Proc. Workshop on Architecture Compiler Interaction, 3rd HPCA, San Anotnio, Feb. 1997

Optimizing Out-of-Core Computations in Uniprocessors*

M. Kandemir† A. Choudhary‡ J. Ramanujam§ R. Bordawekar¶

Abstract

Programs accessing disk-resident arrays perform poorly in general due to excessive number of I/O calls and insufficient help from compilers. In this paper, in order to alleviate this problem, we propose a series of compiler optimizations. Both the analytical approach we use and the experimental results provide strong evidence that our method is very effective on uniprocessors for out-of-core nests whose data sizes far exceed the size of available memory.

1 Introduction and Motivation

Due to the wide gap between processor and disk speeds in current parallel architectures, optimizing I/O performance presents a special challenge for designers of parallel systems. Current multicomputers either have virtual memory (VM) on each node as in Intel Paragon or do not have VM on nodes. Unfortunately, the studies of performance of VM using OSF/1 on Paragon have shown that the Paragon does not perform well on scientific programs due to frequent paging in and out of data [10]. Our experiments confirm this observeation as shown in Figure 1 for a typical out-of-core loop nest shown in Figure 3:A. Figure 1 shows the performance improvement (in terms of normalized execution times) obtained by explicitly-optimized I/O against VM on a single node of an Intel Paragon. Since the Intel Paragon on which the experiments have been conducted has 32 MBytes node memory, three double-precision floating-point arrays[1] each of which is $1K \times 1K$ represent the largest amount of data in our input set that can fit into memory. So, as expected, for arrays of size 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. Note that with the optimized approach, even for 512×512 and $1K \times 1K$ arrays, explicit I/O has been performed. This indicates that there is a clear need for optimized explicit I/O approach.

On the other hand, most of the current massively parallel systems such as the CM-5, iPSC/860, nCUBE-2 etc. do not have virtual memory on nodes. Currently, in order to program scientific applications which require large amounts of data, the programmer has to use explicit I/O calls to the underlying operating system. The main disadvantage of this approach is that it is requires low-level programming, and is tedious in addition to suffering from portability. The process of inserting explicit I/O calls is also error-prone and may require some knowledge about architectural parameters, if it is to be successful. We believe that high levels of performance in applications that work on disk-resident data can be achieved partly by an out-of-core compiler.

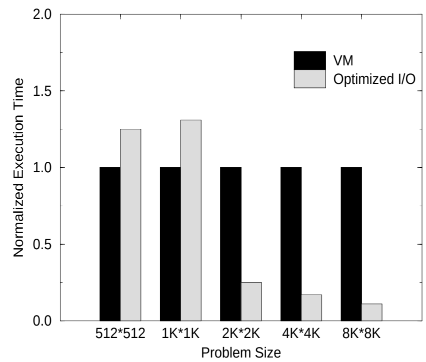

Figure 1: Optimized I/O vs. VM on a single node of Intel Paragon.

The problem addressed in this paper is to compile applications that use very large amounts of data. A computation is called *out-of-core* if the data used by it cannot fit in the memory; that is, parts of data should reside in files. A compiler for out-of-core computations should minimize the number as well as the volume of disk accesses. In this paper, we make the following contributions:

- We describe an abstract storage model based on *logical local disk* concept.
- We present a three-step compiler algorithm to optimize the locality on disks.
- We measure effectiveness of the algorithm on single nodes of IBM SP-2 and Intel Paragon.

The empirical results as well as the simulation and analytical results provide encouraging evidence that our algorithm is successful at reducing the time spent in disk I/O.

2 Abstract Storage Model

Deriving I/O optimizations for architectures with different disk subsystems can be a very difficult task. To alleviate this problem, we propose a storage subsystem model, called *local placement model* (LPM), which can be implemented on any multicomputer. The main function of this subsystem is to isolate the peculiarities of the underlying architecture and present a unified platform to experiment with. This subsystem specifies how the out-of-core arrays are placed on disks and how they are accessed by the processors. The local out-of-core arrays of each processor are stored in separate files called *local array files*. The local array file can be assumed

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

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

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

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

¶CACR, Caltech, Pasadena, CA 91125.

[1] For our experiments, we use PASSION library [12] which can associate different disk layouts for different arrays in C and Fortran.

to be *owned* by that processor. The node program explicitly reads from and writes into the local array file when required.

In other words we assume that each processor has its own *logical disk* with the local array files stored on that disk. The mapping of the logical disk to the physical disks is system dependent. Data sharing is performed by explicit message passing; so, this system is a natural extension of distributed-memory paradigm. During working of an out-of-core program under LPM, portions of out-of-core arrays are fetched and stored in memory. These portions are called *data tiles*. The node memory should be divided among tiles of different out-of-core arrays.

In order to implement the on IBM SP-2 we used the locally attached disk space to store the local array files. On the Intel Paragon, we created a separate file on the disk subsystem for each local array. We should note that under those implementations there is a strong parity between the LPM and the underlying I/O architecture on the SP-2, whereas there is a disparity between those on the Paragon.

3 Our Approach for Optimizing Locality on Disks

In this section we show analytically that our approach is very effective at reducing the I/O costs of out-of-core nests. To illustrate the significance of our I/O optimizations we first consider the example shown in Figure 2:A, assuming A and B are $n \times n$ out-of-core arrays and reside on (logical) disk. A straightforward translation of this loop nest by compiler is shown in Figure 2:B. In the translated code loops IT, JT and KT are called *tiling loops*. It should be emphasized that each array reference is replaced by its sub-matrix version. For example $A[IT, JT]$ denotes a sub-matrix (data tile) of the out-of-core matrix A and the size of this sub-matrix depends on bounds and step sizes of tiling loops. The computation enclosed by tiling loops implies the addition of two data tiles belonging to arrays A and B. We will show the bounds and the step sizes symbolically as the actual values they have are not significant for the purposes of this paper. However for all examples given in this paper, it is assumed that the loops in the original nest (such as Figure 2:A) iterate from 1 to n with unit stride.

