JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 14, 107-138 (1998)

Locality Optimization Algorithms for Compilation of Out-of-Core Codes*

M. KANDEMIR[†], A. CHOUDHARY[‡], J. RAMANUJAM[§] AND M. KANDASWAMY[¶]
[†]*EECS Dept., Syracuse University*
Syracuse, NY 13244
E-mail: mtk@ece.nwu.edu
[‡]*ECE Dept., Northwestern University*
Evanston, IL 60208-3118
E-mail: choudhar@nwu.edu
[§]*ECE Dept., Louisiana State University*
Baton Rouge, LA 70803
E-mail: jxr@ee.lsu.edu
[¶]*EECS Dept., Syracuse university*
Syracuse, NY 13244
E-mail: meena@ece.nwu.edu

The difficulty of handling out-of-core data limits the performance of supercomputers as well as the potential of parallel machines. Since writing an efficient out-of-core version of a program is a difficult tasks, and since virtual memory systems do not perform well on scientific computations, we believe that there is a clear need for a compiler-directed explicit I/O approach for out-of-core computations. In this paper, we present a compiler algorithm to optimize the locality of disk accesses in out-of-core codes by choosing a good combination of file layouts on disks and loop transformations. The transformations change the access order of array data. Our solution goes beyond the single-loop level (called global I/O optimization in this paper) for out-of-core computations. Since the general problem is NP-complete, we present efficient heuristics that produce near-optimal solutions for several programs encountered in practice. Experimental results obtained on an IBM SP-2 and Intel Paragon provide encouraging evidence that our approach is successful at optimizing programs which depend on disk-resident data in distributed-memory machines.

Keywords: compilation techniques, out-of-core computations, parallel I/O, global I/O optimization, loop transformations.

Received April 30, 1997; revised November 20, 1997.
Communicated by Chau-Huang Huang.
* This work was supported in part by an NSF Young Investigator Award CCR-9357840, NSF CCR-9509143 and in part by the Scalable I/O Initiative, contract number DABT63-94-C-0049 from the Defense Advanced Research Projects Agency (DARPA), administered by the US Army at Fort Huachuca. The work of J. Ramanujam was supported in part by an NSF Young Investigator Award CCR-9457768, by an NSF grant CCR-9210422 and by the Louisiana Board of Regents through contract LEQSF (1991-94)-RD-A-09.

1. INTRODUCTION

Individual processor speeds are increasing at a very high rate both in the commercial and scientific arenas of the computing world and are barely able to keep up with the high demands of large-scale applications in these areas. In addition to being compute-intensive, these large-scale applications also store, retrieve and process huge quantities of data, which in turn increases the need for a powerful I/O subsystem. Unfortunately, advances in I/O subsystem technology have not kept pace with those of processors, hence leading to poor overall performance of I/O intensive applications. Database processing, climate prediction, computational chemistry codes, and computational physics codes all perform I/O intensive operations and render optimizations and tuning to the I/O subsystem a necessity. To achieve high I/O performance, in addition to architectural advances, appropriate operating systems support, parallel file system and run-time library support, and compiler support are critical. In compiler support, recognizing parallelism in I/O operations, performing appropriate loop/data transformations, and balancing computation, communication and I/O are particularly important. In this paper, we approach the I/O problem through an out-of-core compilation framework. Specifically, we present optimization algorithms to be used by compilers to bridge the gap between processor and I/O performance.

Although parallel file systems and run-time libraries also seem to be viable solutions to the I/O problem, it is known that run-time libraries and parallel file systems lack complete knowledge of the program's data structures, files and access patterns, unlike the compiler, and hence are at a disadvantage. Moreover, it is very difficult to use parallel file systems and run-time libraries as users must almost completely rewrite their applications to insert I/O calls. Also, such low-level I/O decisions made by programmers render the application less portable across machines with different I/O hardware/software [20]. At the compiler level, many different analysis have already been performed statically for in-core memory accesses, and we believe that they can be further extended and adapted to optimize out-of-core accesses. Out-of-core compilers can help a novice programmer who is not aware of the low-level details of the I/O system or the high-level details, such as locality, by making I/O optimization fully automatic. An adventurous programmer can still provide hints to the compiler if she wishes to, so that the compiler can better schedule I/O accesses and organize the interaction between computations, communication and the I/O.

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

memory machines and uniprocessors. Thirdly, we present global locality algorithms which go beyond the single-loop level and consider multiple loop nests.

This paper is organized as follows. In the next section, we briefly overview an out-of-core compilation framework. In In Section 3, we discuss the locality problem in out-of-core computations. In Section 4, we describe our loop-level file locality optimization algorithm, and a modified version of it, which makes use of file layout constraints. In Section 5, we present our global I/O optimization algorithm. We discuss related work in Section 6, and finally we present our conclusions in Section 7.

2. AN OUT-OF-CORE COMPILATION FRAMEWORK

In this section, we present a general framework which several out-of-core compilation approaches [3, 4, 8, 22] more or less adhere to for compiling I/O intensive applications. Some representative work that has been done in the area of out-of-core compilation techniques will be presented in Section 6.

An out-of-core compilation strategy might constitute two phases: In the *in-core phase*, the compiler performs lexical analysis, file/data distribution analysis, computation distribution and communication detection. All of these steps are performed in the global name space. The compiler assumes HPF-like distribution directives, by which the data are decomposed across the logical local disks of processors. In other words, the compiler directives determine the parts of the global data which will reside in each processor's local logical disk(s). After data placement, the computation is distributed using the *owner-computes* rule, where each processor computes only the values of data it owns [28]. Then the references in assignment statements are analyzed to detect the need for communication. The communication descriptors are computed for each reference, but communication optimizations are not performed in this phase.

The second phase is responsible for both detecting points where I/O statements should be inserted, and performing optimizations to handle communication and I/O in an optimized manner. The analyses in this phase are conducted in the local name space. After applying necessary loop transformations, the loops are tiled [27], and I/O points are detected. In principle, the compiler can apply the classical communication optimizations, such as message vectorization, message coalescing, message aggregation and collective communication, but the interaction between I/O and communication requirements should be taken into account as well. For example, it may not be a good idea to apply collective communication blindly without considering the file layout of data on disks. We refer the interested reader to Bordawekar *et al.* [3] for an in-depth discussion of the interaction between I/O and communication optimizations. Also in this second phase, file locality optimizations [14, 15] can be considered to increase the locality of accesses to files. Finally, low-level I/O optimizations can be performed to implement strategies such as two-phase I/O [24] and disk-directed I/O [16] to combine multiple small I/O requests into fewer large requests.

