The Journal of Supercomputing, 21, 257–284, 2002

An I/O-Conscious Tiling Strategy for Disk-Resident Data Sets

MAHMUT KANDEMIR kandemir@cse.psu.edu
CSE Department, The Pennsylvania State University, University Park, PA 16802

ALOK CHOUDHARY choudhar@ece.nwu.edu
ECE Department, Northwestern University, Evanston, IL 60208

J. RAMANUJAM jxr@ee.lsu.edu
ECE Department, Louisiana State University, Baton Rouge, LA 70803

Abstract. This paper describes a tiling technique that can be used by application programmers and optimizing compilers to obtain I/O-efficient versions of regular scientific loop nests. Due to the particular characteristics of I/O operations, a straightforward extension of the traditional tiling method to I/O-intensive programs may result in poor I/O performance. Therefore, the technique presented in this paper adapts iteration space tiling for I/O-performing loop nests to deliver high I/O performance. The generated code results in huge savings in the number of I/O calls as well as the volume of data transferred between the disk subsystem and main memory. Our experimental results on the IBM SP-2 distributed-memory message-passing multiprocessor demonstrate that the reduction in these two parameters, namely, the number of I/O calls and the transferred data volume, can lead to a marked decrease in overall execution times of I/O-intensive loop nests. In a number of loop nests extracted from several benchmarks and math libraries, we were able to improve the execution times by an average 42.5% for one data set and by an average 47.4% for another.

Keywords: optimizing compilers, I/O-intensive codes, tiling, file layouts, disk-resident arrays

1. Introduction

An important problem that scientific programmers face today is one of writing programs that perform I/O in an efficient manner. Unfortunately, a number of factors render this problem very difficult. First, programming I/O is highly architecture-dependent, i.e., low-level optimizations performed with a specific I/O model in mind may not work well in systems with different I/O models and/or architectures. Second, there is very little help from compilers and runtime systems to optimize I/O operations. While optimizing compiler technology [49] has made impressive progress in analyzing and improving regular array access patterns and loop structures in scientific codes, the main focus of almost all the work published so far is the so called CPU-intensive computations, i.e., computations that make frequent use of the cache–main memory hierarchy rather than I/O-intensive computations [8, 9, 50, 51], where the main memory–disk subsystem hierarchy is heavily utilized. Exploiting locality in I/O-intensive computations that operate on disk-resident data might be more important since the disk access costs are 3 to 5 orders of magnitude

higher than memory access costs, whereas memory access times are just an order of magnitude higher than cache accesses. Third, the large quantity of data involved in I/O operations makes it difficult for the programmer to derive suitable data management and transfer strategies. A feasible solution perhaps is to use existing parallel I/O libraries [42, 46] or parallel file systems [15, 38] that, in general, are highly architecture-dependent and application-insensitive. Also, the extensibility of the parallel I/O libraries is severely limited by the design principles and the programming language used in the implementation.

It is extremely important to perform I/O operations as efficiently as possible, especially on parallel architectures, which are natural target platforms for grand-challenge I/O-intensive applications [20]. It is widely acknowledged by now that the per-byte cost (time) of I/O is orders of magnitude higher than those of communication and computation [17]. A direct consequence of this phenomenon is that no matter how well communication and computation are optimized, poor I/O performance can lead to unacceptably low overall performance on parallel architectures. Given the current improvement rate of processor speeds, and memory and disk access times, we can only expect this problem to become worse in the future if not addressed correctly. Therefore, we believe that writing I/O-intensive codes is very crucial. Unfortunately, we cannot let the job of handling very large datasets to virtual memory (VM), as it is already known that a VM-based approach may not perform well for scientific codes.

A subproblem within this context is that of writing I/O-efficient versions of loop nests that are common in regular scientific codes. This subproblem is very important since the bulk of the execution time in scientific codes is spent in loop nests. Thus, it is reasonable to assume that in scientific codes that perform frequent I/O, a majority of the execution time will be spent in I/O-intensive loops, i.e., loop nests that access disk-resident multidimensional arrays. Having decided to focus on loop structures, the important issue to be addressed is the extent to which the existing optimizing compiler techniques developed for nested loops can be used for improving the behavior of the loops that perform I/O. Although at a first glance, it appears to be perfectly reasonable to assume that the available compiler technology [31, 48, 49] developed for optimizing the cache–main memory hierarchy can be used for optimizing main memory—disk subsystem hierarchy as well, as we show in this paper, this does *not* necessarily hold. In particular, iteration space (loop) tiling [14, 22, 29, 37, 49], a prominent cache locality enhancing technique, can result in suboptimal I/O performance if applied to I/O-performing nests without modification.

In this paper, we propose a new tiling approach called *I/O-conscious tiling* that is customized for I/O performing loops that access large disk-resident datasets. Our approach takes as input an unoptimized loop nest (without any I/O calls inserted) and outputs an optimized version with explicit I/O calls. We believe that this approach will be useful for at least two reasons. First, we express our solution in a simple yet powerful framework so that in many cases it can be carried out by application programmers without much difficulty. Second, we show that it is also possible to automate the approach so that it can be implemented within an optimizing compiler framework without increasing the complexity of the compiler and compilation time excessively. Experimental results obtained on the IBM

SP-2 distributed-memory message-passing multiprocessor reveal that our approach achieves significant improvements in overall execution times in comparison with traditional tiling. This is largely due to the fact that, while performing tiling, our approach takes I/O specific factors (e.g., file layouts) into account, which leads to substantial reductions in the number of I/O calls as well as the number of bytes transferred between main memory and disk.

The remainder of this paper is organized as follows. Section 2 reviews the basic principles of tiling, focusing, in particular, on state-of-the-art tiling strategy. Section 3 discusses why the traditional tiling approach may not be very efficient in coding I/O-performing versions of loop nests found in regular scientific codes and makes a case for our modified (I/O-conscious) tiling approach. Section 4 describes our technique in detail and also discusses briefly how multiple loop nests and whole programs with procedures can be handled. Section 5 presents our experimental results obtained on the IBM SP-2. Section 6 reviews related work on data locality and I/O optimizations. Section 7 summarizes our experience and briefly comments on future work.

2. Tiling for data locality

In this section, we briefly discuss traditional tiling and explain how it improves locality properties of scientific loop nests.

We say that an array element has *temporal reuse* when it gets accessed more than once during the execution of a loop nest [21]. *Spatial reuse*, on the other hand, occurs when nearby items are accessed [21]. The definition of *nearby* can take on different forms depending on the granularity of the computation at hand and the memory hierarchy in question. For example, with many CPU-intensive computations where the data sets fit in main memory, the array elements (data items) mapped on the same cache line can be considered to have spatial locality; whereas in I/O-intensive computations, the elements mapped on the same memory page are said to have spatial locality.

Tiling (or blocking) [14, 22, 29, 37, 47, 52] is a well known optimization technique for enhancing memory performance and parallelism in nested loops. Instead of operating on entire columns or rows of a given array, tiling enables operations on multi-dimensional sections of arrays at one time. The aim is to keep the active sections of the arrays in faster levels of memory hierarchy as long as possible so that when an array item (data element) is reused, it can be accessed from the faster (higher level) memory instead of the slower (lower level) memory. In the context of I/O-performing loop nests, the faster level is the *main memory* and the slower level is the *disk subsystem (secondary storage)*. Therefore, we want to use tiling to enable the reuse of the array sections in memory as much as possible while minimizing disk activity.

For an illustration of tiling, consider the matrix-multiply code given in Figure 1(a). Let us assume that the layouts of all the arrays are *row-major*. It is easy to see that from the locality perspective, this loop nest may not exhibit a very good performance (depending on the actual array sizes and cache capacity). The reason is that,

```
for i = 1, N
 for j = 1, N
  for k = 1, N
   Z(i,j) += X(i,k)*Y(k,j)
  endfor
 endfor
endfor
```
(a)

```
for i = 1, N
 for k = 1, N
  for j = 1, N
   Z(i,j) += X(i,k)*Y(k,j)
  endfor
 endfor
endfor
```
(b)

```
for kk = 1, N, B
 kkbb = min(N,kk+B-1)
 for jj = 1, N, B
  jjbb = min(N,jj+B-1)
  for i = 1, N
   for k = kk, kkub
    for j = jj, jjub
     Z(i,j) += X(i,k)*Y(k,j)
    endfor
   endfor
  endfor
 endfor
endfor
```
(c)

```
for kk = 1, N, B
 kkbb = min(N,kk+B-1)
 ioread X[1:N,kk:kkub]
 for jj = 1, N, B
  jjbb = min(N,jj+B-1)
  ioread Z[1:N,jj:jjbb]
  ioread Y[kk:kkub,jj:jjub]
  for i = 1, N
   for k = kk, kkbb
    for j = jj, jjbb
     Z(i,j) += X(i,k)*Y(k,j)
    endfor
   endfor
  endfor
  iowrite Z[1:N,jj:jjbb]
 endfor
endfor
```
(d)

```
for ii = 1, N, B
 iibb = min(N,ii+B-1)
 for kk = 1, N, B
  kkbb = min(N,kk+B-1)
  for i = ii, iibb
   for k = kk, kkbb
    for j = 1, N
     Z(i,j) += X(i,k)*Y(k,j)
    endfor
   endfor
  endfor
 endfor
endfor
```
(e)

```
for ii = 1, N, B
 iibb = min(N,ii+B-1)
 ioread Z[ii:iibb,1:N]
 for kk = 1, N, B
  kkbb = min(N,kk+B-1)
  ioread X[ii:iibb,kk:kkbb]
  ioread Y[kk:kkbb,1:N]
  for i = ii, iibb
   for k = kk, kkbb
    for j = 1, N
     Z(i,j) += X(i,k)*Y(k,j)
    endfor
   endfor
  endfor
 endfor
 iowrite Z[ii:iibb,1:N]
endfor
```
(f)

Figure 1. (a) Matrix-multiply nest (b) Locality-optimized version (c) Tiled version (d) Tiled version with I/O calls (e) I/O-conscious tiled version (f) I/O-conscious tiled version with I/O calls.

although array `Z` has temporal reuse in the innermost loop (the `k` loop), and successive iterations of this loop access consecutive elements from array `X` (i.e., array `X` has spatial reuse in the innermost loop), the successive accesses to array `Y` touch different rows of this array; this is *not* a good style of access for a row-major array. Fortunately, using state-of-the-art optimizing compiler technology [31, 48], we can derive the code shown in Figure 1(b), given the code in Figure 1(a). In this so called optimized code, the array `X` has temporal reuse in the innermost loop (the `j` loop now) and the arrays `Z` and `Y` have spatial reuse, meaning that the successive iterations of the innermost loop touch consecutive elements from both the arrays.