Let $C_{I/O}$, $t_{I/O}$ and M be the startup cost[2] for a file read (write), cost of reading (writing) an element from (into) file, and memory size respectively. Suppose the memory is divided equally between A and B, and the out-of-core compiler works on square tiles of size $S \times S$ as shown in Figure 2:E. The overall I/O cost of the nest shown in Figure 2:B ($T_{overall}$) assuming *column-major* file layouts and considering *file reads* only can be calculated as follows.

$$T_{overall} = \underbrace{\frac{n^2 C_{I/O}}{S} + n^2 t_{I/O}}_{T_A} + \underbrace{\frac{n^3 C_{I/O}}{S^2} + \frac{n^3 t_{I/O}}{S}}_{T_B} \qquad (1)$$

where T_A is the cost of array A and T_B is the cost of array B with constraints $2S^2 \leq M$ and $0 < S \leq n$. By substituting $\sqrt{M/2}$ for S (to utilize the available memory as much as possible), we obtain

$$T_{overall} = \underbrace{\frac{\sqrt{2}n^2 C_{I/O}}{\sqrt{M}} + n^2 t_{I/O}}_{T_A} + \underbrace{\frac{2n^3 C_{I/O}}{M} + \frac{\sqrt{2}n^3 t_{I/O}}{\sqrt{M}}}_{T_B}$$

We have assumed that the cost of reading (writing) l consecutive elements from (into) a file can be approximated by $C_{I/O} + lt_{I/O}$ and that at most n elements can be accessed in a single I/O call. It is easy to see that in our example array B is more costly than array A, as it is used more frequently. Intuitively, instead of partitioning available memory equally between A and B, it would be wiser to allocate more memory to B in order to reduce the number of file accesses for this array. Since KT is in the innermost position in the loop, we can allocate memory of size Sn to $B[JT, KT]$ instead of a memory of size S^2. This allocation scheme is shown in Figure 2:F. The overall cost in this case is

$$T_{overall} = \underbrace{\frac{n^2 C_{I/O}}{S} + n^2 t_{I/O}}_{T_A} + \underbrace{\frac{n^3 C_{I/O}}{S^2} + \frac{n^3 t_{I/O}}{S}}_{T_B}$$

under the constraints $S^2 + nS \leq M$ and $0 < S \leq n$. It is clear that allocating more memory to B has a negative impact on the overall cost (due to stricter memory constraint). The reason is that since B is stored in column major order, as far as coefficient of $C_{I/O}$ is concerned it does not make much difference to read a *row* segment of size Sn into memory at once instead of reading segments of size S^2 in n/S steps. Also notice that allotting a tile of size nS for B would also be inefficient as a large portion of data brought into memory would be waiting without being processed for a long time.

Given the nest in Figure 2:B and column-major file layouts, an optimizing compiler can transform it to the nest shown in Figure 2:C[3]. By giving a memory of size nS to A we obtain

$$T_{overall} = \underbrace{\frac{n^2 C_{I/O}}{S} + n^2 t_{I/O}}_{T_B} + \underbrace{nC_{I/O} + n^2 t_{I/O}}_{T_A}$$

with constraints $nS + S^2 \leq M, 0 < S \leq n$. Since $S = \frac{\sqrt{n^2+4M}-n}{2}$,

$$T_{overall} = \underbrace{\frac{2n^2 C_{I/O}}{\sqrt{n^2+4M}-n} + n^2 t_{I/O}}_{T_B} + \underbrace{nC_{I/O} + n^2 t_{I/O}}_{T_A}$$

It is clear now that for the typical values of M ($n \leq M < n^2$), this cost is lower than the original. This memory allocation is shown in Figure 2:G. Notice also that since a tile of size nS is allocated for A, the tiling loop IT disappears.

So far the compiler has performed two optimizations. It has found an optimum loop order and allocated the available memory such that the overall I/O time of the nest has been reduced. Going one step further, suppose that the compiler is to create the out-of-core arrays A and B from scratch. Practically that means it can attach different file layouts for different arrays. In that case it can associate column-major file layout with A and row-major file layout with B, and then allocate a data tile of size nS for A and a data tile of size Sn for B (see Figure 2:H), changing the nest as shown in Figure 2:D. The overall I/O cost of this new loop order and allocation scheme,under our assumptions, is

$$T_{overall} = \underbrace{nC_{I/O} + n^2 t_{I/O}}_{T_A} + \underbrace{nC_{I/O} + n^2 t_{I/O}}_{T_B} \qquad (2)$$

Comparing (1) and (2), it is clear that our approach is very effective at reducing the number as well as the volume of disk accesses.

The out-of-core compiler automatizes the preceding optimizations. Specifically,

Step 1: it fisrt determines the most appropriate file layouts for each out-of-core array, then

[2] Startup cost can be thought as sum of the average values for seek time and rotational latency.

[3] Assuming that trip count is equal to the size of the array in the relevant dimension and step size is 1.

Step 2: permutes the tiling loops to optimize locality, and finally

Step 3: partitions the available memory among competing arrays based on *I/O cost estimation*.

Throughout the paper we assume that the file layout for an out-of-core array is either row-major or column-major. Unless otherwise stated, *loop* refers to a tiling loop.

4 I/O Optimizations

This section presents the details of our three-step algorithm. We also comment on possible extensions and interaction between the algorithm and prefetching which is another optimization. Our objective is to find file layout(s) and a loop order that will enable us to partition the memory among references such that the total I/O cost is minimized. The problem is that finding the best order requires an exhaustive search of all possible permutations of tiling loops. Moreover, for all of these permutations all possible memory allocations should be evaluated. Instead we propose a three-step heuristic to solve this problem [2, 3]. First we need a few definitions.

Definition: Assume a loop index IT, an array reference R with associated file *layout* and an array index position r. Also assume a data tile with size S in each dimension except r^{th} dimension where its size is n provided that $n = \Theta(N) \gg S$ where N is size of the array in r^{th} dimension. Then *Index I/O Cost* of IT with respect to R, *layout* and r is the number of I/O calls required to read such a tile from the associated file into memory, if IT appears in the r^{th} position of R; else *Index I/O Cost* is zero. Index I/O Cost is denoted by $ICost(IT, R, r, layout)$ where *layout* can be either row-major (*rowmajor*) or column-major (*colmajor*).