In the rest of the paper, we will discuss both loop-level and inter-loop file locality algorithms that can be applied in this second phase of out-of-core compila-

tion. Our techniques take into account access patterns and file layouts and can also be used for compilation of out-of-core codes in uniprocessors.

3. LOCALITY ISSUE IN OUT-OF-CORE COMPILATION

In this section, we discuss some problems in optimizing locality in out-of-core computations, and emphasize the importance of locality optimizations in an approach based on explicit file I/O.

3.1 Explicit File I/O vs. Virtual Memory

Traditionally, in scientific computations, I/O is handled in two different ways: virtual memory (VM) and explicit file I/O. Although VM ensures correctness for programs whose data sizes far exceed the size of available memory, it has been observed that the performance of scientific applications that rely on VM is generally poor due to frequent paging in and out of data [10]. We argue that the performance of I/O intensive programs based on VM will be limited as compared to that of the programs based on the explicit file I/O for following reasons.

1. The fixed page sizes present a problem. Even if the computation requires a small portion of a data file, 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 bandwidth.
2. The performance of VM depends mostly on the page replacement policy of the operating system, which in turn, is not under the control of the com0piler. In particular, even if a chunk of data will not be used any more, the replacement policy can keep that chunk in memory for a long time [20].

Our experiments show that explicit I/O significantly outperforms VM as shown in Fig. 1 for the nest given in Fig. 3(a). Fig. 1 shows the performance improvement obtained by explicit I/O against VM on the Intel Paragon. Since the

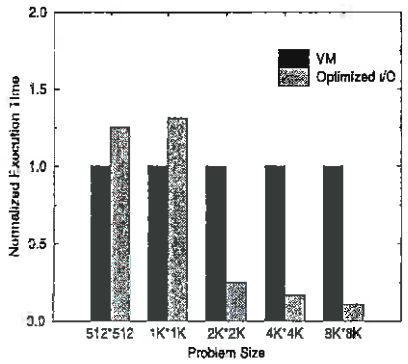

Fig. 1. Optimized I/O vs. virtual memory (VM) on Intel Paragon.

Intel Paragon has 32 Mbytes of node memory, three arrays, each of which contains $1K \times 1K$ elements of size 8 bytes (double-precision floating point data), represent the largest amount of data per node in our input set that can fit into memory. (Note that all sizes of data sets in this paper are given on a per-node basis). So, as expected, for small arrays, such as 512×512 and $1K \times 1K$, VM performs better. But starting from $2K \times 2K$ arrays, explicit I/O outperforms VM, and the performance improvement is highly significant. In the optimized approach, even for 512×512 and $1K \times 1K$ arrays, explicit I/O has been performed. This and other similar experiments convince us that the VM is not the best alternative for out-of-core computations.

Explicit file I/O, however, has its own problems. The task of programming I/O is tedious and error-prone. Low-level I/O decisions made by programmers also severely affect the portability of applications [20]. We show in this paper that a compiler approach based on explicit I/O can be very successful, provided that careful attention is paid to the layout of data in files. Note that the term layout is used to describe the layout of data in files, e.g., column-major or row-major storage, and *does not* denote partitioning or decomposition of data among processor memories.

3.2 Importance of Locality Optimizations in Explicit I/O

In order to compile out-of-core programs, the compiler has to take into account the data distribution on disks, the number of disks used for storing data etc. We identify three major issues to be exploited in order to generate efficient code for out-of-core computations.

Access pattern: The access pattern is usually a function of distribution directives and control constructs, such as loops, conditional statements etc. Since, in scientific computations, most of the execution time is spent in loop nests, we will consider loops as the sole factor determining the access pattern along with the data distribution directives.

Storage layouts in file: The storage layout for an h-dimensional array can be in one of the $h!$ forms, each of which corresponds to linear layout of data in file by means of 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 storage layout of a two-dimensional array, the second axis is the fastest changing dimension.

Memory allocation: Since node memory is a limited resource, it should be divided optimally among competing out-of-core arrays such that the total I/O time is minimized.

A compiler for out-of-core codes should optimize the access pattern, storage layout and memory allocation together in order to exploit locality. Later in this paper, we will offer automatic methods by which an optimizing compiler can achieve this goal for both individual loops and the whole program.

In order to illustrate the performance improvement that can be obtained by locality optimizations over the naive explicit I/O approach (which does not consider file layout optimizations), we will consider the nest shown in Fig. 3(a), assuming that arrays A, B and C are $n \times n$ out-of-core arrays. In the naive translation, after obtaining the node program, the compiler tiles all four loops and inserts I/O statements between tiling loops. A sketch of the resulting code is given in Fig. 3(b), assuming that n is an exact multiple of S, the tile size, and that p is the number of processors.[1] In the translated code the loops, u, v, w and y are called *tiling loops*, and the computation inside the tiling loops is performed on data tiles (sub-matrices) rather than individual array elements. In other words, there are four more loops called *element loops* (not shown for sake of clarity) that iterate over the individual elements of the data tiles of A, B and C. It should be emphasized that a reference such as $A[u, v]$ denotes a data tile of size $S \times S$ from file coordinates (u, v) as upper-left corner to $(u + S - 1, v + S \times 1)$ as lower-right corner. A reference like A[u, 1:n], on the other hand, denotes a data tile of size $S \times n$ from $(u, 1)$ to $(u + S - 1, n)$, i.e. a block of S consecutive rows of the out-of-core matrix A.

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

We ran the naive and optimized versions of this example nest on a single node of an IBM SP-2 with $4K \times 4K$ double arrays while varying the amount of available node memory. Fig. 2 shows the I/O times normalized with respect to the naive

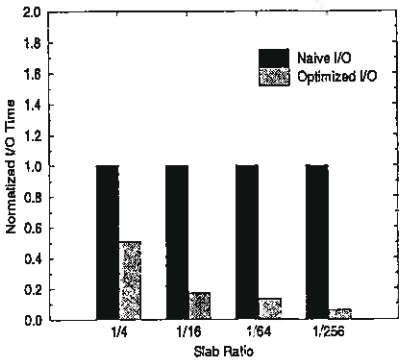

Fig. 2. Optimized I/O vs. unoptimized (native)I/O on IBM SP-2.