However, unless the faster memory in question is large enough to hold the entire $N \times N$ array `Y`, many elements of this array will probably be replaced (in the case of caches and virtual memory systems) from the faster memory before they are reused in successive iterations of the outermost `i` loop. Instead of operating on individual array elements, tiling achieves reuse of array sections by performing the calculations (in our case matrix-multiply) on array sections (in our case sub-matrices). Figure 1(c) shows the tiled version of Figure 1(b). This tiled version is from [14]. In this tiled code, the loops `kk` and `jj` are called the *tile loops*, whereas the loops `i`, `k`, and `j` are the *element loops*. Here, it is important to choose the *blocking factor* `B` such that all the $B^2 + 2NB$ array items accessed by the element loops `i`, `k`, `j` should fit in the faster memory. In other words, the tiled version of the matrix-multiply code operates on $N \times B$ sub-matrices of arrays `Z` and `X`, and a $B \times B$ sub-matrix of

array Y at one time. Assuming that the matrices in this example are in main memory to begin with, ensuring that $B^2 + 2NB$ array elements can be kept in cache will be sufficient to obtain high levels of performance. In practice, however, depending on the cache size, cache associativity, and absolute array addresses in memory, *cache conflicts* [21] can occur. Consequently, the blocking factor B is set to a much smaller value than necessary [14, 29].

One important issue that needs to be clarified is *how* to select the loops that are to be tiled. A simple solution would be, as long as it is legal to do so (i.e., data dependences [49] allow it), to tile all the loops in the nest. However, this straightforward approach may result in unnecessary tiling; it also leads to problems with imperfectly nested loops [26]. More sophisticated techniques attempt to tile only the loops that carry some form of reuse. For example, the *reuse-driven tiling* approach proposed by Xue and Huang [52] attempts to tile a given loop nest such that the outer untiled loops will not carry any reuse and the inner tiled loops will carry all the reuse and, will consist of as few loops as possible. Notice that unnecessary tiling only introduces extra loop control overhead. Although in the matrix-multiply nest, it might be both legal and beneficial to tile all the loops (as all three loops carry some kind of reuse), the outermost loop (the i loop) is not tiled in most published works (e.g., [14]), most probably due to some overhead concerns. We stress that since, after the linear locality optimizations (a step preceding tiling), most of the inherent reuse in the nest will be carried by the *innermost* loop, almost all tiling approaches tile the *innermost* loop, provided it is legal to do so.

To sum up, tiling improves locality in scientific loop nests by capturing data reuse in the inner *element* loops. Among the important issues in applying tiling are the choice of the loops to be tiled and the selection of a suitable blocking factor (tile size). However, a straightforward application of traditional tiling to I/O-performing loop nests may not be very effective, as shown in the following section.

3. Tiling for disk-resident data

3.1. The problem

At first glance, the traditional tiling method summarized above seems to be readily applicable to computations on disk-resident arrays as well. Returning to our matrix-multiply code, assuming that we have a total memory of size H that can be allocated to this computation and all the arrays are disk-resident, we only need to ensure that $B^2 + 2NB \leq H$. This tiling scheme is shown in Figure 1(d). The routine (ioread) is an I/O routine that reads a section of a disk-resident array from file on disk into main memory; (iowrite) performs a similar transfer in the reverse direction. For a two-dimensional disk-resident array U, U[a:b,c:d] denotes a section of (b - a + 1) $\times$ (d - c + 1) elements; the sections for higher-dimensional arrays are defined similarly. It should also be emphasized that since the sections are brought into main memory by explicit I/O commands, the conflict problem mentioned above for cache memories (and for virtual memories as well) does not arise here.

While such a straightforward adaptation of the state-of-the-art tiling leads to *data reuse* for the array sections brought into memory, it can cause *poor* I/O performance. The reason for this becomes obvious from Figure 2(a), which illustrates the layout of the arrays and representative sections from arrays (bounded by thick lines) that are active in memory at the same time. The lines with arrows within the arrays indicate the storage order—row-major in our case—and each circle corresponds to an array element. Let us take a closer look at how a section of array `Z` is read from the file into main memory. Since the array is row-major in file, in order to read the 16 elements shown in the section, it requires 8 I/O calls to the file system, each for only 2 consecutive array elements. Note that even though a state-of-the-art parallel I/O library (e.g., [12, 42], or [46]) allows us to read this rectangular array section using only a single high-level library call, since the elements in the section to be read are not entirely consecutive in file, it will still require 8 internal I/O calls for the said library to read the section. It should also be noted that the alternative of reading the whole array and sieving out the unwanted array elements is, in most cases, unacceptable due to huge array sizes in I/O-intensive codes. Similar situations also occur with arrays `X` and `Y`.

The source of the problem here is that the traditional tiling attempts to optimize 'what' to read into the faster memory, not 'how' to read it. While this does not cause a major problem for the cache–main memory interface, the high disk access times render '*how to read*' a real issue. A simple rule is to always read array sections in a *layout conformant* manner. For example, if the file layout is row-major, we should try to read as many rows of the array as possible in a single I/O call, constrained only by data dependences and available memory size. This technique has been successfully used in optimizing I/O-intensive parallel codes in the past [42, 44, 45]. In our matrix-multiply example, we have failed to achieve this due to the tiling of the innermost

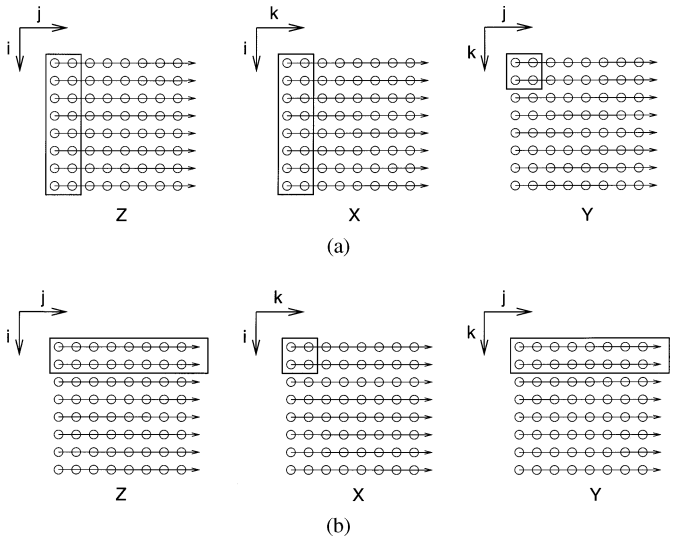


Figure 2. (a) Unoptimized and (b) Optimized tile access patterns for a matrix-multiply nest.

j loop, which reduces the number of elements that can be consecutively read in a single I/O call. Since this loop carries spatial reuse for both Z and Y, we should use this loop to read as many consecutive elements as possible from the said arrays. For example, instead of reading an $N \times B$ section from array Z, we should try to read a $B \times N$ section as shown in Figure 2(b) if it is possible to do so.

Two important points merit further discussion. First, we note from Figures 1(d) and 2(a), although the traditional tiling approach results in poor I/O performance, the locality for the array sections that are active in memory at a given time is quite good. For example, once the highlighted array section of Z (in Figure 2(a)) is read into memory, the innermost element loop j accesses this section along the (sub-rows) that are consecutive in memory. This good behavior is not surprising, as this is why the traditional tiling theory is developed: to optimize array accesses *in memory*. Our approach, as presented in the next subsection, *retains* this good memory behavior of traditional tiling while *improving* its poor I/O behavior significantly.

The second point concerns file layouts. One could argue in favor of retaining the original tiling pattern shown in Figure 1(d) and selecting non-traditional file layouts for arrays. For example, in Figure 2(a), instead of using row-major file layout for array Z and X, we could have used a so called *blocked layout* [4] so that all 16 array elements shown in the highlighted section would be stored in consecutive locations in file. While such an approach would be a definite improvement, we did not pursue it further in this paper for two reasons. First, as noted by Wolf [47] among others, although such a layout optimization technique may be implemented in a sophisticated optimizing compiler, scientific programmers are generally used to working with traditional layouts. For example, performance-conscious programming becomes tedious when different arrays have different blocked layouts; in addition, this severely reduces portability. In particular, complex array access functions that appear when blocked layouts are used may require non-trivial strength-reduction techniques [4]. Second, in some I/O-intensive applications the file layouts are inherited from a preceding computation (i.e., they are fixed); it might be prohibitively expensive to transform the layouts in files. Instead, our approach attempts to get most of the benefits of the blocked layouts by reconsidering carefully which loops to tile and by reading as many consecutive elements as possible from files in individual I/O calls, observing the limitations such as data dependences and limited main memory. Nevertheless, in our experiments (as presented in Section 5), we compare our approach to a hand-optimized approach, which uses blocked file layouts.

3.2. Our solution

Our solution to the tiling problem is as follows. As a first step, prior to tiling, the loops in the nest in question are reordered (or transformed) for maximum data locality in the innermost loop. While such an order can be obtained using an optimizing compiler such as SGI MIPSpro [39], experienced scientific programmers are also good at selecting appropriate loop orders for a given language and its default memory layout. In the second step, we tile all the loops by a blocking factor B except the *innermost* loop, which goes *untiled*. Since, after ordering the loops for

locality, the innermost loop will (hopefully) carry the highest amount of spatial reuse in the nest, by not tiling it, our approach ensures that the number of array elements read by individual I/O calls will be maximized.