Definition: Assuming a file layout, the *Basic I/O Cost (BCost)* of a loop index IT with respect to a reference R is the sum of index I/O costs of IT with respect to all index positions of reference R:

$$BCost(IT, R, layout) = \sum_r ICost(IT, R, r, layout)$$

Note that the $BCost$ function can be used in two different ways: (1) If IT is fixed and (R,$layout$) pair is changed, it gives the I/O cost induced by loop index IT for different array layouts. (2) If (R,$layout$) pair is fixed and IT is changed over all loop indices in the nest, then it gives the I/O cost induced by R with the associated $layout$. The following definition employs the latter usage, while the former is used in Section 4.2.

Definition: *Array Cost (ACost)* of an array reference R, assuming a *layout*, is the sum of $BCost$ values for all loop indices with respect to reference R:

$$ACost(R, layout) = \sum_{IT} BCost(IT, R, layout)$$

4.1 Layout Determination Heuristic

This heuristic first computes the $ACost$ values for all arrays under all the possible layouts. It then chooses the combination that will allow the compiler to perform efficient file access. Consider the statement $A[IT, JT] = A[IT, JT] + B[JT, KT]$ in Figure 2:B. The Basic I/O costs for this statement are given in Table 1. Using these BCost values, array costs ($ACost$ values) for A and B can be found as in Table 2. Notice that $ACost$ values are listed term by term and each term corresponds to the $BCost$ of a loop index (IT,JT and KT in that order) under the given file layout. Next our heuristic considers all possible layout combinations by summing up

Table 2: ACost values for the program shown in Figure 2:B

Array	Layout	ACost	Array	Layout	ACost
A	colmajor	S+n+0	B	colmajor	0+S+n
A	rowmajor	n+S+0	B	rowmajor	0+n+S

the *ACost* values for the components of each combination term by term. Since in our example there are four possible combinations, the term by term additions of *ACost* values are as shown in Table 3.

Definition: The *Order* of a term is the greatest symbolic value it contains. For example the order of $(S + n)$ is n whereas the order of S is S. A term that contains neither n nor S is called *constant-order* term.

After listing all possible layout combinations term by term, our layout determination algorithm chooses the combination with greatest number of *constant-order* and/or *S-order* terms. The reason for this choice is that each loop index with only *S-order* and/or *constant-order* terms will enable the out-of-core compiler to read all the data along the relevant dimension(s) with low I/O cost. If there are more than one combination with maximum number of constant-order and/or S-order terms, our heuristic chooses the one with the minimum cost. It is easy to see from Table 3 that, for our example, column-major layout for array A and row-major layout for array B (Combination 2) is an optimum combination, since it contains two *S-order* terms (S and S). It should be noted that the optimum combination is not unique.

4.2 Loop Permutation Heuristic

This heuristic determines an optimal tiling loop order that will enable efficient file reading and writing.

Definition: The *Total I/O Cost (TCost)* of a loop index IT is the sum of the Basic I/O costs (*BCost*) of IT with respect to every distinct array references it surrounds. Generally speaking, $TCost(IT)$ is the estimated I/O cost caused by loop IT when all array references in the nest are considered.

$$TCost(IT) = \sum_{R, layout_R} BCost(IT, R, layout_R)$$

where R is the array reference and $layout_R$ is the layout of the associated file as determined in the previous step.

Our algorithm for desired loop permutation is as follows:

- Calculate $TCost(IT)$ for each tiling loop IT,
- Permute the tiling loops from outermost to innermost position according to non-increasing values of $TCost$[4], and
- Apply necessary loop interchange(s) to improve the temporal locality for (the tile of) the reference being updated. This step prevents write operation from occurring in inner tiling loops.[5]

Returning to our example under the chosen file layouts (Combination 2), $TCost(IT) = S$, $TCost(JT) = 2n$ and $TCost(KT) = S$. So, the desired loop permutations from outermost to innermost position are JT,IT,KT and JT,KT,IT.

[4] It should be noted that the desired loop permutation should observe the existing dependences. If it does not, our heuristic keeps the loop order as it is, and applies the memory allocation scheme only. Another option is to try the next desirable loop permutation. Our choice is simpler and guarantees that the optimized nest will be at least as good as the original one.

[5] Loop indices that do not have any n expression should be placed into innermost positions. Other loop indices however can be interchanged with one another if doing so promotes the temporal locality for the reference(s) being updated.

Table 1: BCost values for the program shown in Figure 2:B

Index	Reference	Layout	BCost	Index	Reference	Layout	BCost	Index	Reference	Layout	BCost
IT	A[IT,JT]	colmajor	S	JT	A[IT,JT]	colmajor	n	KT	A[IT,JT]	colmajor	0
IT	A[IT,JT]	rowmajor	n	JT	A[IT,JT]	rowmajor	S	KT	A[IT,JT]	rowmajor	0
IT	B[JT,KT]	colmajor	0	JT	B[JT,KT]	colmajor	S	KT	B[JT,KT]	colmajor	n
IT	B[JT,KT]	rowmajor	0	JT	B[JT,KT]	rowmajor	n	KT	B[JT,KT]	rowmajor	S

Table 3: Possible file layout combinations for the program shown in Figure 2:B

Combination	Array A	Array B	Cost
1	colmajor	colmajor	S+(n+S)+n
2	colmajor	rowmajor	S+2n+S
3	rowmajor	colmajor	n+2S+n
4	rowmajor	rowmajor	n+(S+n)+S

4.3 Memory Allocation Heuristic

Since memory capacity is limited and in general a loop nest may contain a number of out-of-core arrays, the available memory should be partitioned among the tiles of these out-of-core arrays optimally.

Definition: *Column-Conformant* (*Row-Conformant*) position of an array reference is the *first* (*last*) index position of the reference.

The memory allocation heuristic works as follows.