[1] In this example, using HPF-like distribution directives, array A is distributed in row-block, array B is distributed in column-block across the processors, and the array C is replicated. Recall that in out-of-core computations, compiler directives apply to data in files.

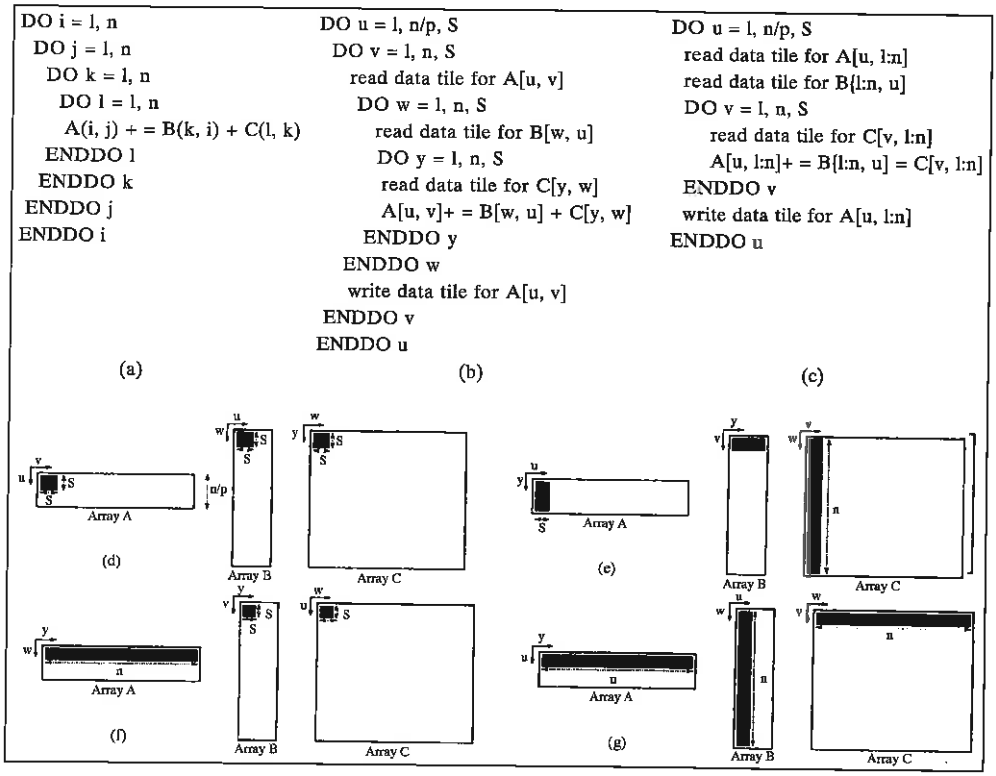

Fig. 3. (a) An out-of-core loop nest. (b) Strightforward translaion. (c) I/O optimized translation. (d)-(g) Different Tile Allocations.

(unoptimized) version. It should be noted that the performance improvement is significant especially with small values of the *slab ratio* (*SR*), the ratio of the available node memory size to the total size of the out-of-core arrays. (A smaller slab ratio indicates less available node memory.) We believe that Figs. 1 and 2 indicate the need for the compiler-directed optimized explicit I/O approach for out-of-core computations.

4. LOOP-LEVEL LOCALITY

Optimization Because of high file I/O startup costs, accessing data on disks usually is several orders of magnitude slower than accessing data in memory. Therefore, compilers must reduce the number as well as the volume of the disk accesses. In this section, we present a loop-level algorithm based on explicit file I/O to reduce the time spent in disk I/O on distributed-memory message-passing machines. Our algorithm automatically transforms a given loop nest to exploit locality in files, assigns appropriate file layouts for out-of-core arrays, and partitions the available node memory across the out -of-core arrays, all in a unified framework. We also present performance results for several kernels on an IBM

SP-2 and on an Intel Paragon. These results provide strong evidence that our algorithm can be very useful for compiling out-of-core codes in distributed-memory machines and uniprocessors.

We assume that 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 the *array reference matrix*, $\vec{b}$ is the offset vector and $\vec{I}$ is a column vector representing the loop indices $i_1, i_2, \ldots, i_n$, starting from the outermost loop. Linear mappings between iteration spaces of loop nests can be modeled by nonsingular matrices [13].[2] 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, after applying T, $T\vec{d}$ is the new distance/direction vector. A transformation is *legal* if and only if $T\vec{d}$ is lexicographically positive for every $\vec{d}$ [27]. On the other hand, since $\mathcal{L}\vec{I} = \mathcal{L}T^{-1}\vec{J}$, it is clear that $\mathcal{L}T^{-1}$ is the new array reference matrix after the transformation. We denote T^{-1} by Q. An important characteristic of our approach is that by using the array reference matrices, the entries of Q are derived systematically. For the rest of the paper, the reference matrix for array X will be denoted by $\mathcal{L}X$ whereas the i^{th} row of the reference matrix for array X will be denoted by $\vec{\ell}_i^{\,X}$. Unless otherwise stated, the word *loop* in this paper refers to a *tiling loop*; the element loops and communication statements will not be shown for sake of clarity.

4.1 Loop-Level File Locality Algorithm

Let $i_1, i_2, \ldots, i_n$ be loop indices of the original nest, and let $j_1, j_2, \ldots, j_n$ be the loop indices of the transformed nest, starting from outermost position. The algorithm works as follows. The modifications necessary for multiple LHSs and multiple nests are discussed in Section 5.

- The transformation matrix should be such that the LHS array of the transformed loop should have the innermost index as the only element in one of the array dimensions, and that 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 $C(*, \ldots, *, j_n, *, \ldots, *)$, 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 $(0, 0, \ldots, 0, 1)$, and that 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 file such that the r^{th} dimension is the fastest changing dimension. This exploits the spatial locality in the file for this reference.
- Then, the algorithm works on one reference from the RHS at a time. If a row s in the data reference matrix is identical to the r^{th} row of the original reference matrix of the LHS array, then this RHS array is considered to be stored in file such that the s^{th} dimension will be the fastest changing

[2] Since the transformation matrices for general array references obtained by our approach are not necessarily unimodular, we need a general non-singular transformation scheme like [13] rather than a simpler unimodular transformation framework.

dimension. We note that having such a row s does not guarantee that the array will be stored in the file such that the s^{th} dimension will be the fastest changing dimension.