As an application of this approach, we consider once more the matrix-multiply code given in Figure 1(a). After the first step, we obtain the code shown in Figure 1(b). After that, tiling the `i` and `k` loops and leaving the innermost loop `(j)` untiled, we reach the code shown in Figure 1(e). Finally, by inserting the `ioread` and the `iowrite` calls in appropriate places between tile loops, we have the final code given in Figure 1(f). The tile access pattern for this code is shown in Figure 2(b). We note that the section-reads for arrays `Z` and `Y`, and the section-writes for array `Z` are very efficient (given the fact that the file layouts are row-major). For example, as compared to the 8 calls required to read an $\mathtt{N} \times \mathtt{B}$ section of array `Z` in Figure 2(a), in Figure 2(b), a $\mathtt{B} \times \mathtt{N}$ section of the same array can be read by issuing only $\lceil 16/\mathtt{w} \rceil$ I/O calls where `w` is the maximum number of elements that can be read in a single I/O call. Of course, the figures given in this example (e.g., 8, 16) are for illustration purposes only. In some I/O-intensive applications, it is possible to reduce the number of I/O calls to as low as 1 per array section [23]. It should be stressed that the amount of memory used by the sections shown in Figures 2(a) and (b) is exactly the *same*.

It should also be noted that the I/O access pattern of array `X` is not as efficient as compared to arrays `Z` and `Y`. This is due to the nature of the matrix-multiply nest and may not be as important. Since the sections of array `X` are read much less frequently as compared to those of the other two arrays (because it has temporal reuse in the innermost loop), the impact of the I/O behavior of array `X` on the overall I/O performance of this loop nest will be less than that of the other two arrays. Of course, it is always possible to store array `X` in file such that the elements that reside in the same array section are stored in consecutive locations in file. As mentioned earlier, however, such a blocked storage scheme has its own problems.

Another important issue that we have not discussed so far is the placement of I/O calls in a given tiled nest. In fact, there are two subproblems here: (1) where are the I/O calls placed?; and (2) what are the parameters to the I/O calls? (i.e., what sections are to be read and written?) Determining the placement of I/O calls is relatively simple. For one, the I/O calls should definitely be outside the element loops, as these loops work on the array sections that are active in memory. Then, all we need to do is to look at the indices used in the subscripts of the reference in question, and insert the I/O call associated with the reference in between appropriate tile loops. For example, in Figure 1(e), the subscript functions of array `Z` use only loop indices `i` and `j`. Since there are only two tile loops, namely `ii` and `kk`, and only `ii` controls the loop index `i`, we insert the I/O read routine for this reference just after the `ii` loop, as shown in Figure 1(f). But, the other two references use the element loop index `k`, which varies with the tile loop index `kk`; therefore, we need to put the read routines for these references inside the `kk` loop (just before the element loops). The write routines are placed using similar reasoning.

For handling the second subproblem, namely, determining the sections to read and write, we can use the method of *extreme values of affine functions* first used by Banerjee [5, 49] for data dependence testing. Given an affine function of a

number of variables and inequalities that represent the bounds for the variables, the extreme values method determines the maximum and minimum values of the affine function in the bounded region. This method applies to non-rectilinear regions as well. We can describe this method using a simple example. Consider the affine function $f(i, j) = 6i - 2j + 1$ with the bounding region $\{(i, j) | 1 \leq i \leq 10 \text{ and } i + 1 \leq j \leq 30 - i\}$. Then,

For the upper bound	For the lower bound
$f(i, j) \leq 6i - 2j + 1$	$f(i, j) \geq 6i - 2j + 1$
$f(i, j) \leq 6i - 2(i + 1) + 1$	$f(i, j) \geq 6i - 2(30 - i) + 1$
$f(i, j) \leq 4i - 1$	$f(i, j) \geq 8i - 59$
$f(i, j) \leq 39$	$f(i, j) \geq -51$

Therefore, the upper bound of $f(i, j)$ is 39 and the lower bound is -51; that is, the values that the function f take on fall in the interval $-51 : 39$. We use a similar analysis to compute the range of array elements accessed by a complete execution of the *element* loops. Let us concentrate now on the reference `Z (i,j)` in Figure 1(e). We want to find the range of elements for each subscript function of this reference. Since `ii` $\leq$ `i` $\leq$ min(`N, ii + B - 1`), the range of elements accessed by the first subscript is `ii`: min(`N, ii + B - 1`). Similarly, the range of elements accessed by the second subscript is `1:N` as `1`$\leq$ `j`$\leq$ `N`. Consequently, the array section that needs to be read from a file for the reference `Z(i, j)` is `[ii`: min(`N, ii + B - 1), 1:N]`. The array sections for the other references are found using the same approach. For a more complex example, consider a reference such as `Z(i + j, k)` in a loop nest enclosed by the tile loops `ii` and `kk`, and by the element loops `i`, `k` and `j` (in that order) from the outermost to innermost. Assuming that `ii` $\leq$ `i` $\leq$ min(`N, ii + B - 1`), `kk` $\leq$ `k` $\leq$ min(`N, kk + B - 1`), and `1` $\leq$ `j` $\leq$ `N`, we need to read the array section `[ii + 1` : min(`N, ii + B - 1 + N, kk` : min `(N, kk + B - 1)]` for this reference.

In order to demonstrate the inefficiency of the traditional tiling approach using another example, we consider the successive-over-relaxation (SOR) kernel shown in Figure 3(a) assuming that the default file layout is *column-major*. The tiled version shown in Figure 3(b) is from Coleman and McKinley [14]. Assuming an available memory of size `H`, reading a `B` $\times$ `N` (`H`) array section requires `N` I/O calls. On the other hand, using the I/O-conscious tiling approach, we can obtain the code shown in Figure 3(c). Now, reading an `N` $\times$ `B` section requires only `B` I/O calls, assuming that in a single I/O call, *at most* `N` items can be read from file into memory. Of course, in this nest, due to multiple (uniformly generated [19]) references to the same array, we can only process `(N - 1)B` of the total `NB` elements transferred from a file. The issue of optimizing I/O accesses in the presence of stencil-like computations (taking boundary data into account) is orthogonal to the problem addressed in this paper and can be found elsewhere [7, 8].

```
for i = 2, N-1
 for j = 2, N-1
  Z(i,j)=α(Z(i+1,j)
      +Z(i-1,j)
      +Z(i,j+1)
      +Z(i,j-1))
 endfor
endfor
     (a)
```

```
for ii = 2, N-1, B
 for j = 2, N-1
  for i = ii, min(N,ii+B-1)
   Z(i,j)=α(Z(i+1,j)
      +Z(i-1,j)
      +Z(i,j+1)
      +Z(i,j-1))
  endfor
 endfor
endfor
     (b)
```

```
for jj = 2, N-1, B
 for j = jj, min(N,jj+B-1)
  for i = 2, N-1
   Z(i,j)=α(Z(i+1,j)
      +Z(i-1,j)
      +Z(i,j+1)
      +Z(i,j-1))
  endfor
 endfor
endfor
     (c)
```

Figure 3. The SOR kernel. (a) Original nest (b) Tiled nest (c) I/O-conscious tiled nest.

4. Automating the approach

4.1. Desired forms for array references

In the examples given so far, it was always straightforward to tile the loop nest. The reason for this is the presence of the innermost loop index in *at most* one subscript position of each reference. In those cases where the innermost loop index appears in *more than one* subscript position, we have the problem of determining the section shape. Consider a reference such as `Y(j + k, k)` to a disk-resident array `Y`, where `k` is the innermost loop index. Since, in our approach, we do not tile the innermost loop, when we try to read the entire bounding box that contains all the elements accessed by the innermost loop, we may end up having to read almost the whole array. Of course, this is not acceptable in general as in I/O-intensive routines, we may not have the luxury of reading the whole army into memory at one move. This situation occurs in the example shown in Figure 4(a). Assuming *row-major* layouts, if we tile this nest as is using the I/O-conscious approach, there will be no problem with array `Z`, as we can read a `B` $\times$ `N` section of it using the fewest number of I/O calls. However, for array `Y`, it is not trivial to identify the section to be read because the innermost loop index accesses the array in a diagonal fashion. The source of the problem is that this array reference does not fit our criterion, which assumes, at most, a single occurrence of the innermost loop index. In our example reference, the innermost loop index `k` appears in both the subscript positions.

```
for i = 1, N
 for j = 1, N
  for k = 1, N
   Z(i,k+j)=
     Y(k,i+j+k)
  endfor
 endfor
endfor
     (a)
```

```
for u =
 for v =
  for w =
   Z(u,v)=
     Y(u+v,v-w)
  endfor
 endfor
endfor
     (b)
```

```
for u =
 for v =
  for w =
   Z(u,v)=
     Y(u+v,v+w)
  endfor
 endfor
endfor
     (c)
```

```
for u =
 for v =
  for w =
   Z(u+w,v-w)=
     Y(v,u+v)
  endfor
 endfor
endfor
     (d)
```

```
for u =
 for v =
  for w =
   Z(u,w)=
     Y(u,u+v+w)
  endfor
 endfor
endfor
     (e)
```

Figure 4. (a) An example loop nest; (b–e) Transformed versions of (a).

In general, we want each reference to an m-dimensional *row-major* array Z to be in either of the following two forms:

$Z(f_1, f_2, \ldots, f_m)$: In this form, f_m is an affine function of all loop indices with a coefficient of 1 for the innermost loop index whereas f_1 through $f_{(m-1)}$ are affine functions of all loop indices except the innermost one.

$Z(g_1, g_2, \ldots, g_m)$: In this form, all g_1 through g_m are affine functions of all loop indices except the innermost one.

In the first case, we have *spatial reuse* for the reference in the *innermost loop*, and in the second case, we have *temporal reuse* in the *innermost loop*. Since, according to our I/O-conscious tiling, all loops except the innermost one will be tiled using a blocking factor B, for the arrays that fit in the first form, we can access $B \times \cdots \times B \times N$ sections (assuming N is the number of iterations of the innermost loop), which minimizes the number of I/O calls per array section. As for the arrays that fit into the second form, we can access sections of $B \times \cdots \times B \times B$, as all the loops that determine the section are tiled. Our task, then, is to bring each array (reference) to one of the forms given above. The good news is that, especially in linear-algebra kernels, the array references are generally in one of these forms to begin with. However, in banded-matrix codes, we have references with coupled-indices such as `Z(k, j + k)`, where k is the innermost loop index [31].