- The memory allocation algorithm starts with tiles of size S for each dimension in each reference. For example, if a loop nest contains a one-dimensional array, two two-dimensional arrays and a three-dimensional array, it 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 allotment scheme implies the memory constraint $S^3 + 2S^2 + S \leq M$.
- Then the compiler divides array references in the nest into two disjoint groups: a group whose associated files have row-major layout, and a group whose associated files have column-major layout. For column-major (row-major) layout group: The compiler considers all loop indices in turn. For each loop whose index appears in at least one column-conformant (row-conformant) position *and* does not appear in any other position of any reference in this group, it increases the tile size, in the column-conformant (row-conformant) position to full array size, for that reference.

  Of course, the memory constraint should be adjusted accordingly. It should be noted that, after these adjustments, any inconsistency between those two groups (due to a common loop index) should be resolved by not changing the original tile sizes in the dimensions in question.

For our example, the compiler first allocates a tile of size $S \times S$ for each reference. It then divides the array references into two groups: $B[JT, KT]$ in the first group, and $A[IT, JT]$ in the second group. Since KT appears in the row-conformant position of the first group and does not appear elsewhere in this group, our algorithm allocates a data tile of size Sn for $B[JT, KT]$. Similarly, since IT appears in the column-conformant position of the second group and does not appear elsewhere in this group, our algorithm allocates a data tile of size nS for $A[IT, JT]$ as shown in Figure 2:H. Notice that after these tile allocations tiling loops IT and KT disappear and the node program shown in Figure 2:D is obtained.

Table 6: Possible file layout combinations for the program shown in Figure 3:(B)

Combination	Array A	Array B	Array C	Cost
1	colmajor	colmajor	colmajor	(S+n)+n+(S+n)+S
2	colmajor	colmajor	rowmajor	(S+n)+n+2S+n
3	colmajor	rowmajor	colmajor	2S+n+2n+S
4	colmajor	rowmajor	rowmajor	2S+n+(S+n)+n
5	rowmajor	colmajor	colmajor	2n+S+(S+n)+S
6	rowmajor	colmajor	rowmajor	2n+S+2S+n
7	rowmajor	rowmajor	colmajor	(n+S)+S+2n+S
8	rowmajor	rowmajor	rowmajor	(n+S)+S+(n+S)+n

4.4 Different Layouts vs. Fixed Layouts

In this section, we show that relaxing disk layouts can save substantial amounts of time for some out-of-core nests. To illustrate this, consider the loop nest shown in Figure 3:A. Figure 3:B shows the simplified resulting node program (tiling loops) without any I/O optimization performed. The Basic I/O costs for this nest are given in Table 4 and the array costs for A, B and C are shown in Table 5. The compiler considers all possible layout combinations by summing up the *ACost* values for each combination term by term, obtaining costs for different layout combinations as shown in Table 6.

Assuming square data tiles of size S^2 for all arrays as default, without performing any optimization, the overall I/O cost is

$$T_{overall} = \underbrace{\frac{n^2 C_{I/O}}{S} + n^2 t_{I/O}}_{T_A} + \underbrace{\frac{n^3 C_{I/O}}{S^2} + \frac{n^3 t_{I/O}}{S}}_{T_B} + \underbrace{\frac{n^4 C_{I/O}}{S^3} + \frac{n^4 t_{I/O}}{S^2}}_{T_C}$$

under the memory constraint $3S^2 \leq M$. The tile allocations for the arrays are shown in Figure 3:D.

In the following we consider three possible I/O optimized compilation techniques:

- Assume a fixed column-major file layout for all arrays as in Fortran. In that case, $TCost(IT) = S + n$, $TCost(JT) = n$, $TCost(KT) = S + n$ and $TCost(LT) = S$ (from the first row of Table 6). So, from outermost to innermost position IT,KT,JT,LT and KT,IT,JT,LT are two desirable loop permutations. But considering the write (update) operation to the reference $A[IT, JT]$ it seems better to choose the order IT,JT,KT,LT without disturbing the spatial locality too much. In other words for a fixed column-major layout the initial loop order is the most appropriate one. Our memory allocation scheme allocates a tile of size nS for C, and tiles of size S^2 for each of A and B as shown in Figure 3:E.

DO i = lb_i, ub_i, s_i
  DO j = lb_j, ub_j, s_j
    DO k = lb_k, ub_k, s_k
      A(i,j)=A(i,j)+B(j,k)
    ENDDO k
  ENDDO j
ENDDO i

(A)

DO IT = lb_{IT}, ub_{IT}, s_{IT}
  DO JT=lb_{JT}, ub_{JT}, s_{JT}
    read data tile for A
    DO KT =lb_{KT}, ub_{KT}, s_{KT}
      read data tile for B
      A[IT,JT]=A[IT,JT]+B[JT,KT]
    ENDDO KT
    write data tile for A
  ENDDO JT
ENDDO IT

(B)

DO JT = lb_{JT}, ub_{JT}, s_{JT}
  read data tile for A
  DO KT =lb_{KT}, ub_{KT}, s_{KT}
    read data tile for B
    A[1:n,JT]=A[1:n,JT]+B[JT,KT]
  ENDDO KT
  write data tile for A
ENDDO JT

(C)

DO JT = lb_{JT}, ub_{JT}, s_{JT}
  read data tile for A
  read data tile for B
  A[1:n,JT]=A[1:n,JT]+B[JT,1:n]
  write data tile for A
ENDDO JT

(D)

