures is discussed. Bordawekar *et.al* [4] offer a strategy to compile out-of-core programs on distributed-memory message-passing systems. It should be noted that our algorithms are general in the sense that they can be incorporated to any out-of-core compilation framework for parallel and sequential machines.

Previous work considers optimizing the performance of virtual memory (VM). Abu-Sufah *et al.* [1], dealt with optimizations to enhance the locality properties of programs in a VM environment. 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). But, we believe that the impact of the I/O optimizations based on VM will be limited as compared to that of the optimizations based on the explicit file I/O. Because, (1) the fixed page sizes present a problem. Even if the computation requires a small portion of a data tile, a full page containing the data is brought into memory. Or, conversely, even if there is enough bandwidth for fetching a number of pages, the VMs generally bring one or two pages after every page fault, wasting the bandwidth; (2) the performance of the VM depends mostly on the page replacement policy of the operating system, which in turn, is out of control of the compiler. [19]; (3) the concept of locality in uniprocessors on which paging is based does not directly extend to parallel computers because of different interleaving of accesses by processors [21].

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

(a)

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

(b)

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

(c)

Figure 8: (a) A four-deep out-of-core loop nest; (b) Parallelism and I/O optimized translation of (a); (c) Parallelism and I/O optimized translation of (a).

9 Summary

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 time by as much as an order of magnitude on IBM SP-2 and 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 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 (layout determination) and control transformations 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 the highest granularity of parallelism and optimized file layouts, and that in turn, minimizes the overall execution time.

References

[1] W. Abu-Sufah. On the performance enhancement of paging systems through program analysis and transformations. *IEEE Transactions on Computers*, C-30(5), pages 341–355, May 1981.

[2] J. M. Anderson and M. S. Lam. Global Optimizations for Parallelism and Locality on Scalable Parallel Machines. In *Proc. SIGPLAN '93 Conference on Programming Language Design and Implementation (PLDI)*, June 1993.

[3] J. M. Anderson, S. P. Amarasinghe, and M. S. Lam. Data and computation transformations for multiprocessors. In *Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, July 1995.

[4] R. Bordawekar, A. Choudhary, K. Kennedy, C. Koelbel, and M. Paleczny. A model and compilation strategy for out-of-core data-parallel programs. In *Proc. 5th ACM Symposium on Principles and Practice of Parallel Programming*, July 1995.

[5] R. Bordawekar, A. Choudhary, and J. Ramanujam. Automatic optimization of communication in out-of-core stencil codes. In Proc. *10th ACM International Conference on Supercomputing*, May 1996.

[6] R. Bordawekar. *Techniques for compiling I/O intensive parallel programs*. Ph.D. Thesis, Dept. of Electrical and Computer Eng., Syracuse University, April 1996.

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

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

[9] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformations. *Journal of Parallel and Distributed Computing*, 5:587-616, 1988.

[10] S. Hiranandani, K. Kennedy, and C. -W. Tseng. Compiling Fortran D for MIMD distributed memory machines. *Communications of the ACM*, 35(8):66-88, August 1992.

[11] M. Kandemir, R. Bordawekar and A. Choudhary. Data access reorganizations in compiling out-of-core data parallel programs on distributed memory machines. In *Proc. International Parallel Processing Symposium*, April 1997.

[12] M. Kandemir, A. Choudhary, J. Ramanujam, and M. Kandaswamy. A unified compiler algorithm for optimizing locality, parallelism and communication in out-of-core computations. Technical Report, ECE Department, Northwestern University, August 1997.

[13] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, and M.Zosel. *High Performance Fortran Handbook*. The MIT Press, 1994.

[14] U. Kremer. *Automatic data layout for distributed memory machines*. Ph.D. thesis, Rice University, 1995.

[15] W. Li. *Compiling for NUMA parallel machines*. Ph.D. Thesis, Cornell University, 1993.

[16] W. Li and K. Pingali. Access normalization: Loop restructuring for NUMA compilers. *ACM Transactions on Computer Systems*, November 1993.