4.2. The algorithm

In the following, we propose a *compiler algorithm* to transform a loop nest such that the resultant nest will have references in one of the forms mentioned above. We assume that the compiler will determine the most appropriate file layouts for each array as well as a suitable loop (iteration space) transformation; that is, we assume that the layouts are *not* fixed as a result of a previous computation phase. Our algorithm, however, can be extended to accommodate those cases where file layouts are fixed at specific forms.

The file layout for an m-dimensional disk-resident array can be in one of $m!$ 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* [25]. As an example, for row-major file layout of a two-dimensional array, the second dimension is the fastest changing dimension. Thinking of each element as a sub-matrix, the method that will be presented can also handle the blocked file layouts. In other words, the methods presented in this paper are applicable, with appropriate modifications, to blocked layout cases as well. In the following, we assume that the transformed arrays will be stored in file(s) as row-major.

In our framework, each execution of an n-*deep* loop nest is represented using an *iteration vector* $\bar{I} = (i_1, i_2, \ldots, i_n)$, where i_j corresponds to jth loop from the outermost position. We assume that the army subscript expressions and loop

bounds are *affine functions* of enclosing loop indices and loop-index-independent variables. With this assumption, each reference to an m-*dimensional* array Z is represented by an *access (reference) matrix* L_z and an *offset vector* $\bar{o}_z$ such that $L_z\bar{I} + \bar{o}_z$ is the element accessed by a specific iteration $\bar{I}$ [49]. As an example, consider a reference Z(i + j, j - 1) to a two-dimensional array in a two-deep loop nest with i is the outer loop and j is the inner loop. We have,

$$L_z = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \quad \text{and} \quad \bar{o}_z = \begin{bmatrix} 0 \\ -1 \end{bmatrix}.$$

In general, if the loop nest is of depth n and the array in question is m-dimensional, the access matrix is of size $m \times n$ and the offset vector is m-dimensional.

The class of loop transformations we are interested in can be represented using non-singular square *transformation matrices*. For a loop nest of depth n, the iteration space transformation matrix T is of size $n \times n$. Such a transformation maps each iteration vector $\bar{I}$ of the original loop nest to an iteration $\bar{I}' = T\bar{I}$ of the transformed loop nest. Therefore, after the transformation, the new subscript function is $LT^{-1}\bar{I}' + \bar{o}$, meaning that the new access matrix is LT^{-1} [31, 49]. The problem investigated in works such as [48] and [31] is selection of a suitable T such that the locality of the reference is improved and all the data dependences in the original nest are preserved.

A data transformation is applied by transforming the dimensions (subscript expressions) of the reference. A square non-singular data transformation matrix M transforms the reference $L\bar{I} + \bar{o}$ to $ML\bar{I} + M\bar{o}$. Thus, the access matrix gets transformed to ML [24, 30, 35]. In this paper, we are interested in only permutation matrices for data transformations, i.e., we are only interested in dimension permutations.

Given an L matrix, our aim is to find matrices T and M such that the transformed access matrix will fit in one of our two desired forms as discussed above. Notice also that while T is unique to a loop nest, we need to find an M for each disk-resident array. A reference $L\bar{I} + \bar{o}$, becomes $MLT^{-1}\bar{I}' + M\bar{o}$ on the application of the loop transformation T and the data transformation M. Since, for a given L, determining both T and M simultaneously such that MLT^{-1} will be in a desired form involves solving a system of non-linear equations (not a trivial task), we solve it using a *two-step* heuristic approach. In the first step, we find a matrix T such that $L' = LT^{-1}$ will have a zero last column except for the element in an rth row, which is 1 (for spatial locality in the innermost loop) or we find a T such that the last column of $L' = LT^{-1}$ will be zero column (for temporal locality in the innermost loop). If the reference is optimized for spatial locality in the first step, in the second step, we find a matrix M such that this rth row in L' (mentioned above) will be the *last* row in $L'' = ML'$[1].

The overall algorithm is given in Figure 5. In Step 1, we select a *representative reference* for each array accessed in the nest. Using profile information might be helpful in determining run-time access frequencies of individual references. For each array, we select a reference that will be accessed a maximum number of times in a typical execution. This step is important as the layout of each array will largely be dictated by its representative reference.

INPUT: access matrices for the references in the nest.
OUTPUT: a loop transformation matrix T and a data transformation matrix M_i for each disk-resident array i

1: using profiling determine a representative access matrix for each array i ($1 \leq i \leq s$)
2: *for* each of the 2^s alternatives *do*
  2.1: determine target $L'_1, L'_2, \ldots, L'_s$
  2.2: using $L_i T^{-1} = L'_i$ determine a T
  2.3: *for* each array j with the spatial locality *do*
    2.3.1: let r_k be the row (if any) containing the only non-zero element in the last column for L'_i
    2.3.2: find an M_j such that $L''_j = M_j L'_j$ will be in the desired form (i.e., r_k will be the last row)
  2.4: *endfor*
  2.5: record for the current alternative the number of references with temporal locality, spatial locality, and no locality in the innermost loop
3: *endfor*
4: select the most suitable alternative (see the explanation in the text)
5: *I/O-conscious tile* the loop nest and insert the I/O read/write routines

Figure 5. I/O-conscious tiling algorithm.

Since (for each array) we have two desired candidate forms (one corresponding to temporal locality in the innermost loop and one corresponding to spatial locality in the innermost loop) and we have s arrays, in Step 2, we exhaustively try all 2^s possible alternatives, each corresponding to a specific combinations of localities (spatial or temporal) for the arrays.

Note that each possible locality combination implies a target (desired, optimal) access matrix for each array. In Step 2.1, we set the desired access matrix L'_i for each array i and in the next step, we determine a loop transformation matrix T that obtains as many desired forms as possible. This will allow us to optimize as many arrays as possible considering the current alternative. Note that Step 2.2 involves solving multiple equations for the entries of matrix T. A typical optimization scenario follows. Suppose that, in an alternative v where $1 \leq v \leq 2^s$, we want to optimize references 1 through b for temporal locality and references $b+1$ through s for spatial locality. After Step 2.2, we typically have c references that can be optimized for temporal locality and d references that can be optimized for spatial locality, where $0 \leq c \leq b$ and $0 \leq d \leq (s-b)$. This means that a total of $s-(c+d)$ references (arrays) will have *no locality* in the *innermost* loop. It should be emphasized that we do not apply any data transformations for the c arrays that have temporal locality in the innermost loop (as they are accessed infrequently and temporal locality is the best type of locality that we could obtain). For each array j (of maximum d arrays) that can be optimized for spatial locality, within the loop starting at Step 2.3, we find a data transformation matrix such that the resulting access matrix will be in our desired form. In Step 2.5 we record c, d, and $s-(c+d)$ for this alternative and move to the next alternative[2]. After all the alternatives are processed, we select the *most suitable* one. Although different techniques can be employed to select the most suitable alternative, we adopt the following strategy. First, for each alternative, we calculate $c+d$ (that is, the number of references with some form of locality in the innermost loop). If there is a *unique* alternative with the largest $c+d$ value, we select this alternative as the most suitable. Otherwise, if there are multiple alternatives with the largest $c+d$ value, we compare the respective c values

and select the one with the largest c value. If there are still multiple alternatives, we select the one that optimizes the LHS reference for the temporal locality. If there are still multiple alternatives, the choice is arbitrary. Although this selection mechanism slightly favors temporal locality over spatial locality and cannot guarantee the best solution for every loop nests, in practice, we have found it quite effective for the I/O-intensive loop nests encountered. It should be emphasized that by considering all types of localities for all arrays, our approach tries to catch the best (locality) combination.

There are three important points to note here. First, in completing the partially-filled loop transformation matrix T, we use the approach proposed by Bik and Wijshoff [6] to ensure that the resulting matrix *preserves* all *data dependences* [49] in the original nest. Second, we also need a mechanism (where necessary) to favor some arrays over others. The reason is that it may not always be possible to find a T such that all L'_i arrays targeted in a specific alternative are realized. In those cases, we need to omit some references from consideration, and attempt to satisfy (optimize) the remaining. Again, profile information can be used for this purpose. Third, even if an alternative does not specifically target the optimization of a reference for a specific locality, it may happen that the resultant T matrix generates such a locality for the said reference. In deciding the most suitable alternative, we need also take such (unintentionally optimized) references into account.

For an example application of the algorithm of Figure 5, consider the loop nest shown in Figure 4(a) and assume that optimizing array Z is more important than optimizing array Y. It is easy to see that there are no data dependences in this nest; thus, any arbitrary ordering of loop iterations is legal from the loop transformation point of view. The original access matrices are

$$L_z = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix} \quad \text{and} \quad L_y = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}.$$

We have four possible optimization alternatives here: (i) temporal locality for both arrays; (ii) temporal locality for Z and spatial locality for Y; (iii) temporal locality for Y and spatial locality for Z; and (iv) spatial locality for both arrays. These four alternatives result in the following loop/data transformation matrices and transformed access matrices.

- Alternative (i): Both $L_z T^{-1}$ and $L_y T^{-1}$ should have zero last columns. Since such a T is not possible, we try to have a zero last column for $L_z T^{-1}$ only. Then, a data transformation matrix (M_y) is applied to array Y to bring it to our desired form:

$$T^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & -1 \end{bmatrix}, \; M_y = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \Rightarrow L''_z = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

$$\text{and} \quad L''_y = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & -1 \end{bmatrix}.$$

- Alternative (ii): This alternative is very similar to Alternative (i). A data transformation is applied only for array Y:

$$T^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 1 \end{bmatrix}, M_y = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \Rightarrow L''_z = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

$$\text{and} \quad L''_y = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix}.$$

- Alternative (iii): $L_y T^{-1}$ should have zero last column. Then a data transformation matrix (M_z) is applied to array Z to bring it to our desired form. However, the form of this data transformation matrix here would necessitate extra disk space (i.e., it is not a permutation matrix) [30], therefore it is not applied. Consequently,

$$T^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 1 \end{bmatrix} \Rightarrow L''_z = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & -1 \end{bmatrix} \text{ and } L''_y = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix}.$$

- Alternative (iv): We need a loop transformation matrix such that both the references will be in our desired form for spatial locality. Therefore,

$$T^{-1} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}, M_x = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \Rightarrow L''_z = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \text{ and } L''_y = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}.$$

The resulting programs are shown in Figures 4(b)–(e) for alternatives (i), (ii), (iii), and (iv), respectively (the transformed loop bounds are omitted for clarity). It is easy to see that the alternatives (i) and (ii) are superior to the other two. Alternative (iii) cannot optimize array Z whereas alternative (iv) optimizes both arrays for spatial locality. Our algorithm chooses alternative (i) or (ii), as both ensure temporal locality for the LHS array and spatial locality for the RHS array in the innermost loop.