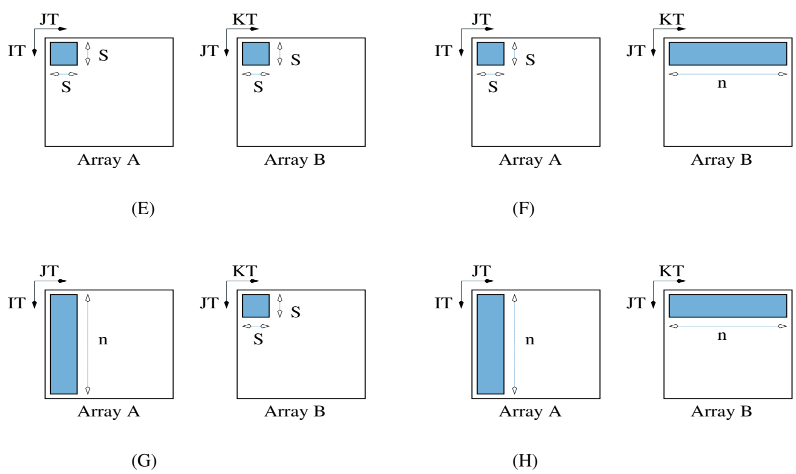

Figure 2: (A) An out-of-core loop nest (B) Straightforward translation of (A) (C) I/O optimized translation of (A) assuming column-major layouts only (D) I/O optimized translation of (A) assuming different file layouts (E) Tile allocations for (B) (F) Tile allocations for (B) (G) Tile allocations for (C) (H) Tile allocations for (D)

Table 4: BCost values for the program shown in Figure 3:(B)

Index	Reference	Layout	BCost	Index	Reference	Layout	BCost
IT	A[IT,JT]	colmajor	S	JT	A[IT,JT]	colmajor	n
IT	A[IT,JT]	rowmajor	n	JT	A[IT,JT]	rowmajor	S
IT	B[KT,IT]	colmajor	n	JT	B[KT,IT]	colmajor	0
IT	B[KT,IT]	rowmajor	S	JT	B[KT,IT]	rowmajor	0
IT	C[LT,KT]	colmajor	0	JT	C[LT,KT]	colmajor	0
IT	C[LT,KT]	rowmajor	0	JT	C[LT,KT]	rowmajor	0
KT	A[IT,JT]	colmajor	0	LT	A[IT,JT]	colmajor	0
KT	A[IT,JT]	rowmajor	0	LT	A[IT,JT]	rowmajor	0
KT	B[KT,IT]	colmajor	S	LT	B[KT,IT]	colmajor	0
KT	B[KT,IT]	rowmajor	n	LT	B[KT,IT]	rowmajor	0
KT	C[LT,KT]	colmajor	n	LT	C[LT,KT]	colmajor	S
KT	C[LT,KT]	rowmajor	S	LT	C[LT,KT]	rowmajor	n

Table 5: ACost values for the program shown in Figure 3:(B)

Array	Layout	ACost	Array	Layout	ACost	Array	Layout	ACost
A	colmajor	S+n+0+0	B	colmajor	n+0+S+0	C	colmajor	0+0+n+S
A	rowmajor	n+S+0+0	B	rowmajor	S+0+n+0	C	rowmajor	0+0+S+n

The overall I/O cost of this approach is

$$T_{overall} = \underbrace{\frac{n^2 C_{I/O}}{S} + n^2 t_{I/O}}_{T_A} + \underbrace{\frac{n^3 C_{I/O}}{S^2} + \frac{n^3 t_{I/O}}{S}}_{T_B} + \underbrace{\frac{n^3 C_{I/O}}{S^2} + \frac{n^4 t_{I/O}}{S^2}}_{T_C}$$

under the memory constraint $2S^2 + nS \leq M$. Notice that for reasonable values of M, this cost is better than that of the original.

- Assume a fixed row-major layout for all arrays as in C. In that case, $TCost(IT) = n+S$, $TCost(JT) = S$, $TCost(KT) = n+S$ and $TCost(LT) = n$ (from the last row of Table 6). So, from outermost to innermost position IT,KT,LT,JT and KT,IT,LT,JT are two desirable loop permutations. Going on with the first order, the compiler allocates a tile of size Sn for A, and tiles of size S^2 for each of B and C as shown in Figure 3:F. The overall I/O cost of this approach is

$$T_{overall} = \underbrace{nC_{I/O} + n^2 t_{I/O}}_{T_A} + \underbrace{\frac{n^2 C_{I/O}}{S} + n^2 t_{I/O}}_{T_B} + \underbrace{\frac{n^3 C_{I/O}}{S^2} + \frac{n^3 t_{I/O}}{S}}_{T_C}$$

  under the memory constraint $2S^2 + nS \leq M$. Notice that this cost is better than both that of the original nest and that of the one with column-major layout assumption.

- *Our Approach*: It is easy to see from Table 6 that row-major layout for arrays A and C and column-major layout for array B (Combination 6) is an optimum combination. With this combination, $TCost(IT) = 2n$, $TCost(JT) = S$, $TCost(KT) = 2S$ and $TCost(LT) = n$. So, desired loop permutation from outermost to innermost position is IT,LT,KT,JT. The compiler allocates a tile of size nS for B, and tiles of size Sn for each of A and C as shown in Figure 3:G. The overall I/O cost of this method is

$$T_{overall} = \underbrace{nC_{I/O} + n^2 t_{I/O}}_{T_A} + \underbrace{nC_{I/O} + n^2 t_{I/O}}_{T_B} + \underbrace{\frac{n^2 C_{I/O}}{S} + \frac{n^3 t_{I/O}}{S}}_{T_C}$$

  under the memory constraint $3nS \leq M$. Notice that this cost is much better from those of all previous techniques. The optimized loop order is shown in Figure 3:C. It should be noted that since the compiler allocates tiles of size Sn and nS the tiling loops KT and JT disappear.

4.5 Simulation Results

In order to evaluate the effectiveness of our approach at reducing the number of I/O calls, we have simulated the number of I/O calls for different problem sizes for the nest shown in Figure 3. We present in Figure 4:A and B the curves for $2K \times 2K$ and $4K \times 4K$ double arrays (in logarithmic form) respectively. It should be emphasized that the curves presented, especially for the Opt version, are overestimates in the sense that we assume at most n elements can be read in a single I/O call. The effectiveness of our approach in reducing the number of I/O calls is clearly reflected on these curves. For example, in Figure 4:B, with a memory size of 10^3, the number of I/O calls required for Ori, Col, Row and Opt versions are approximately 5×10^5, 2×10^5, 1.5×10^5 and 1.5×10^4 respectively.