- If the condition above does not hold for RHS array A, then the algorithm attempts to transform the reference to $A(*, \ldots,*, \mathcal{F}(j_{n-1}),*, \ldots,*)$, where $\mathcal{F}(j_{n-1})$ is an affine function of j_{n-1}, and other indices except jn, and * indicate terms independent of both j_{n-1} and j_n. This helps to exploit the spatial locality at the second innermost loop. If no such transformation is possible, j_{n-2} is tried and so on. If all loop indices are tried unsuccessfully, then the remaining entries of Q are determined by considering the data dependences and non-singularity. Notice that a modified version of the com-pletion algorithms presented by Li [13] can be used for this purpose.
- After a transformation and corresponding file layouts are found, the next alternative layout for the LHS is considered and so on. Among all feasible solutions, the best one is chosen.[3] Although several approaches can be used to select the best alternative, we have found the following scheme both accurate and practical: Each loop in the nest is numbered with its level (depth), the outermost loop getting the number 1. Then, for each reference in the nest, the level number of the loop whose index sits in the fastest changing dimension for this reference is recorded. The numbers for all references in the nest are summed up, and the alternative with the maximum sum is chosen. As an example, if, for a two-deep nest with three references, an alternative exploits the locality for the first reference in the outer loop and for the other references in the inner loop the sum for this alternative is 1 + 2 + 2 = 5.
- After choosing the best alternative, the following memory allocation scheme [15] is applied: First, the data tile size for each dimension of each array is set to S, a parameter whose value depends on the size of the available memory. For example, if a loop nest contains a one-dimensional array, two two-dimensional arrays and a three-dimensional array, the algorithm first allocates a tile of size S for the one-dimensional array, a tile of size $S \times S$ for each of the two-dimensional arrays, and a tile of size $S \times S \times S$ for the three-dimensional array. This allocation scheme implies the memory constraint $S^3 + 2S^2 + S \leq M$, where M is the size of the node memory. After that, the arrays are divided into groups according to the file layouts of the associated files (i.e., the arrays with the same file layout are placed in the same group). The algorithm then handles the groups one by one. If a loop index appears in the fastest changing array dimension of the group and does not appear in any other dimension of any reference in that group, the compiler increases the tile size in that dimension to full (local) array size for that reference. After this increase, the memory constraint should be adjusted accordingly. Note that any inconsistency between the groups (due to a common loop index) should be resolved by not changing the original (initial) tile sizes.

[3] We should emphasize that, for the global I/O problem (see Section 5), it might be better to record and consider all feasible solutions (alternatives).

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

4.2 Constraints

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

We will now focus on the problem of optimizing locality when some or all array layouts are *fixed*, as frequently occurs in practice. We note that each fixed layout requires that the innermost loop index be in the appropriate array index position (dimension), depending in 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 chang-ing dimension, k_3 is the third etc. The algorithm should first try to place the new innermost loop index j_n only in the k_1^{th} dimension of this array. If this is not possible, then it should try to place j_n only in 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 dimension and so on. As we will show later, this modified algorithm with the constrained layouts is very important for global I/O optimization.

4.3 Example

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

Consider the example shown in Fig. 3(a). Fig. 3(b) presents a naive out-of-core translation for it. The tile allocations for this naive version are illustrated in Fig. 3(d). The array reference matrices are as follows:

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

$L^A.Q = \begin{bmatrix} 0\,0\,0\,1 \\ \delta\,\delta\,\delta\,0 \end{bmatrix}$. Therefore, $q_{11} = q_{12} = q_{13} = q_{24} = 0$ and $q_{14} = 1$. $L^B.Q = \begin{bmatrix} \delta\,\delta\,\delta\,0 \\ 0\,0\,0\,1 \end{bmatrix}$. Therefore, $q_{34} = 0$. $L^C.Q = \begin{bmatrix} \delta\,\delta\,1\,0 \\ \delta\,\delta\,0\,0 \end{bmatrix}$. Therefore, $q_{33} = q_{44} = 0$ and $q_{43} = 1$. At this point $T^{-1} = Q = \begin{bmatrix} 0 & 0 & 0 & 1 \\ q_{21} & q_{22} & q_{23} & 0 \\ q_{31} & q_{32} & 0 & 0 \\ q_{41} & q_{42} & 1 & 0 \end{bmatrix}$. We set the unknowns to the following values: $q_{22} = q_{23} = q_{31} = q_{41} = q_{42} = 0$ and $q_{21} = q_{32} = 1$, and obtain $T^{-1} = Q = \begin{bmatrix} 0\,0\,0\,1 \\ 1\,0\,0\,0 \\ 0\,1\,0\,0 \\ 0\,0\,1\,0 \end{bmatrix}$. The resulting code is shown in Fig. 4(a).

Arrays A and C are column-major whereas array B is row-major. By using our memory allocation scheme explained earlier, a tile of size $n/p \times S$ is allocated for A, one of size $n \times S$ for C, and one of size $S \times p$ for B. The final memory constraint is $2nS/p + nS \leq M$. Tile allocations are shown in Fig. 3(e).

Next, the algorithm considers the other alternative layout (row-major) for A. $L^A.Q = \begin{bmatrix} \delta\,\delta\,\delta\,0 \\ 0\,0\,0\,1 \end{bmatrix}$. Therefore $q_{14} = q_{21} = q_{22} = q_{23} = 0$ and $q_{24} = 1$. $L^B.Q = \begin{bmatrix} \delta\,\delta\,1\,0 \\ \delta\,\delta\,0\,0 \end{bmatrix}$. Therefore $q_{13} = q_{34} = 0$ and $q_{33} = 1$. $L^C.Q = \begin{bmatrix} \delta\,\delta\,0\,0 \\ \delta\,\delta\,1\,0 \end{bmatrix}$. Therefore $q_{43} = q_{44} = 0$. At this point $T^{-1} = Q = \begin{bmatrix} q_{11} & q_{12} & 0 & 0 \\ 0 & 0 & 0 & 1 \\ q_{31} & q_{32} & 1 & 0 \\ q_{41} & q_{42} & 0 & 0 \end{bmatrix}$. By setting $q_{12} = q_{31} = q_{32} = q_{41} = 0$ and $q_{11} = q_{42} = 1$, $T^{-1} = Q = \begin{bmatrix} 1\,0\,0\,0 \\ 0\,0\,0\,1 \\ 0\,0\,1\,0 \\ 0\,1\,0\,0 \end{bmatrix}$. The resulting code is shown in Fig. 4(b).