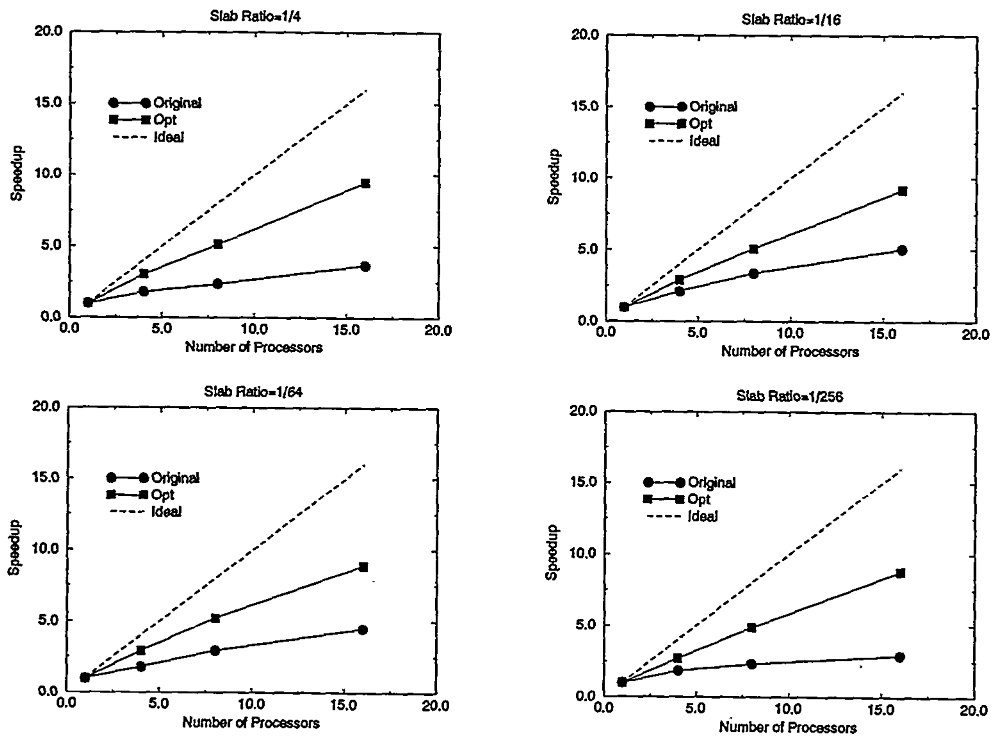


Figure 9: Speedups for unoptimized and optimized versions with $4K \times 4K$ double arrays on SP-2.

[17] A. C. McKellar, and E. G. Coffman. The organization of matrices and matrix operations in a paged multiprogramming environment. *Comm. ACM* 12, 3 (March 1969), pages 153-165.

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

[19] T. C. Mowry, A. K. Demke and O. Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *Proc. Second Symposium on Operating Systems Design and Implementations*, Seattle, WA, October 1996, pages 3-17.

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

[21] A. Purakayastha, C. S. Ellis, D. Kotz, N. Nieuwejaar, and M. Best. Characterizing parallel file-access patterns on a large-scale multiprocessor. In *Proceedings of the Ninth International Parallel Processing Symposium*, pages 165-172, April 1995.

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

[23] J. Ramanujam. Beyond unimodular transformations. *The Journal of Supercomputing*, 9(4):365–389, 1995.

[24] J. Ramanujam and A. Narayan. Integrating Data Distribution and Loop Transformations for Distributed Memory Machines. In *Proc. 7th SIAM Conference on Parallel Processing for Scientific Computing*, D. Bailey *et al.*, Eds., SIAM Press, pages 668–673, February 1995.

[25] J. Ramanujam and A. Narayan. Automatic data mapping and program transformations. In *Proc. Workshop on Automatic Data Layout and Performance Prediction*, Houston, TX, April, 1995.

[26] E. Torrie, C-W. Tseng, M. Martonosi, and M. W. Hall. Evaluating the impact of advanced memory systems on compiler-parallelized codes. In *Proc. International Conference on Parallel Architectures and Compilation Techniques*, June 1995.

[27] K. S. Trivedi. On the paging performance of array algorithms. *IEEE Transactions on Computers*, C-26(10):938-947, 1977.

[28] M. Wolf and M. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452-471, October 1991.

[29] M. Wolf and M. Lam. A data locality optimizing algorithm. In *Proc. ACM SIGPLAN 91 Conf. Programming Language Design and Implementation*, pages 30-44, June 1991.

[30] M. Wolfe. *High performance compilers for parallel computers*. Addison-Wesley Publishing Company, CA, 1996.

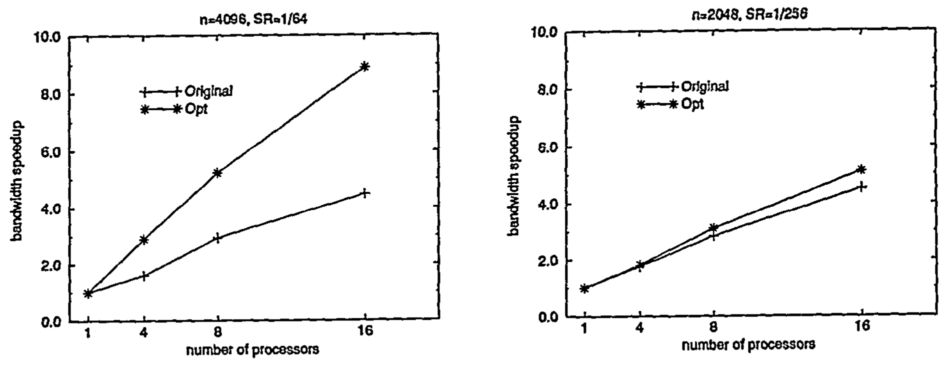


Figure 10: Bandwidth speedups for the example shown in Figure 8(a) on SP-2.