4.6 Applicability

Our approach is applicable to loop nests that have arbitrary reuse spaces as well [13]. As an example consider $A[IT, JT] = A[IT, JT] + B[JT + LT, KT]$. Assuming column-major layouts for all arrays, $TCost(IT) = S$, $TCost(JT) = n + S$, $TCost(KT) = n$ and $TCost(LT) = S$.

We should emphasize that our three-step approach can be used in different ways. If the way data arrives (e.g. from archival storage, satellite or over the network) does not conform to the I/O optimized layout, the compiler either

- keeps the original layout, and applies only loop permutation and memory allocation heuristics, or
- redistributes data based on the result of the layout determination algorithm, and then applies the loop permutation and memory allocation heuristics.

If, on the other hand, the compiler is to create the out-of-core arrays from scratch, then it can apply all three steps of the proposed technique.

An important characteristic of our scheme is that all three steps are independent from each other and depending on the specific requirements of the application in hand, some combination of them can be applied. The version of our locality algorithm for distributed-memory message passing machines along with the related issues can be found in [2].

5 Discussion

5.1 Prefetching

Our algorithm effectively reduces the number of I/O calls and that in turn causes a considerable reduction in the disk bandwidth requirements of the application. This reduction combined with the improved reuse of data tiles in memory leads to a decrease in the total time spent in I/O. It can be argued that some of the remaining I/O time can be hidden if *compiler-directed prefetching* is applied after the algorithm presented here. The out-of-core compiler can issue the request to read data tile $x + 1$ and immediately start to compute on data tile x. In this way, reading of tile $x + 1$ from file overlaps the computation performed on tile x. Moreover, unlike prefetching for cache memories [7] or paged-memory systems [8], with a compilation scheme based on explicit file I/O, prefetching always helps. The reason is that in the explicit I/O, the software (compiler) is the sole controller of data transfers between disks and memory, and the loop nests we deal with make it possible to determine the addresses for the array references ahead of time and that in turn enables the compiler to issue I/O request for the next data tile. The only problem is that prefetching can, in some cases, increase the demand for the disk subsystem bandwidth significantly. As explained in [8], this problem can be eliminated by using more disks. Alternatively, a viable compiler solution for this problem might be prefetching the data tiles for *only* some arrays referenced in the nest instead of the naive approach which prefetches for all arrays. The problem of choosing the most appropriate data tiles to

DO i = lb_i,ub_i,s_i
  DO j = lb_j,ub_j,s_j
    DO k = lb_k,ub_k,s_k
      DO l = lb_l,ub_l,s_l
        A(i,j)=A(i,j)+B(k,i)+C(l,k)
      ENDDO l
    ENDDO k
  ENDDO j
ENDDO i

(A)

DO IT = lb_{IT},ub_{IT},s_{IT}
  DO JT=lb_{JT},ub_{JT},s_{JT}
    read data tile for A
    DO KT =lb_{KT},ub_{KT},s_{KT}
      read data tile for B
      DO LT =lb_{LT},ub_{LT},s_{LT}
        read data tile for C
        A[IT,JT]=A[IT,JT]+B[KT,IT]+C[LT,KT]
      ENDDO LT
    ENDDO KT
    write data tile for A
  ENDDO JT
ENDDO IT

(B)

DO IT = lb_{IT},ub_{IT},s_{IT}
  read data tile for A
  read data tile for B
  DO LT =lb_{LT},ub_{LT},s_{LT}
    read data tile for C
    A[IT,1:n]=A[IT,1:n]+B[1:n,IT]+C[LT,1:n]
  ENDDO LT
  write data tile for A
ENDDO IT

(C)

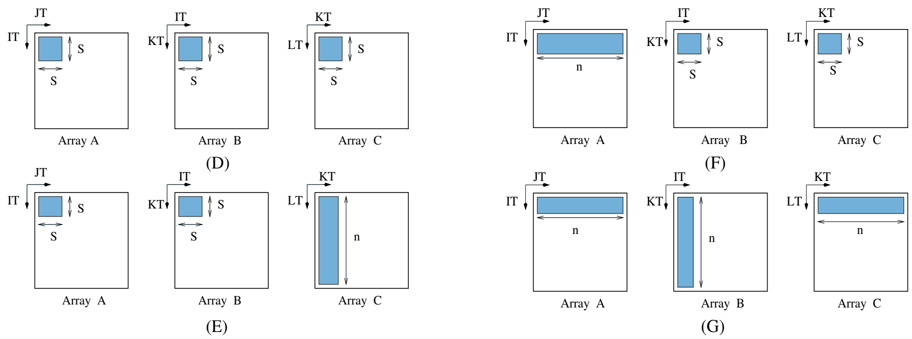

Figure 3: (A) An out-of-core loop nest (B) Straightforward translation of (A) (C) I/O optimized translation of (A) (D) Tile allocations for (B) (E) Tile allocations assuming column-major layout for all arrays (F) Tile allocations assuming row-major layout for all arrays (G) Tile allocations for (C)

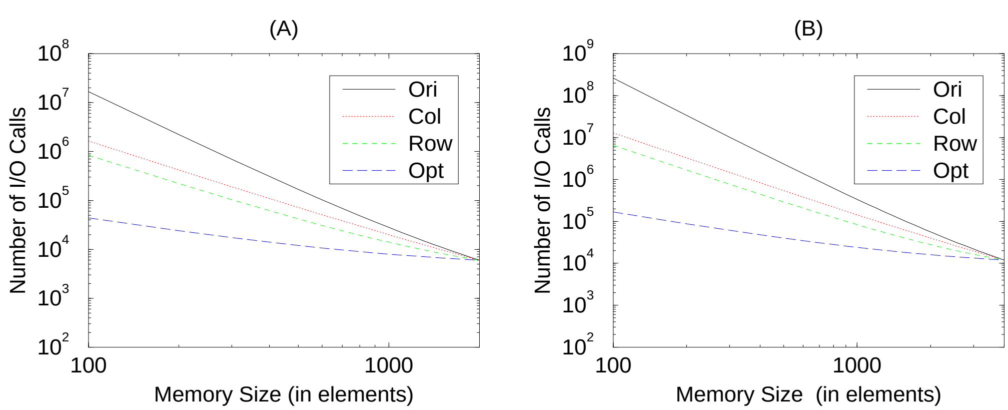

Figure 4: Number of I/O calls in the example shown in Figure 3 for (A) $2K \times 2K$ and (B) $4K \times 4K$ double arrays.