4.3. Multiple nests and inter-procedural problem

The technique proposed in the previous subsection to bring an array reference into the desired form to enable I/O-conscious tiling involves a data (file layout) transformation. Unlike loop transformations, the impact of a data transformation is *global* in the sense that it affects locality properties of all references to the same disk resident array in every loop nest and in every procedure. Therefore, the program-wide impact of a data transformation needs to be taken into account.

To handle this issue, when a data transformation matrix is applied to an array reference, it should also be applied to all other references to the same array whether they appear in the same nest or not. In the following, we briefly discuss how to

control this global effect. We can address this global file layout optimization problem at two levels: intra-procedural and inter-procedural.

First, we focus on the *intra-procedural* locality optimization problem and present an approach to optimize a series of loop nests collectively. Given a series of loop nests that access (possibly different) subsets of disk resident arrays declared in the procedure, our optimization strategy is as follows.

1. Transform the program into a sequence of independent loop nests using loop fusion [49], distribution [49], and code sinking [49].
2. Build an *interference graph* [3] and identify the *connected components.* The interference graph is a bipartite graph (V_n, V_α, E) where V_n, is the set of loop nests, V_a is the set of arrays, and E is the set of edges between loop nodes and array nodes. There is an edge $e \in E$ between $v_a \in V_a$ and $v_n \in V_n$ if and only if v_n references v_a.
3. For each connected component:
   - 3.1. Order the loop nests according to a *cost* criterion using profile information.
   - 3.2. Optimize the most costly (i.e., the most important) nest using the technique discussed in the previous subsection and then (I/O-conscious) tile this nest.
   - 3.3. For each of the remaining nests in the connected component (according to their cost orders):
     - 3.3.1. Optimize the nest being analyzed using the mentioned technique. However, in determining the loop transformation, first apply the data transformations found so far to respective references. Then, (I/O-conscious) tile the nest.
     - 3.3.2. Propagate the data transformations found so far to the remaining nests in the procedure.

In this algorithm, we first apply loop-level transformations to isolate as many nested loops as possible. The objective of this step is to make the subsequent analysis easier (by converting as many imperfect-nest as possible to perfectly-nested loops). We then build an interference graph to divide program into *disjoint segments*. Each segment consists of a group of nests and the arrays they operate on; *no* array is shared between two different segments. In the interference graph representation, each segment corresponds to a connected component. Since these segments do not share arrays, each of them can be optimized independently. Consequently, the rest of our approach works on a single segment at a time.

Our optimization strategy for each segment is based on the concept of the *most costly nest* (or *most important nest*). Intuitively, this is the nest that takes the most I/O time and, thus, it is the one that should definitely be optimized as much as possible. Different methods can be adopted to choose this nest. For example, profiling the sequential execution of the code might be useful in determining the most costly nest. Apart from this information, profiling can also help in determining some important parameters such as array size, loop bounds, and probabilities of conditional statements. Alternatively, the programmer can use compiler directives to give hints about this nest. We can also use different metrics (e.g., the number of loops in the nest or the number of arrays) to determine the most expensive nest.

After determining the most costly nest, the algorithm proceeds as follows. First, the most cost nest is optimized using the strategy given earlier for a single nest case. Note that, after this step, file layouts for some of the arrays will be determined. Then, each of the remaining nests can be optimized using the same approach except that we first need to apply a data transformation for each array whose file layout has been determined during the optimizition of the most costly nest. After each nest is optimized, new file layouts are obtained, and the associated data transformation matrices are propagated for the optimization of the next nest. Note that the order of processing for the remaining nests may also be important. If the number of nests is small, a more aggressive approach can apply this heuristic by considering each nest in turn as the most costly nest.

We now briefly discuss the *inter-procedural* locality optimization problem. It is easy to see that a naive approach that re-maps all disk-resident arrays across procedure boundaries can be prohibitively expensive and can easily out-weigh all gains obtained from I/O-conscious tiling.

Our current approach, instead, *propagates* data transformation matrices across procedures. It is similar in spirit to the global data distribution algorithm proposed by Anderson in her thesis [3] and can be summarized briefly as follows.

Our approach performs two traversals on the *call graph* representation of the program. A call graph $G_c = (V_c, E_c)$ is a multi-graph where each node $p_i \in V_c$ represents a procedure and there is an edge $e \in E_c$ between p_i and p_j if the former calls the latter [2]. In such a graph, the leaves represent the procedures that do not contain any calls. If desired, the edges can be annotated by some useful information related to call sites such as the actual parameters passed to the procedure, the line number where the call occurs, and so on. Currently, we do not handle programs that contain recursive procedure calls and we do not handle the arrays that are re-shaped across procedure boundaries.

Before the first traversal, we run an intra-procedural locality optimization algorithm (summarized above) on each leaf node. In the first traversal, called *bottom-up*, we start with the leaves and process each node in the call graph if and only if all the nodes it calls have been processed. After all the callee nodes for a given caller have been processed, we propagate a system of equalities (called *file layout* or *locality constraints*) to the caller. These equalities are such that, when solved, they give us loop and data transformations that collectively bring the references in the procedure being analyzed to our desired forms.

The caller adds this system to its own local set of equalities (obtained using the intra-procedural locality optimization algorithm) and propagates the resulting system to its callers, and so forth. When we reach the root (the main program), we have all the locality constraints of the program. We solve these constraints at the root and determine the data transformation matrices for the (global and local) arrays accessed by the root and the loop transformation matrices.

Afterwards, the *top-down* traversal starts; in this traversal, each caller propagates down the data transformation matrices determined so far to its callees. Using the equalities solved so far, the callees, in turn, determine the data transformation matrices for their local arrays as well as the loop transformations for the nests that

they contain. Then, they apply I/O-conscious tiling to these nests. When all the leaf nodes have been processed, the algorithm terminates.

5. Experimental results

In this section, we present experimental results obtained on an IBM SP-2 message-passing parallel architecture. Each node of SP-2 has an I/O subsystem containing two SCSI buses and six Starfire 7200 SCSI disks. Each node has 66.7 MHz clock speed and each SCSI disk has a minimum bandwidth of 8 MB/s.

We built our compiler optimizations upon a storage subsystem model called the *local placement model* (LPM) [7]. The main function of this model is to isolate the peculiarities of the underlying I/O architecture and present a unified platform on which experiment. Under this data storage subsystem, each *global disk-resident array* is divided into *local disk-resident arrays.* The local arrays of each processor are stored in separate files called *local array files* that in turn reside on a *logical local disk.* During the execution of an I/O-intensive program under the LPM, portions of local disk-resident arrays, called *data tiles*, are fetched and stored in local memory. The data sharing between processors is performed through explicit message passing; therefore, this system is a natural extension of the distributed-memory paradigm. We used an implementation of the LPM using the PASSION parallel I/O library on the IBM SP-2. Additional details of this implementation can be found in [7].

Our experimental methodology follows. We extracted a set of loops from several benchmarks and math libraries, as shown in Table 1. We used the PASSION library [42] to code I/O-intensive versions of these loop nests that do explicit file I/O for disk-resident datasets. The library performs several optimizations such as data sieving [42] and collective I/O [43, 42] to minimize the number of I/O calls to the underlying parallel file system (in our case, PIOFS). We conducted experiments with two different input sizes. In the first input size, called *small*, we set the sizes of all dimensions of all arrays to 2,048 double-precision elements (except for some small hard-coded dimensions). In the second input size, called large, a dimension was set to 4,096 double-precision elements. We also reduced the size of available memory of each processor to 1/128th of the size of its local disk-resident arrays to make the datasets appear really large. The loop nests were parallelized such that the entire interprocessor communication was eliminated (when necessary, array replication is also used to eliminate all communication). This allows us to concentrate solely on the I/O performance of the nests. Then, we generated three versions of each code. The first version, `orig`, is the I/O-intensive version of the original program that manipulates disk-resident arrays. In this version, only the loops that carry some form of reuse are tiled (following the tiling method proposed in [52]). The second version, `c-opt`, is the version obtained using the approach explained in this paper, and the third version, `h-opt` is an hand-optimized version using blocked file layouts. In obtaining the `c-opt` version, the tiled loop nests are generated automatically; however, the PASSION calls are inserted by hand. Our initial evaluation indicates that the amount of time required to generate the tiled nest is at most 8% of the original compilation time. Given the fact that optimizing large, I/O-intensive