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

(a)

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

(b)

Fig. 4. Code resulting from loop transformations.

loop transformation for a given nest. However, different nests in the program may impose conflicting requirements on the file layouts of out-of-core arrays. The main contribution of this section is to present algorithms to resolve those conflicts. The success of our strategy depends greatly on the ability to resolve these conflicts, such that the file locality will be exploited for as many references as possible. We note that this problem is similar to that of determining the global data distribution across multiple nests [12, 17] on distributed-memory machines and can be attacked with similar algorithms. However, the assumptions we make, the interaction between file layouts and loop orders, and the excessive cost of redistributing disk-resident data between loop nest boundaries force (and enable) different methods to solve the global I/O optimization problem. Specifically, the separate nests assumption enables us to re-order loop nests for the sake of determining appropriate global file layouts. In the rest of this section, we will use *local()* to denote the loop-level optimization algorithm (see Section 4.1), which takes all file layout combinations into account, and we will use *constrained-local()* to denote the algorithm which considers the fixed file layouts (see Section 4.2).

5.1 First Approach

In this subsection, we show how the file layouts can be determined globally, assuming that *local()* is run for each individual nest. Since a number of arrays can be accessed by a number of nests and each of these nests may require a different file layout for a specific out-of-core array, the algorithm should find a layout for the array that satisfies a majority of the nests.

General Problem

Let $\{\mathcal{N}_1, \mathcal{N}_2, \ldots, \mathcal{N}_n\}$ denote the different loop nests in the program, and let $\{\mathcal{A}_1, \mathcal{N}_2, \ldots, \mathcal{N}_m\}$ denote the different out-of-core arrays. In general, each nest can access a subset of these arrays. We assume that *local()* is run for each nest, and that a number of possible optimized layout combinations are obtained for each nest. In the following, we will show that the problem of *finding a global array layout combination that satisfies all the nests* is NP-complete even for the restricted case where only r-m and c-m file layouts are considered.

When *local()* is run for each nest in the program, we obtain optimal file layout combinations similar to those shown in Table 2(a). For example, nest

Table 2. Local layout assignments for different examples.

(a)

	$\mathcal{A}_1$	$\mathcal{A}_2$	$\mathcal{A}_3$	$\mathcal{A}_4$	$\mathcal{A}_5$
	r-m	r-m	r-m		
$\mathcal{N}_1$	r-m	c-m	c-m		
		c-m			c-m
$\mathcal{N}_2$		r-m			c-m
$\mathcal{N}_3$	c-m		c-m	r-m	

(b)

	$\mathcal{A}_1$	$\mathcal{A}_2$	$\mathcal{A}_3$	$\mathcal{A}_4$	$\mathcal{A}_5$
$\mathcal{N}_1$	r-m	c-m	c-m		
$\mathcal{N}_2$		c-m	c-m	r-m	r-m
$\mathcal{N}_3$		r-m	r-m	r-m	

(c)

	$\mathcal{A}_1$	$\mathcal{A}_2$	$\mathcal{A}_3$	$\mathcal{A}_4$	$\mathcal{A}_5$
	r-m	c-m	c-m		
$\mathcal{N}_1$	r-m	r-m	c-m		
$\mathcal{N}_2$		c-m	c-m	r-m	r-m
		r-m	r-m	r-m	
		r-m	c-m	r-m	
$\mathcal{N}_3$		c-m	c-m	c-m	

$\mathcal{N}_1$ accesses three out-of-core arrays, and *local()* returns two optimal layout combinations for that nest. We define the number of entries in the table as the size of the problem.

First, the problem belongs to the NP class; this is because a nondeterministic algorithm need only guess a solution and check in polynomial time whether or not it satisfies all the nests. Next, we reduce the *satisfiability* problem [9] to our problem as follows: A given formulation is transformed into multiplications of sums (a polynomial-time operation). After that, each multiplicative term is associated with a nest, and each sub-term (clause) in a multiplicative term is associated with a layout combination. With each logical variable x, we associate an out-of-core array X. If the logical variable itself appears, we assign a c-m file layout for X; if $\overline{x}$ (complement of x) appears, we assign an r-m file layout for X.

For example, the layout assignments shown in Table 2(a) correspond to the following formulation, where a_i and a_i are the logical variables associated with array $\mathcal{A}_i$:

$$(\overline{a}_1\overline{a}_2\overline{a}_3 + \overline{a}_1 a_2 a_3) \cdot (a_2 a_5 + \overline{a}_2 a_5) \cdot (a_1 a_3 \overline{a}_4).$$

There might be some special cases, however, which must be dealt with. For example, after obtaining the desired form, a multiplicative term can contain expressions such as in $(ac + b\overline{c})$, which does not have all the variables. This expression should be transformed into $(a(b + \overline{b})c = (a + \overline{a})b\overline{c}) = abc + a\overline{b}c + ab\overline{c} + \overline{a}b\overline{c}$ so that each sub-term contains the logical variables a, b and c or complements of them.

It is easy to see that the formulation is satisfied *if and only if* there is a layout assignment that satisfies all the nests. Since the reduction can be achieved in polynomial time, the problem is NP-hard; and since it belongs to the class NP as well, it is NP-complete. Note that as this problem is a restricted version of the most general problem of finding suitable layout assignments such that the value of a cost function will be $\leq k$; we argue that the general problem is also NP-complete.[5]

Heuristic Solution

Given that even the restricted form of the global layout problem is NP-complete, we can search for a near-optimal solution in polynomial time which is good enough in practice. Let $LL^{\ell}_{\mathcal{N}}(\mathcal{A})$ be a local layout for an array $\mathcal{A}$ in a combination ℓ for nest $\mathcal{N}$, and let $GL(\mathcal{A})$ be the global layout for array $\mathcal{A}$. We define the following parameter:

$$\mu(\mathcal{A}, \mathcal{N}, \ell) = \begin{cases} 0 & \text{if } LL^{\ell}_{\mathcal{N}}(\mathcal{A}) = GL(\mathcal{A}) \text{ or } \mathcal{A} \text{ is not referred in } \mathcal{N} \\ 1 & \text{otherwise} \end{cases}.$$

Given this definition of μ, the cost of nest $\mathcal{N}$ under local layout combination ℓ is $LCost(\mathcal{N}, \ell) = \Sigma_{\mathcal{A}}\mu(\mathcal{A}, \mathcal{N}, \ell)$. Similarly, $ACost(\mathcal{A}, \ell) = \Sigma_{\mathcal{A}}\mu(\mathcal{A}, \mathcal{N}, \ell)$ is the cost

[5] In this restricted version k = 0.