Table 7: I/O times for the first example (Figure 2).

SR	IBM SP-2				Intel Paragon			
	$2K \times 2K$		$4K \times 4K$		$2K \times 2K$		$4K \times 4K$	
	Ori	Opt	Ori	Opt	Ori	Opt	Ori	Opt
1/4	63	16	228	44	595	62	2005	91
1/16	191	21	482	50	704	70	2831	101
1/64	535	28	1228	62	1884	158	4480	162
1/256	2192	37	4110	82	6401	357	12809	385

prefetch considering hardware and software parameters, however, is beyond the scope of this work.

5.2 Possible Extensions

The approach discussed in this paper can be extended in several ways. First of all, our approach considers only loop permutations as the search space, limiting the number of possible loop transformations. Secondly, our current implementation considers two possible file layouts only: row-major and column-major. In fact, an m-dimensional array can be stored on disk in one of the $m!$ forms, each of which corresponding to layout of data on disk linearly by a nested traversal of the axes in some order. Although our algorithm can easily be modified to incorporate all possible layout combinations, this will increase the cost of file layout determination part substantially, as that step performs exhaustive search over all possible file layout combinations. Finally, in some out-of-core applications, it is not possible to change the file layouts for one or more arrays. Our algorithm can be enhanced to handle this case as well. After the table for the file layout combinations is obtained, the algorithm can take layout constraints into account. As an example, if, for Table 3, layout of array B is fixed as row-major, then for the optimal layout combination only combinations 2 and 4 need to be considered. The rest of the algorithm proceeds as before.

6 Experimental Results

In this section, we present the results of our experiments on IBM SP-2 and Intel Paragon. The experiments were performed for different values of *slab ratio* (SR), the ratio of available memory to the total size of out-of-core arrays. All reported times are in seconds.

Table 7 compares the I/O times of the nest shown in Figure 2:B (Ori) and Figure 2:D (Opt) on SP-2 and Paragon. Figure 5 presents the same information as normalized I/O times with respect to the Ori within each slab ratio. Table 8 shows the normalized I/O times for four different versions of the second example (Figure 3): Original version (Ori), I/O optimized version using column-major file layouts for all arrays (Col), I/O optimized version using row-major file layouts for all arrays (Row), and the version that is optimized by our approach (Opt). The same information is illustrated, as normalized with respect to the Ori, in Figure 6.

From these results we conclude the following:

- For the examples presented, the Opt version performs much better than the Ori version. The reason for this result is that in the Opt version array accesses are optimized as much as possible. For example as shown in Figure 3:G, by using the I/O optimization heuristic presented in this paper all three array accesses have been optimized.

- When the slab ratio is decreased, the effectiveness of our approach increases (see Figures 5 and 6). As the amount of memory is reduced, the Ori version performs many number

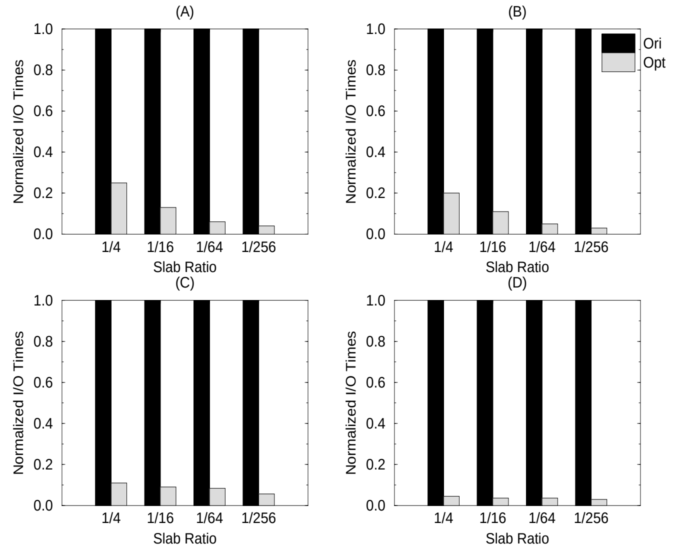

Figure 5: Normalized I/O times for the first example (Figure 2). (A) $2K \times 2K$ arrays on IBM SP-2 (B) $4K \times 4K$ arrays on IBM SP-2 (C) $2K \times 2K$ arrays on Intel Paragon (D) $4K \times 4K$ arrays on Intel Paragon.

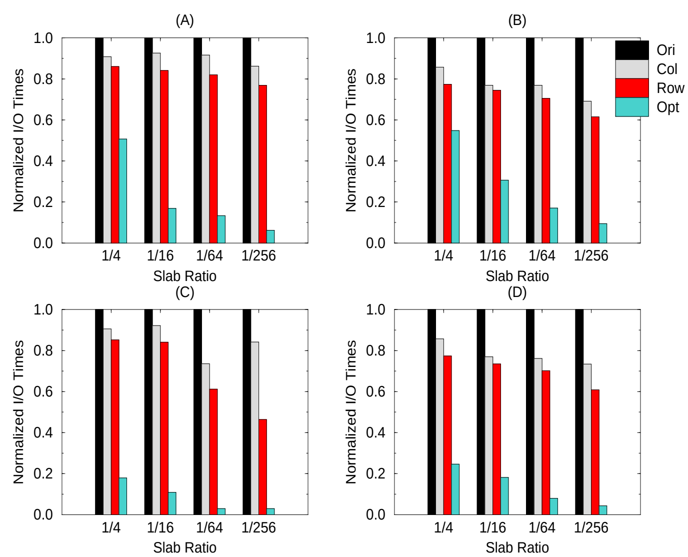

Figure 6: Normalized I/O times for the second example (Figure 3). (A) $2K \times 2K$ arrays on IBM SP-2 (B) $4K \times 4K$ arrays on IBM SP-2 (C) $2K \times 2K$ arrays on Intel Paragon (D) $4K \times 4K$ arrays on Intel Paragon.