Table 1. Programs used in the experiments

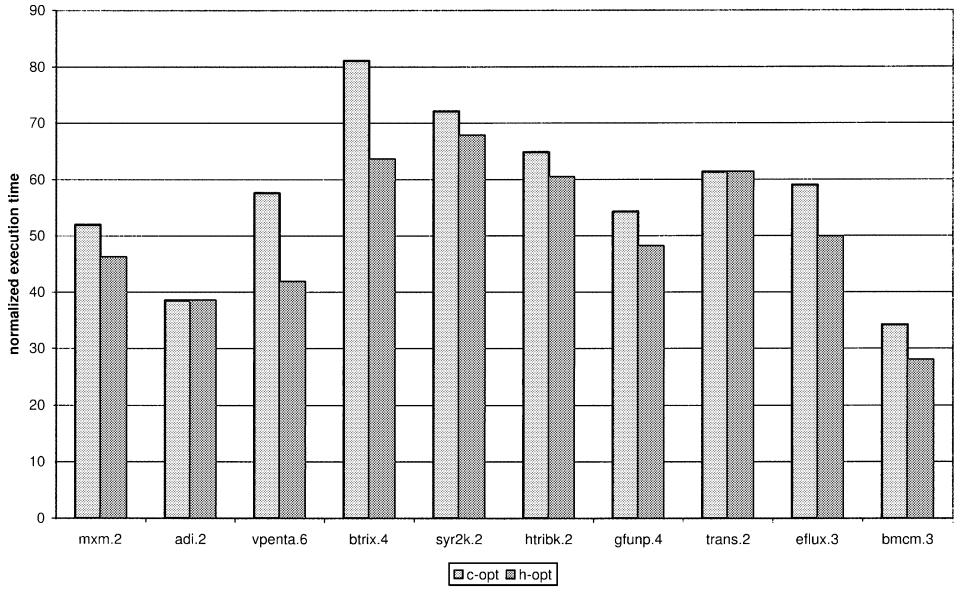
Program	Source	iter	Disk-resident arrays	I/O times	(small)	I/O times	(large)
mxm.2	specfp	4	three 2-D	114.35	(137.21)	234.01	(257.44)
adi.2	livermore	4	three 1-D, three 3-D	66.21	(85.36)	131.28	(150.90)
vpenta.6	specfp	2	seven 2-D, two 3-D	59.20	(301.92)	126.05	(642.85)
btrix.4	specfp	1	twenty-five 1-D, four 4-D	29.45	(176.82)	77.54	(441.00)
syr2k.2	blas	1	three 2-D	88.94	(97.86)	199.35	(239.25)
htribk.2	eispack	2	five 2-D	76.50	(84.15)	218.90	(240.71)
gfunp.4	hompack	2	one 1-D, five 2-D	41.20	(46.11)	79.50	(95.47)
trans.2	nwchem (PNL)	2	two 2-D	78.96	(101.20)	167.87	(210.66)
eflux.3	perfect club	1	four 3-D	69.03	(338.84)	155.61	(737.08)
bmcm.3	perfect club	2	five 1-D, one 2-D	44.70	(312.23)	93.32	(669.67)

*Note that the 'iter' column shows the number of times the outermost timing loop has been iterated for each code. The number at the end of the program name denotes the nest used. The last two columns show the execution times in *seconds* of the *original* codes with disk-resident arrays using two different input sizes. The numbers outside parantheses give the execution time of the nest (indicated in the first column) whereas the numbers inside parantheses give the execution time of the entire code (that encloses the corresponding nest).

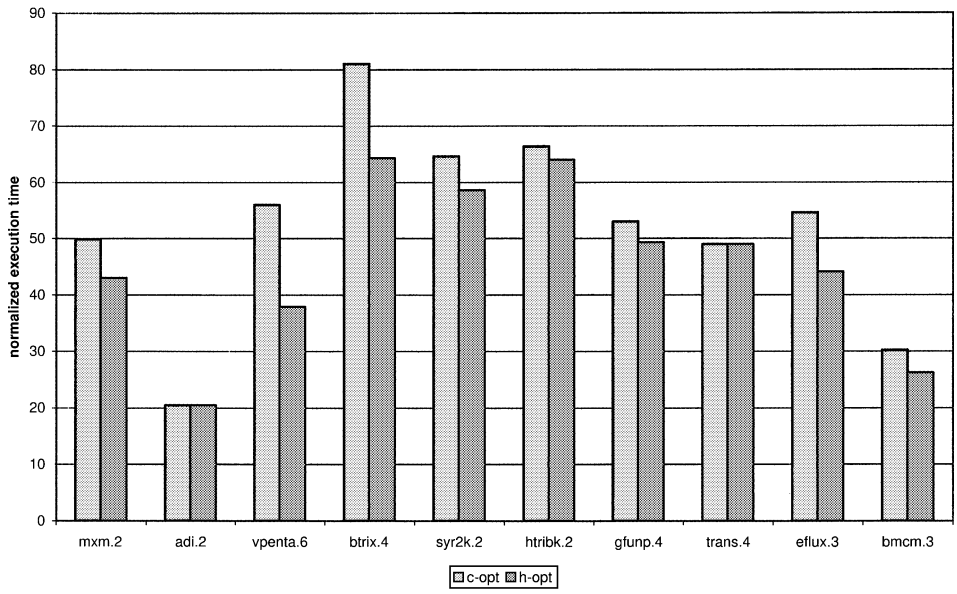
computations may bring huge performance savings at runtime, this small overhead in compilation time is reasonable.

The resulting codes are then compiled using the native compiler with the highest optimization level on the IBM SP-2. Performance numbers on 16 processors are shown in Figure 6 for both data sets. For each nest, the `c-opt` and `h-opt` bars give the respective execution times as a *fraction* of that of `orig` (see Table 1 for the original execution times). As an example, the execution time of the `c-opt` version of `trans. 2` with `small` data set is 61.3% of that of `orig`. We see that the `c-opt` version achieves on average a 42.5% reduction and a 47.4% reduction for the `small` and `large` input sizes, respectively. Using the hand-optimized version brings about a 6.9% (6.8%) additional reduction over `c-opt` for `small (large)` input sizes. In order to further understand the impact of the file layout optimizations, in Figure 7, we present the total number of I/O calls issued from within the application as a fraction of those of `orig`. Note that the numbers given in this figure are in *thousands.* We observe significant reductions in the number of I/O calls, which explains the gains in I/O times. Our approach is, in some cases, up to 81.1 percent better than the unoptimized code and the `h-opt` version brings up to an 18.2% additional improvement.

In addition to reducing the total number of messages and execution times, our optimizations improve scalability as well. Table 2 shows the speedup of each version of the nests for the *large* data sets. Speedup here is calculated with respect to the single processor time of the same version. For example, the execution time of `c-opt` for `adi.2` on a single processor is 29.4 times the execution time of the same version

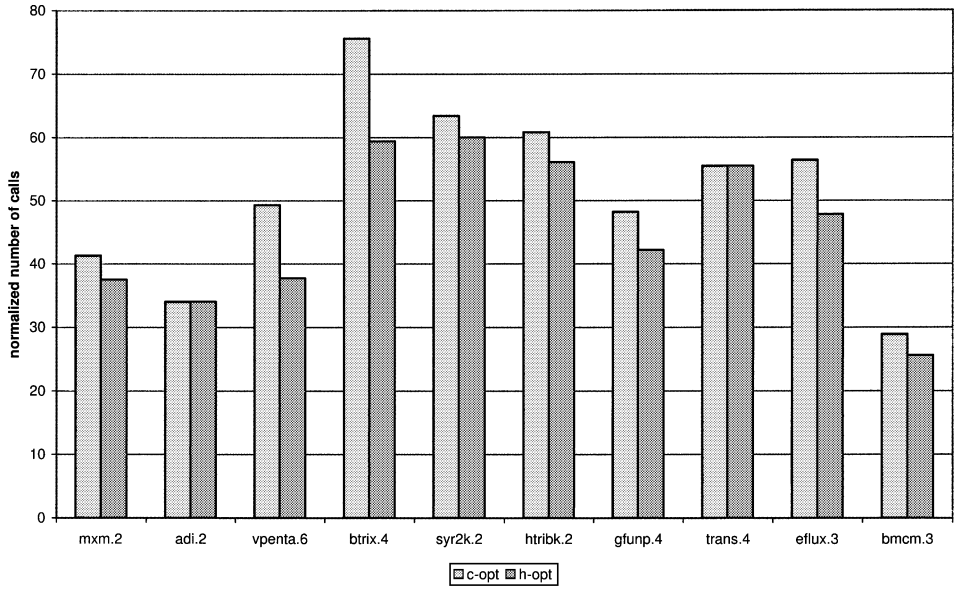


(a)

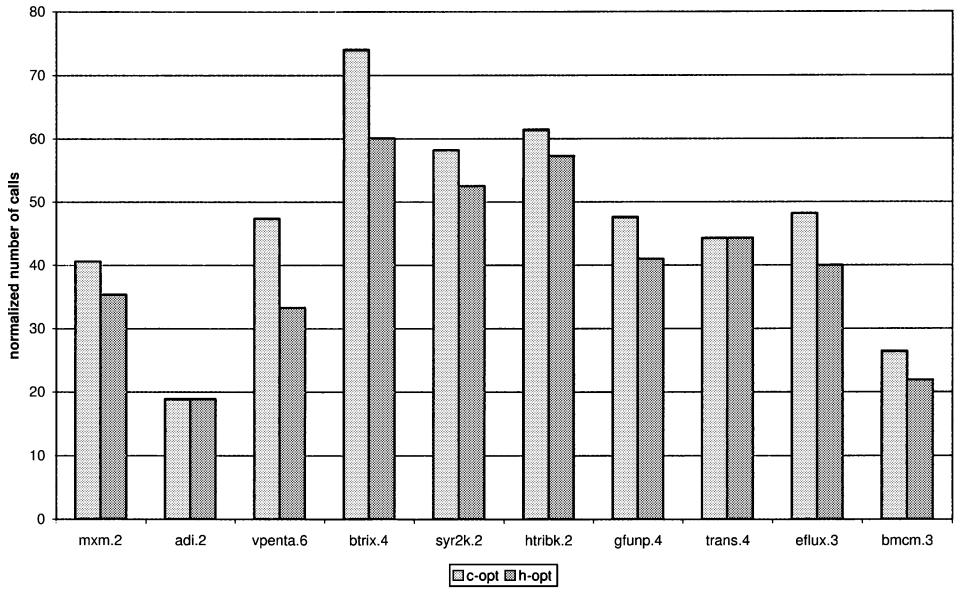


(b)

Figure 6. Normalized execution times for `small` input sizes (a) and `large` input sizes (b).