Step 3: Suppose that, without loss of generality, $\{\mathcal{N}_1, \mathcal{N}_2, \ldots, \mathcal{N}_n\}$ is the order obtained by the previous step. We construct our locality graph as follows: For each alternative layout combination of each nest, we create a node. This node is given the name $N_{i,j}$, where i is the nest number, and j is the number of the alternative. There is a directed edge from $N_{i,j}$ to $N_{i+1,k}$ for all $1 \leq j \leq$ *alternatives* $(\mathcal{N}_i)$ and $1 \leq k \leq$ *alternatives* $(\mathcal{N}_{i+1})$. This edge is annotated by a set of arrays whose local file layouts differ in $\mathcal{N}_i$ and $\mathcal{N}_j$. The *cost* of this edge is defined as the number of arrays. A source node(S) and a target node (T) (both with zero cost) are also added to the locality graph such that there is an edge from S to $\mathcal{N}_{1,j}$ for all $1 \leq j \leq$ *alternatives* $(\mathcal{N}_1)$ and an edge from $\mathcal{N}_{n,k}$ to T for all $1 \leq k \leq$ *alternatives* $(\mathcal{N}_n)$. Then, a shortest path algorithm for this locality graph is run from S to T. The path with the minimum cost gives a good local layout combination for each nest. Fig. 8(a) shows the locality graph obtained by the order $\{\mathcal{N}_1, \mathcal{N}_2, \mathcal{N}_3\}$ for the example given in Table 2(c). The edges are annotated with the arrays whose layouts are different in the neighboring nests. The shortest path algorithm on this locality graph returns with two *near-optimal* solutions: $\mathcal{N}_{1,1}$, $\mathcal{N}_{2,1}$, $\mathcal{N}_{3,2}$ and $\mathcal{N}_{1,1}$, $\mathcal{N}_{2,1}$, $\mathcal{N}_{3,3}$, omitting the source and target. In the first solution, a cost occurs due to array $\mathcal{A}_2$ whereas in the second solution, the cost is due to array $\mathcal{A}_4$. Now let us concentrate on the locality graph in Fig. 8(b), which is obtained from the order $\{\mathcal{N}_2, \mathcal{N}_1, \mathcal{N}_3\}$. In this locality graph, the shortest path algorithm returns the solution $\mathcal{N}_{2,1}$, $\mathcal{N}_{1,1}$, $\mathcal{N}_{3,3}$ with a cost of zero. However, if these local layouts are assigned, there will be a cost originating from the conflicting requirements on $\mathcal{A}_4$ by $\mathcal{N}_{2,1}$ and $\mathcal{N}_{3,3}$. Since our approach only considers the adjacent nest pairs, this cost does not reflect on the cost of the shortest path, and the solution is still sub-optimal. This problem occurs because of the fact that $\mathcal{A}_4$ is not referenced in $\mathcal{N}_1$ but is referenced in both of its neighbors ($\mathcal{N}_2$ and $\mathcal{N}_3$). In fact, this is the case which the max-accuracy heuristic is designed to eliminate.

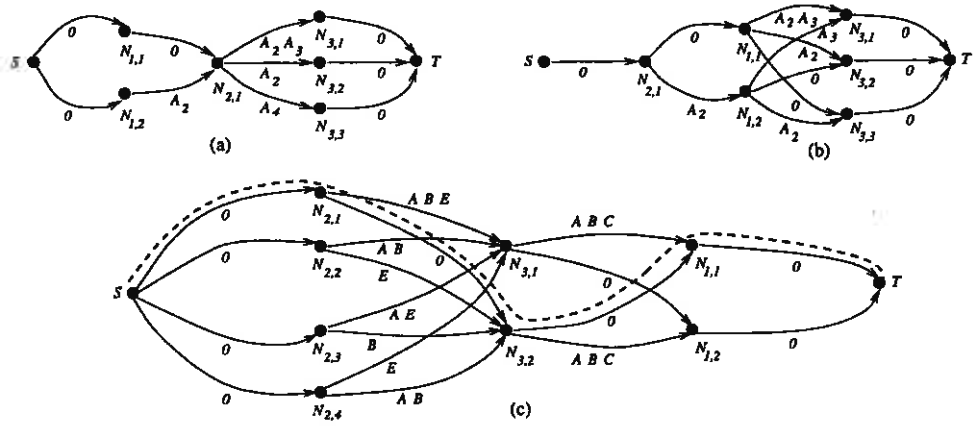

Fig. 8. (a)-(b) *Locality graphs* for the example shown in Table 2(c). (c) *Locality graph* for the example shown Fig. 10.

Step 4: The final phase of the heuristic determines the global file layouts using the local file layout assignments obtained in the previous step. We refer to the shortest path obtained in the previous step as δ, and the i^{th} node of the shortest path (excluding the source and target) is denoted as δ_i. Suppose that there is a conflict between δ_i and δ_{i+1} on an array $\mathcal{A}$. In order to resolve this conflict, the layout for $\mathcal{A}$ should be changed in either δ_i or in δ_{i+1}, as we do not consider data redistribution in this paper. Our approach decides which alternative will be changed by considering all the nodes along the shortest path. The algorithm traverses the shortest path and records, for each array for which there are conflicting demands, the number of r-m and c-m demands. Then, in an attempt to satisfy the majority of the nests, it chooses the layout that occurs most frequently. Notice that this is exactly the same procedure used to solve the simpler case of the problem (see Table 2(b)). After that, the local layouts in a nest which are different from the global layouts are changed accordingly.

To sum up, after the third phase, the local layout combinations for each nest, and after the fourth phase, the global layout combination for the whole program, are determined, and then the local layouts are adjusted accordingly. After the final layouts are determined, the approach given in [14] and [15] can be used to find an appropriate loop order for each nest and optimal memory allocations under memory constraints.