Table 8: I/O times for the second example (Figure 3).

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

Table 9: Explicit I/O vs. prefetch with $2K \times 2K$ arrays in Intel Paragon.

SR	Opt	Pre	Overhead
1/4	62	44	3
1/16	70	48	3
1/64	158	96	4
1/256	357	210	5

of small I/O requests, and that in turn degrades the performance significantly. The Opt version on the other hand continues with optimized I/O no matter how small the memory is.

On Paragon, we have some unexpected results too. For example in Figure 5:C, when we decrease the slab ratio from $1/16$ to $1/64$, we do not observe any increase in the effectiveness of our approach. Similarly the effectiveness of our approach does not change when we move from $1/16$ to $1/64$ in Figure 5:D. It should be noted that in fact our approach is more effective on Paragon than on SP-2, but the problem is that the impact with the varying amount of in-core memory is not as predictable as that on SP-2. None of these unexpected results has occured on SP-2 due to the better match between the LPM and underlying local disk space on this machine.

- Demonstrations on two different platforms (Paragon and SP-2) with varying compile-time/run-time parameters such as available memory, array sizes etc., prove that our algorithm is quite robust.

We now focus on the impact of compiler inserted prefetch on I/O time. Table 9 illustrates the performance of prefetch for the nest shown in Figure 2:D with different slab ratios. $2K \times 2K$ double arrays are used and the experiments were conducted on a single node of Paragon. To keep the code simple, only data tiles of array A are prefetched. To make a fair comparison, we have reduced the available memory size for A by half when prefetch is applied. In other words, if a memory of size M_0 is allocated to array A in the nest shown in Figure 2:D (Opt), in the prefetched version a memory of size $M_0/2$ is allocated. The other half of the memory has been used as buffer for prefetching. The overhead incurred with the prefetched version is due to buffer copying, disk overhead of prefetch requests, and posting the prefetch requests.

Figure 7 shows the improvement in the I/O time through compiler inserted prefetching. Pre denotes the *visible* I/O time when the prefetch is applied. Both Pre and Overhead are normalized according to the Opt. It is observed that the overhead is negligible, and the speedup in I/O performance ranges from 1.41 to 1.70.

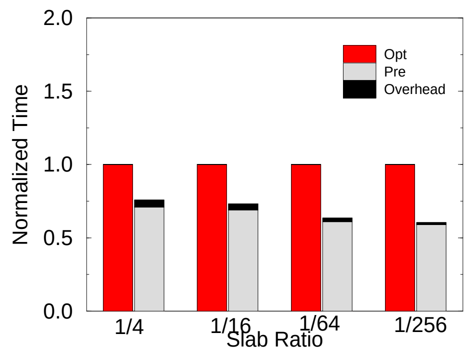

Figure 7: Visible I/O times and overhead of prefetch normalized with respect to optimized I/O time.

7 Related Work and Conclusions

Compiler-based optimization techniques have been offered in [4], [13], [7] and [6] for cache memories, and in [9], [8] and [11] for programs using disk-resident data. In [1] the functionality of ViC*, a compiler-like preprocessor for out-of-core C* is described. Output of ViC* is a standard C* program with the appropriate I/O and library calls added for efficient access to out-of-core parallel variables. A technique to allow the disk servers to determine the flow of data for maximum performance is presented in [5].

We have shown in this paper that unoptimized programs operating on disk-resident (out-of-core) arrays perform poorly due to the excessive number of I/O calls. I/O optimizations introduced in this paper are more effective on shared-disk systems, though in some cases the results may be unpredictable. These results suggest that in order to take full benefit from a parallel application working on disk-resident data, data sets should be distributed among the disks as evenly as possible. This will reduce the contention on I/O nodes.

We observed that for the shared-disk systems I/O optimizations are particularly critical. For the optimized programs, the network latency and the contention on I/O nodes are the main bottlenecks. As a result, for shared-disk systems we advocate the use of extremely efficient I/O nodes. Since some I/O nodes may be over- or under-utilized, we also suggest the techniques that allow co-operation between I/O nodes in servicing disk requests. In fact, it has been shown that those techniques led to much better performance than the traditional file systems [5]. Once the hardware/software techniques to eliminate the contention on the I/O nodes are employed, we believe, the approaches based on I/O optimizations are promising.

References

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

[2] M. Kandemir, R. Bordawekar and A. Choudhary. Data Access Reorganizations in Compiling Out-of-core Data Parallel Programs on Distributed Memory Machines", CACR Technical Report 129, October 1996. To appear in *IPPS'97*.

[3] M. Kandemir, R. Bordawekar and A. Choudhary, I/O Optimizations for Compiling Out-of-Core Programs on Distributed-Memory Machines. To appear in *PP97 (Eighth SIAM Conference for Parallel Processing for Scientific Computing)*.

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

[5] D. Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proceedings of the 1994 Symposium on Operating Systems Design and Implementation*, pages 61-74, November 1994.

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

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

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

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

[10] S. Saini and H. Simon. Enhancing Applications Performance on Intel Paragon through Dynamic Memory Allocation. In *Proceedings of The Scalable Parallel Libraries Conference*, Mississippi State University, October 1993.

[11] R. Thakur, R. Bordawekar, and A. Choudhary. Compiler and run-time support for out-of-core HPF programs. In *Proceedings of the 1994 ACM International Conference on Supercomputing*, pages 382-391, Manchester, UK, July 1994.

[12] R. Thakur, R. Bordawekar, A. Choudhary, R. Ponnusamy and T. Singh. PASSION Runtime Library for Parallel I/O, in *Proc. of the Scalable Parallel Libraries Conference*, October 1994.

[13] M. Wolf and M. Lam. A data Locality Optimizing Algorithm. in *Proc. ACM SIGPLAN 91 Conf.Programming Language Design and Implementation*, pages 30-44, June 1991.