(a)



(b)

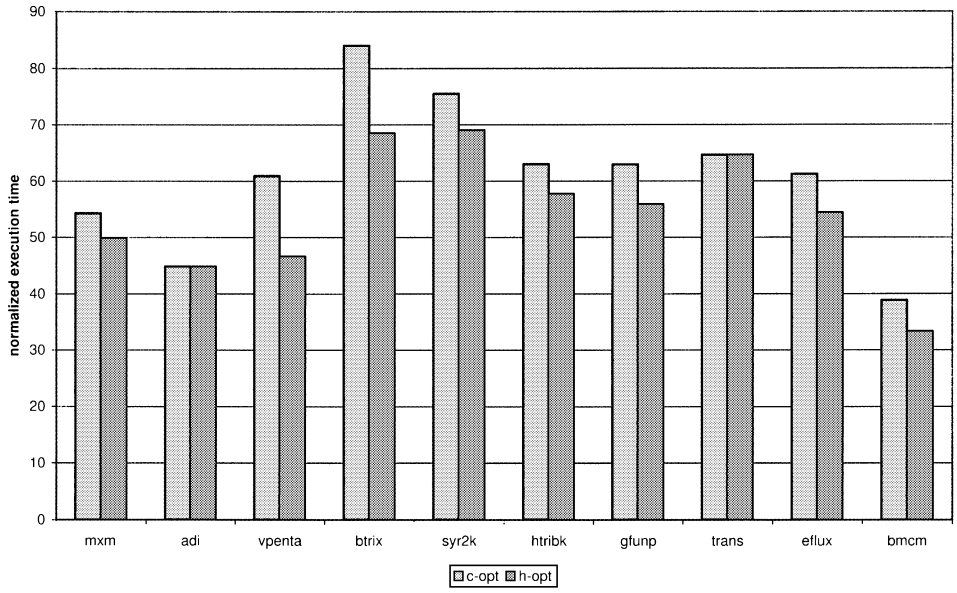
Figure 7. Normalized number of calls for `small` input sizes (a) and `large` input sizes (b).

Table 2. Speedup for different versions on the *large* input sizes

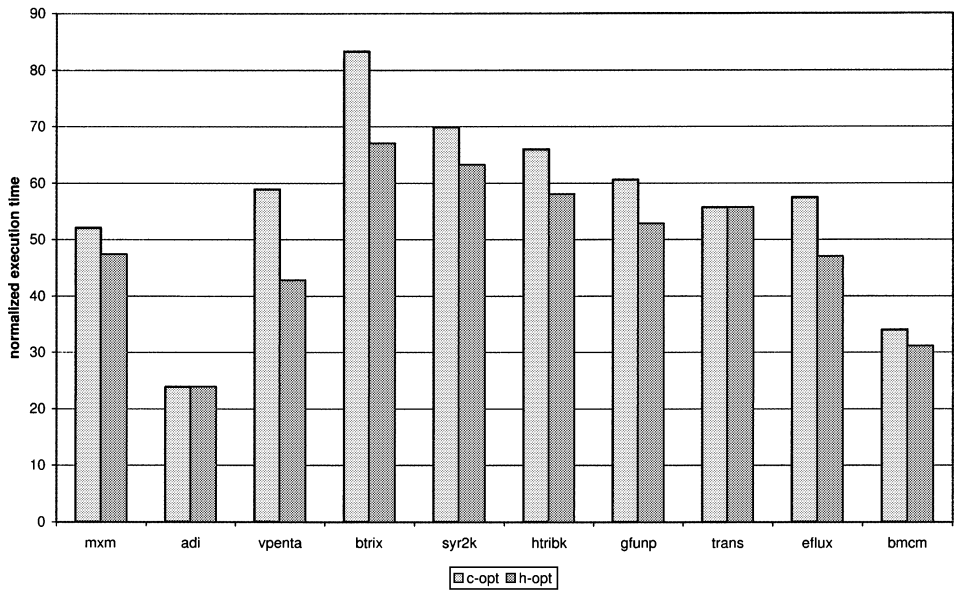
Program	Version	Number of processors		
		16	32	64
mxm.2	orig	10.9	20.2	38.7
	c-opt	13.3	25.1	56.0
	h-opt	13.7	24.8	55.9
adi.2	orig	12.4	22.0	53.4
	c-opt	15.0	29.4	60.4
	h-opt	15.0	29.4	60.4
vpenta.6	orig	10.0	24.2	51.3
	c-opt	14.7	28.6	62.0
	h-opt	14.9	29.9	62.4
btrix.4	orig	8.8	18.0	27.1
	c-opt	13.9	25.0	48.1
	h-opt	13.1	24.7	46.0
syr2k.2	orig	10.0	19.7	36.9
	c-opt	13.8	26.2	50.3
	h-opt	14.0	26.0	50.3
htribk.2	orig	12.7	21.9	38.7
	c-opt	12.9	22.5	38.9
	h-opt	13.5	22.7	40.0
gfunp.4	orig	10.6	20.4	38.4
	c-opt	13.2	24.9	56.1
	h-opt	13.4	24.6	57.0
trans.2	orig	12.0	20.5	35.8
	c-opt	15.6	31.0	62.2
	h-opt	15.6	31.0	62.2
eflux.3	orig	9.9	15.9	27.4
	c-opt	12.5	23.9	60.0
	h-opt	13.8	24.1	63.1
bmcm.3	orig	7.8	15.0	28.4
	c-opt	15.4	31.2	60.6
	h-opt	15.6	31.0	60.0

(c-opt) on 32 processors. We see from this table that the scalability of c-opt is much better than that of the original version, and in some cases, is even better than the hand-optimized code. It should be noted that since there is *no* interprocessor communication, the main factor limiting the scalability is the number of I/O nodes and the contention on the I/O network.

Finally, we investigated the effectiveness of our global approach (as explained in Section 4.3) on complete programs. This is important, as our approach involves data transformations (when we move beyond a single nest) and the impact of data transformations is global. Consequently, the effects of data transformations are spread throughout the program and evaluating their effects for single nests only may exaggerate their impact. Figure 8 gives the performance numbers on 16 processors for



(a)



(b)

Figure 8. Normalized execution times for `small` input sizes (a) and `large` input sizes (b) for the whole programs.

complete codes in our experimental suite for both data sets. As before, for each nest, the `c-opt` and `h-opt` columns give the respective execution times as a fraction of that of `orig`. We observe from these results that `c-opt` version achieves on average a 37.7% reduction and a 43.7% reduction for the `small` and `large` input sizes (over the original execution times given in Table 1), respectively. The hand-optimized version, on the other hand, reduces the execution times (on average) by 45.5% and 50.4% for the `small` and `large` input sizes, respectively. These results show that our approach is also successful for the multiple nest case and that it is able reduce the overall execution times significantly. In other words, we were successful in propagating the file layouts between the nests in a given program without much conflict between the optimal layout requirements of different loop nests.

6. Related work

Compiler researchers have attacked the locality problem from different perspectives. Wolf and Lam [47, 48], Li [31], McKinley et al. [33], Coleman and K. McKinley [14], Kodukula et al. [26], Lam et al. [29], and Xue and Huang [52], among others, have suggested tiling as a means of improving cache locality. In [48], [31], and [33], the importance of linear locality optimizations before tiling is emphasized. In this paper, we show that traditional tiling may not be very effective in optimizing I/O-intensive nests and propose an I/O-conscious tiling strategy.

The data layout transformation techniques proposed by O'Boyle and Knijnenburg [35], Cierniak and Li [13], Anderson et al. [4], Kandemir et al. [24, 25], and Leung and Zahorjan [30] involve the transformation of array layouts to improve spatial locality. O'Boyle and Knijnenburg [35] focus on restructuring the code given a data transformation matrix, although they show their method can be used for optimizing spatial locality. In comparison, we concentrate more on the problem of determining suitable data and loop transformation in a unified framework. Cierniak and Li [13] also propose a unified approach to optimize locality. Their approach employs both data space and iteration space transformations. The notion of the 'stride vector' is introduced and an optimization strategy is developed for obtaining the desired mapping vectors and loop transformation matrix. The `M` matrix that we use in this paper to obtain our desired subscript forms is an example of data layout transformation. In contrast to complex memory layouts, the transformations that are considered for file layouts should be rather simple. Previous work done by Kandemir et al. [23] on out-of-core compilation inspired the tiling strategy proposed in this paper.

In the I/O arena, there are many proposed software techniques for optimizing disk accesses. These techniques can be divided into three main groups: the parallel file system and run-time system optimizations [12, 15, 18, 27, 28, 42, 45], compiler optimizations [8, 9, 34, 36], and application analysis and optimization [32, 41, 44, 50, 51]. Brezany et al. [10] developed a run-time system called VIPIOS that can be used by an out-of-core compiler. Bordawekar et al. [7, 8, 9] focused on stencil computations that can be reordered freely due to lack of flow dependences. They present several algorithms to optimize communication and to improve the I/O performance of the parallel out-of-core applications. Paleczny et al. [36] incorporated out-of-core

compilation techniques into the Fortran D compiler. The main philosophy behind their approach is to choreograph I/O from disks along with the corresponding computation.

Cormen and Colvin [16] introduce ViC* (Virtual C*), a preprocessor that transforms a C* program that uses out-of-core data structures into a program with appropriate library calls from the ViC* library that read/write data from/to disks.

These techniques are all based on *reordering the computation* rather than on the *re-organization of data* in files. In contrast, we show that locality can be significantly improved using a combined approach that includes both loop and data transformations and then by employing an I/O-conscious tiling strategy. The work on prefetching for I/O [18, 34, 40], on the other hand, is complementary to the idea presented in this paper.

The early work on out-of-core computations was done by Abu-sufah, Kuck, and Lawrie [1] from the University of Illinois at Urbana-Champaign. They proposed optimizations to enhance the locality properties of programs in a virtual memory environment. In particular, they evaluated the gains obtained from tiling (page-indexing) and loop fusion. Our approach is different from theirs, as we rely on explicit I/O rather than leaving the job to the virtual memory management system; however, we believe that the I/O-conscious tiling strategy will be very useful in virtual memory based environments as well.

7. Conclusions and future work

In this paper, we present an I/O-conscious tiling strategy that can be used by programmers or can be automated in an optimizing compiler for I/O-intensive programs. We show that a straightforward extension of the traditional tiling strategy to I/O-performing loop nests may lead to poor performance and demonstrate the necessary modifications to obtain an I/O-conscious tiling strategy. Our experimental results (obtained on a set of scientific loop nests drawn from benchmarks and math libraries) reveal that our approach can improve the overall execution times, as well as the scalability of the I/O-performing loop nests.