Example

In order to demonstrate the use of this technique on a complete program, we consider the program shown in Fig. 10. When *local()* is run for each nest of this program, it returns the information shown in Table 3. The first phase of the algorithm returns only a single connected component. In the second phase, min-edge gives two preferred orders: $\{\mathcal{N}_2, \mathcal{N}_1, \mathcal{N}_3\}$ and $\{\mathcal{N}_2, \mathcal{N}_3, \mathcal{N}_1\}$, both with 18 edges on the corresponding locality graphs. The max-accuracy heuristic, on the other hand, returns $\{\mathcal{N}_2, \mathcal{N}_3, \mathcal{N}_1\}$ and $\{\mathcal{N}_3, \mathcal{N}_2, \mathcal{N}_1\}$. Since $\{\mathcal{N}_2, \mathcal{N}_3, \mathcal{N}_1\}$ is common to both heuristics, we select that order for the locality graph to be constructed. The resulting locality graph is shown in Fig. 8(c). The shortest path algorithm returns

Table 3. Local layout assignments for the program shown in Fig. 10.

	A	B	C	D	E
$\mathcal{N}_1$	r-m	c-m	r-m		
	c-m	r-m	c-m		
$\mathcal{N}_2$	r-m	c-m		r-m	r-m
	r-m	c-m		r-m	c-m
	r-m	r-m		c-m	r-m
	c-m	r-m		c-m	r-m
$\mathcal{N}_3$	c-m	r-m	c-m		c-m
	r-m	c-m	r-m		r-m

the path $\{\mathcal{N}_{2,1}, \mathcal{N}_{3,2}, \mathcal{N}_{1,1}\}$ (shown as a thick dashed curve) with a zero pair-wise cost. It should be noted that, for this example, that assignment happens to be optimal as well. That is, the optimal global file layouts for A, B, C, D and E are r-m, c-m, r-m, r-m and r-m, respectively.

5.2 Second Approach

In this subsection, we present our second alternative solution to the global layout determination problem. The key difference between this and the previous method is that this method does not run *local()* for all the loop nests in the program; instead, it runs it for only a subset of the nests (for a single nest if the interference graph of the program contains only one connected component). For the remaining nests, the heuristic runs *constrained-local()*, which is similar in functionality to *local()* except that it takes into account the already determined file layouts (see Section 4.2). Since, in general, *constrained-local()* is less costly than *local()*, the approach to be presented shortly has less overall cost.

The strategy behind this approach is based on propagation of the partial layout constraints across multiple loop nests. After each loop nest is processed, the file layouts of some of the out-of-core arrays are determined, and these are propagated as new layout constraints to the next loop nest. In this manner, we keep building upon the partial file layout information until all file layouts are determined. The entire algorithm for this approach to the global layout determination problem is shown in Fig. 9(b). A description of the approach follows:

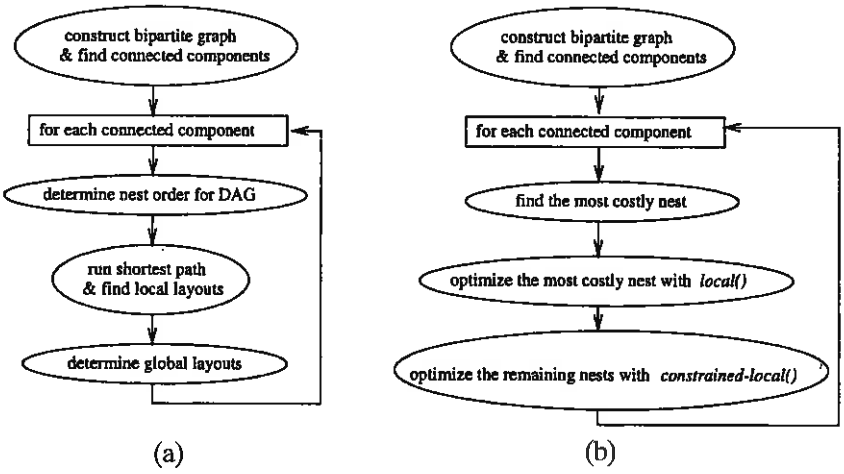

Fig. 9. Two solutions to the global layout determination problem.

(1) First, as before, we construct a bipartite interference graph to find the connected components of the program. Each connected component represents a set of loop nests accessing a set of out-of-core arrays. The following phases operate on a single connected component at a time.

(2)In this phase, the loop nests are ordered (again not textually) according to a cost criterion. The order is based on the concept of the *I/O cost* of a loop nest. Although it is difficult to the define the I/O cost of a loop nest precisely, we find the multiplica-tion of the number of loops and the number of different out-of-core arrays in the nest to be a viable approximation in practice. This is especially true when the arrays have the same dimensionality and the trip counts of the loops are of the same order. Then the nests are ordered according to *non-increasing* I/O costs. The rest of the algorithm is independent of the I/O cost criterion used.

(3)The most costly nest is optimized by using *local()* to select a layout combination. After this step, file layouts for some of the out-of-core arrays are determined. Then each of the remaining nests is optimized using *constrained-layout()*. After each nest is optimized, new file layout constraints will be obtained, and these will be propagated for optimization of the next nest. This procedure continues until either of the following occurs: (a) The file layouts for all arrays are determined before all the loops are processed. In this case, for the remaining loops, only loop transformations [14] are used to optimize the file locality under the fixed file layout constraints. (b) All the loop nests are processed. If undetermined file layouts, still remain, arbitrary layouts were assigned for them.

Notice, that while in the previous approach, local analysis and global analysis were separated, in this approach, they proceed together to find the global layouts.

Example

We will revisit the example shown in Fig. 10. Our simple heuristic as explained above orders the nests as $\{\mathcal{N}_2, \mathcal{N}_3, \mathcal{N}_1\}$with the I/O costs {16, 12, 9}, respectively. For the most costly nest, $\mathcal{N}_2$, *local()* returns r-m, r-m, c-m and r-m for D, A, B and E, respectively, as one of the alternatives, and I, K, L, J is the desired loop order from the outermost position. For $\mathcal{N}_3$, on the other hand, *constrained-local()* orders the loops as J, K, I and assigns an r-m layout to C. Finally, since all the layouts have been determined, *constrained-local()* reorders the loops in $\mathcal{N}_1$ as I, K, J to optimize locality. It should be noted that these layouts are exactly the same ones obtained using the first approach.

```
                          N1:                          N2:  N3:
DO I = 1, n               DO I = 1, n                  DO I = 1, n
  DO J = 1, n               DO J = 1, n                  DO J = 1, n
    DO K = 1, n               DO K = 1, n                  DO K = 1, n
      A(I,J) = B(J,K) + C(I,J)    DO L = 1, n                B(I,J) = E(J,I) + A(J,I) + C(K,I)
    ENDDO                         D(I,J) = E(K,I) + A(K,J) + B(L,I)    ENDDO
  ENDDO                         ENDDO                        ENDDO
ENDDO                         ENDDO                        ENDDO
                            ENDDO
                              ENDDO
```

Fig. 10. An out-of-core program consisting of three loop nests.