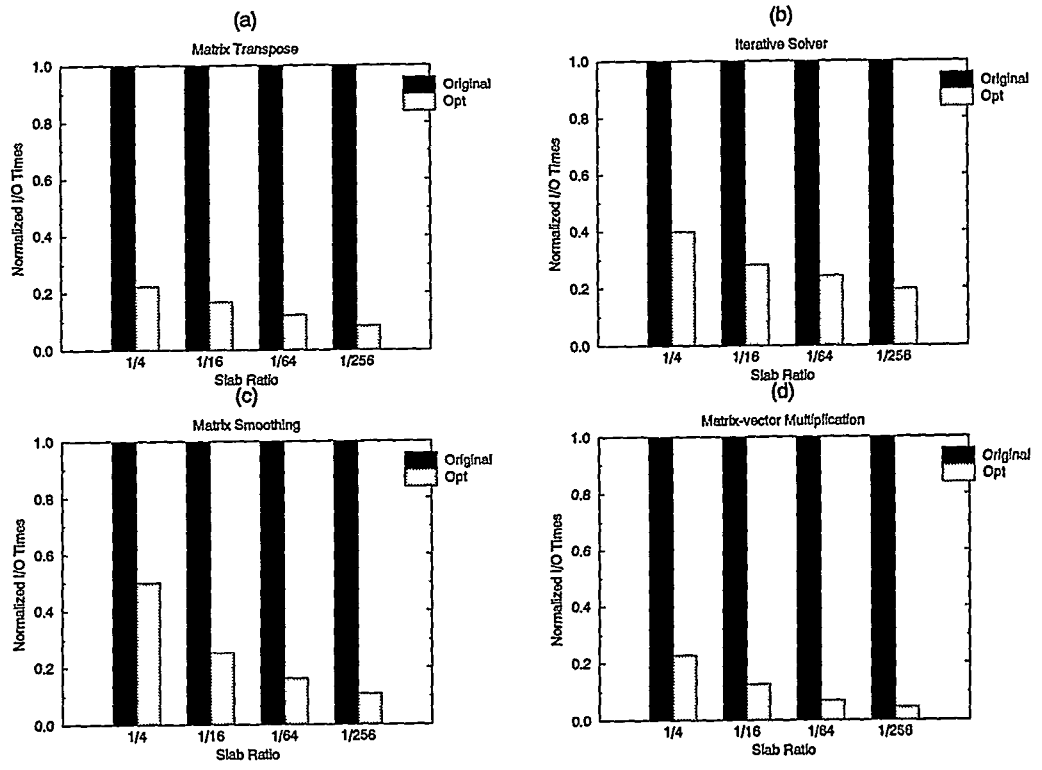


Figure 11: Normalized I/O times for different kernels: (a) Matrix transpose; (b) Iterative solver; (c) Matrix smoothing; (d) Matrix-vector multiplication.

2K by 2K, SR=1/4, p=4

	I/O	Comm	Comp	Ovhd
Orig	704.4	16.0	4.1	9.6
Opt	585.5	15.2	2.1	9.4

2K by 2K, SR=1/16, p=4

	I/O	Comm	Comp	Ovhd
Orig	2846.2	15.7	12.9	20.8
Opt	2109.8	16.8	8.4	22.7

2K by 2K, SR=1/4, p=8

	I/O	Comm	Comp	Ovhd
Orig	680.2	8.8	22.0	6.8
Opt	529.3	9.2	18.0	8.3

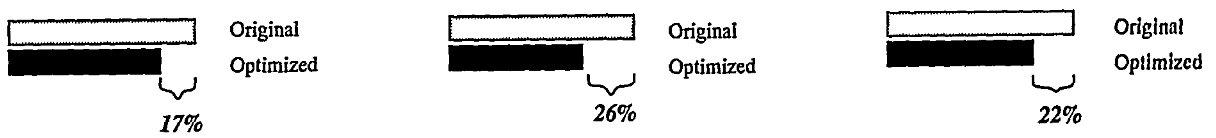


Figure 12: 2-D Out-of-core FFT: Reduction in I/O time and breakdown of execution time. *SR* is the slab ratio, and *p* is the number of processors. All times are in seconds.