Our current interest is in implementing this approach fully in a compilation framework and in testing it using larger programs such as the BTIO benchmark [11]. We plan to work on integrating this tiling strategy with the out-of-core optimizations suggested in [7, 9] and to evaluate the overall system in realistic I/O-intensive workloads on different distributed-memory architectures.

Acknowledgments

Mahmut Kandemir is supported in part by NSF grant CCR-0097998. Alok Choudhary is supported in part by NSF Young Investigator Award CCR-9357840, NSF grants CCR-9509143 and CCR-9796029, the Air Force Materials Command under contract F30602-97-C-0026, and the Department of Energy under the ACSI Academic Strategic Alliance Program Level 2, under subcontract No. W-7405-ENG-48

from Lawrence Livermore Labs. The work of J. Ramanujam is supported in part by NSF Young Investigator Award CCR-9457768 and NSF grant CCR-0073800.

Notes

1. This is assuming *row-major* layout. For *column-major* layouts, this row should be the *first* row.
2. A more aggressive approach can attempt to optimize the remaining $s - (c + d)$ arrays for spatial locality in the outer loops. Our current approach, instead, performs a naive I/O (i.e. accessing each sub-row of the region to be read/written using a separate I/O call) for the unoptimized arrays. Our experience and experiments show that, given the fact that our algorithm is able to optimize a great majority of references, this is a reasonable compromise.

References

1. W. Abu-sufah, D. Kuck, and D. Lawrie. On the performance enhancement of paging systems through program analysis and transformations. *IEEE Transactions on Computers,* C-30(5):341–356, 1981.
2. A. V. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*, 2nd ed. Addison-Wesley, Reading, MA, 1986.
3. J. Anderson. *Automatic Computation and Data Decomposition for Multiprocessors.* Ph.D. dissertation, Stanford University, March 1997. Also available as technical report CSL-TR-97-179. Computer Systems Laboratory, Stanford University.
4. J. Anderson, S. Amarasinghe, and M. Lam. Data and computation transformations for multiprocessors. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, July 1995.
5. U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, 1988.
6. A. Bik and H. Wijshoff. On a completion method for unimodular matrices. Technical report 94–14. Dept. of Computer Science, Leiden University, 1994.
7. R. Bordawekar. *Techniques for Compiling I/O Intensive Parallel Programs.* Ph.D. Dissertation, ECE Dept., Syracuse University, Syracuse, NY, May 1996.
8. R. Bordawekar, A. Choudhary, K. Kennedy, C. Koelbel, and M. Paleczny. A model and compilation strategy for out-of-core data parallel programs. *Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming*, pages 1–10, July 1995.
9. R. Bordawekar, A. Choudhary, and J. Ramanujam. Automatic optimization of communication in out-of-core stencil codes. In *Proceedings of the 10th ACM International Conference on Supercomputing*, pp. 366–373, May 1996.
10. P. Brezany, T. A. Mueck, and E. Schikuta. Language, compiler and parallel database support for I/O intensive applications. In *Proceedings of the High Performance Computing and Networking*, May 1995.
11. R. Carter, B. Ciotti, S. Fineberg, and B. Nitzberg, and B. Nitzberg. NHl-1 I/O benchmarks. Nas technical report, RND-92-016. Moffett Field, CA, November 1992.
12. Y. Chen, M. Winslett, Y. Cho, and S. Kuo. Automatic parallel I/O performance optimization in Panda. In *Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures*, June 1998.
13. M. Cierniak and W. Li. Unifying data and control transformations for distributed shared memory machines. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, June 1995.
14. S. Coleman and K. McKinley. Tile size selection using cache organization and data layout. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, June 1995.
15. P. Corbett, S. Baylor, and D. Feitelson. Overview of the Vesta parallel file system. In *Proceedings of the IPPS'93 Workshop on I/O in Parallel Computer Systems*, pp. 1–16, April 1993.
16. T. Cormen, and A. Colvin. ViC*: A preprocessor for virtual-memory C*. Dartmouth College Computer Science technical report PCS-TR94-243, November 1994.

17. J. del Rosario and A. Choudhary. High performance I/O for parallel computers: problems and prospects. *IEEE Computer*, 27(3):59–68, 1994.
18. C. S. Ellis and D. Kotz. Prefetching in file systems for MIMD multiprocessors. In *Proceedings of the 1989 International Conference on Parallel Processing*, pp. 1:306–314, St. Charles, IL, August 1989. Pennsylvania State Univ. Press.
19. 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.
20. Grand challenge applications projects. *http://www.npac.syr.edu/crpc/.*
21. J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*, 2nd ed. Morgan Kaufmann Publishers, 1995.
22. F. Irigoin and R. Triolet. Super-node partitioning. In *Proceedings of the 15th Annual ACM Symp. Principles of Programming Languages*, pp. 319–329, January 1988.
23. M. Kandemir, R. Bordawekar, and A. Choudhary. Data access reorganizations in compiling out-of-core data parallel programs on distributed memory machines. In *Proceedings of the International Parallel Processing Symposium*, Geneva, Switzerland, April 1997.
24. M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. A matrix-based approach to the global locality optimization problem. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, October 1998.
25. M. Kandemir, A. Choudhary, N. Shenoy, P. Banerjee, and J. Ramanujam. A hyperplane based approach for optimizing spatial locality in loop nests. In *Proceedings of the 1998 ACM International Conference on Supercomputing,* July 1998.
26. I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multilevel blocking. In *Proceedings of the SIGPLAN Conf. Programming Language Design and Implementation*, June 1997.
27. D. Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proceedings of the 1994 Symposium on Operating Systems Design and Implementation*, pp. 61–74. USENIX Association, Nov 1994.
28. D. Kotz. Multiprocessor file system interfaces. In *Proceedings of the Second International Conference on Parallel and Distributed Information Systems*, pp. 194–201. IEEE Computer Society Press, 1993.
29. M. Lam, E. Rothberg, and M. Wolf. The cache performance of blocked algorithms. In *Proceedings of the 4th Int. Conf. Architectural Support for Programming Languages and Operating Systems*, April 1991.
30. S.-T. Leung and J. Zahorjan. Optimizing data locality by array restructuring. Technical report TR 95-09-01, Dept. Computer Science and Engineering, University of Washington, Sept. 1995.
31. W. Li. *Compiling for NUMA Parallel Machines.* Ph.D. dissertation, Cornell University, 1993.
32. T. Madhyastha and D. A. Reed. Input/output access pattern classification using hidden Markov models. In *Proceedings of the Fifth Workshop on Input/Output in Parallel and Distributed Systems*, November 1997.
33. K. McKinley, S. Carr, and C. W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 1996.
34. T. Mowry, A. Demke, and O. Krieger. Automatic compiler-inserted I/O prefetching for our-of-core applications. In *Proceedings of the Second Symposium on Operating Systems Design and Implementations*, pp. 3–17, October 1996.
35. M. O'Boyle and P. Knijnenburg. Non-singular data transformations: Definition, validity, applications. In *Proceedings of the 6th Workshop on Compilers for Parallel Computers*, pp. 287–297, 1996.
36. M. Paleczny, K. Kennedy, and C. Koelbel. Compiler support for out-of-core arrays on parallel machines. In *Proceedings of the IEEE Symposium on The Frontiers of Massively Parallel Computation*, pp. 110–118, February 1995.
37. J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for multicomputer. *Journal of Parallel and Distributed Computing*, 16(2):108–120, 1992.
38. B. Rullman. Paragon parallel file system. *External Product Specification*, Intel Supercomputer Systems Division.
39. *SGI MIPSpro Fortran 77 Programmer's Guide*, SGI Corporation (also available using the InSight tool on the SGI Origin 2000 machines).
40. T. P. Singh and A. Choudhary. ADOPT: A dynamic scheme for optimal prefetching in parallel file systems. Technical report, NPAC, Syracuse, NY, June 1994.

41. E. Smirni, R. Aydt, A. Chien, and D. Reed. I/O requirements of scientific applications: An evolutionary view. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, August 6–9, 1996, pp. 49–59.
42. R. Thakur, A. Choudhary, R. Bordawekar, S. More, and S. Kuditipudi. PASSION: Optimized I/O for parallel applications. *IEEE Computer*, (26)6:70–78, 1996.
43. R. Thakur and A. Choudhary. An extended two-phase method for accessing sections of out-of-core arrays. *Scientific Programming*, (5)4:301–317, 1996.
44. R. Thakur, W. Gropp, and E. Lusk. An experimental evaluation of the parallel I/O systems of the IBM SP and Intel Paragon using a production application. In *Proceedings of the 3rd Int'l Conf. of the Austrian Center for Parallel Computation* (*ACPC*), September 1996.
45. R. Thakur, W. Gropp, and E. Lusk. On implementing MPI-IO portably and with high performance. Preprint *ANL/MCS-P732-1098*, Mathematics and Computer Science Division, Argonne National Laboratory, IL, October 1998.
46. S. Toledo and F. G. Gustavson. The design and implementation of SOLAR, a portable library for scalable out-of-core linear algebra computations, In *Proceedings of the Fourth Annual Workshop on I/O in Parallel and Distributed Systems*, May 1996.
47. M. Wolf. *Improving Locality and Parallelism in Nested Loops*. Ph.D. dissertation, Stanford University, Computer Systems Laboratory, August, 1992.
48. M. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN 91 Conf. Programming Language Design and Implementation*, pp. 30–44, June 1991.
49. M. Wolfe. *High Performance Compilers for Parallel Computing*, Addison-Wesley, 1996.
50. D. Womble, D. Greenberg, S. Wheat, and R. Riesen. Beyond core: Making parallel computer I/O practical. In *Proceedings of the Dartmouth Institute for Advanced Graduate Studies*, June 21–23, 1993.
51. D. Womble, D. Greenberg, R. Riesen, and S. Wheat. Out of core, out of mind: Practical parallel I/O. In *Proceedings of the Scalable Libraries Conference*, Mississippi State University, Oct 6–8, pp. 10–16, 1993.
52. J. Xue and C.-H. Huang. Reuse-driven tiling for data locality. In Z. Li et al., eds., *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pp. 16–33, vol. 1366. Springer-Verlag, 1998.