5.3 Experimental Results

The experiments were conducted on an Intel Paragon, for different values of the *slab ratio* (*SR*), the ratio of the available node memory to the total number of all out-of-core local arrays. The transformations to the original programs were applied manually following the algorithms.

A Simple Benchmark

We experimented with four versions of the example shown in Fig. 10 on a single node of the Intel Paragon: UNOPT-ROW: Unoptimized case with row-major file layout for all arrays; UNOPT-COLUMN: Unoptimized case with column-major file layout for all arrays; UNOPT-ARBITRARY: a case with arbitrary file layout assignments; and OPT: our optimized version.[6] The results for eight different cases on a single node are shown in Table 4. Here, n is the problem size ($n = 2K$ denotes $2K \times 2K$ double arrays), and SR is the slab ratio. Table 5, on the other hand, shows the distribution of the read/write requests by size (in 1024 byte units) for the same cases. It can be observed from Table 4 that our approach (OPT) reduces the execution time by reducing the number of I/O calls significantly. There is a 2-16 factor decrease in the number of reads and a 372-687 factor decrease in the number of writes in the optimized program. Also, it can be noted from Table 5 that the optimized programs issue I/O calls for larger requests, thus utilizing the available bandwidth better.

Out-of-Core 2-D FFT

Fast Fourier transform (FFT) is widely used in many areas, such as digital signal processing and partial differential equation solutions. We implemented 2-D out-of-core FFT on the Intel Paragon. 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 to each of the columns. After that, 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[7] was applied to the original program, and the results are presented in Table 6 for different cases (where p denotes the number of processors). In the table, the breakdown of the total execution times for the unoptimized (UNOPT) and optimized (OPT) cases is shown. The overhead time includes the times for file open and close operations, for buffer copying, and other times for communication initializations. The results show that there is a 17-30% reduction in the I/O time when OPT is used. We also note that the effectiveness of the approach increases with an increasing number of processors, a decreasing slab ratio and increasing problem size. That is, the global layout optimization improves the scalability of the application as well as the execution time.

[6] Recall that both of our approaches resulted in the same global file layouts for this example, but this may not always be the case.
[7] Both approaches produced the same resulting code-OPT.

Table 4. I/O information about the example shown in Fig. 10.

Slab Ratio (SR) = 1/16 and $n = 2K$

UNOPT-ROW				UNOPT-COLUMN			
Operation	Oper. Count	Oper. I/O time	I/O Bandwidth	Operation	Oper. Count	Oper. I/O time	I/O Bandwidth
Open/Close	10	0.53		Open/Close	10	0.51	
Read	310,423	4,086.23	208 *K/sec*	Read	39,045	628.05	836 *K/sec*
Write	16,776	748.15	2,287 *K/sec*	Write	12,295	415.91	161 *K/sec*
Seek	327,199	115.34		Seek	51,340	18.09	

UNOPT-ARBITRARY				OPT			
Operation	Oper. Count	Oper. I/O time	I/O Bandwidth	Operation	Oper. Count	Oper. I/O time	I/O Bandwidth
Open/Close	10	0.47		Open/Close	10	0.42	
Read	229,512	3,436.09	254 *K/sec*	Read	20,895	401.10	3,178 *K/sec*
Write	13,298	937.70	3,650 *K/sec*	Write	33	18.96	5,309 *K/sec*
Seek	242,810	85.30		Seek	20,928	7.34	

Slab Ratio (SR) = 1/18 and $n = 2K$

UNOPT-ROW				UNOPT-COLUMN			
Operation	Oper. Count	Oper. I/O time	I/O Bandwidth	Operation	Oper. Count	Oper. I/O time	I/O Bandwidth
Open/Close	10	0.76		Open/Close	10	0.69	
Read	137,220	1,964.56	307 *K/sec*	Read	24,632	1,094.72	398 *K/sec*
Write	12,368	584.83	1,032 *K/sec*	Write	8,196	490.91	136 *K/sec*
Seek	149,588	51.99		Seek	32,828	11.73	

UNOPT-ARBITRARY				OPT			
Operation	Oper. Count	Oper. I/O time	I/O Bandwidth	Operation	Oper. Count	Oper. I/O time	I/O Bandwidth
Open/Close	10	0.70		Open/Close	10	0.68	
Read	90,168	2,018.67	299 *K/sec*	Read	12,423	328.55	2,348 *K/sec*
Write	8,413	600.48	171 *K/sec*	Write	18	57.70	1,744 *K/sec*
Seek	98,581	34.50		Seek	12,441	4.40	

Table 5. Distributions of read/write requests by size (sz) for the example shown in Fig. 10.

		Slab Ratio (SR) = 1/16 and $n = 2K$			Slab Ratio (SR) = 1/8 and $n = 2K$		
Version	Operation	$sz < 4K$	$4K \le sz \le 256K$	$sz > 256K$	$sz < 4K$	$4K \le sz \le 256K$	$sz > 256K$
UNOPT-ROW	Read	242,832	67,584	7	34,816	102,400	4
UNOPT-ROW	Write	4,096	12,288	392	4,112	8,192	64
UNOPT-COLUMN	Read	38,912	0	133	6,144	18,432	56
UNOPT-COLUMN	Write	12,288	0	7	2,048	6,144	4
UNOPT-ARBITRARY	Read	229,376	0	136	77,824	12,288	56
UNOPT-ARBITRARY	Write	12,288	0	1,010	2,048	6,144	221
OPT	Read	20,480	0	415	12,288	0	135
OPT	Write	0	0	33	0	0	18

(1995, 1996), Supercomputing 94, Scalable High-Performance Computing Conference (SHPCC 94), and the International Symposium on Computer Architecture (1993 an 1994).

M. Kandaswamy graduated with a B.E. (Honors) in Computer Science and Engineering from the Regional Engineering College, Trichy, India in 1989 and with an M.S. in Computer Science from Syracuse University in 1995. She is currently a Ph.D. candidate in the Department of Electrical Engineering and Computer Science at Syracuse University, She was also a summer intern in the areas of parallel compilers and parallel I/O during the summers of 1993 and 1994, respectively, at Intel Supercomputer Systems Division, Beaverton, Oregon. Her research interests include high-performance I/O, parallel applications, multiprocessor file systems and memory hierarchies.