# Data Access Reorganizations in Compiling Out-of-Core Data Parallel Programs on Distributed Memory Machines *

Mahmut Kandemir
CIS Dept., Syracuse University
Syracuse, NY 13244

Rajesh Bordawekar
CACR, Caltech
Pasadena, CA 91125

Alok Choudhary
ECE Dept., Northwestern University
Evanston, IL 60208-3118

## Abstract

*This paper describes optimization techniques for translating out-of-core programs written in a data parallel language to message passing node programs with explicit parallel I/O. We demonstrate that straightforward extension of in-core compilation techniques does not work well for out-of-core programs. We then describe how the compiler can optimize the code by (1) determining appropriate file layouts for out-of-core arrays, (2) permuting the loops in the nest(s) to allow efficient file access, and (3) partitioning the available node memory among references based on I/O cost estimation. Our experimental results indicate that these optimizations can reduce the amount of time spent in I/O by as much as an order of magnitude.*

## 1. Introduction

The use of massively parallel machines to solve large scale computational problems in physics, chemistry, and other sciences has increased considerably in recent times. Many of these problems have computational requirements which stretch the capabilities of even the fastest supercomputer available today. In addition to requiring a great deal of computational power, these problems usually deal with large quantities of data up to a few terabytes. Main memories are not large enough to hold this much amount of data; so data needs to be stored on disks and fetched during the execution of the program. Unfortunately, the performance of the I/O subsystems of massively parallel computers has not kept pace with their processing and communication capabilities. Hence, the performance bottleneck is the time taken to perform disk I/O.

In this paper we describe data access reorganization strategies for efficient compilation of out-of-core data parallel programs on distributed memory machines. In particular, we address the following issues, 1) how to estimate the I/O costs associated with different access patterns in out-of-core computations, 2) how to reorganize data on disks to reduce I/O costs, and 3) when multiple out-of-core arrays are involved in the computation, how to allocate memory to individual arrays to minimize I/O accesses.

*This work was supported in part by NSF Young Investigator Award CCR-9357840, NSF CCR-9509143 and in part by the Scalable I/O Initiative, contract number DABT63-94-C-0049 from Defense Advanced Research Projects Agency (DARPA) administered by US Army at Fort Huachuca.

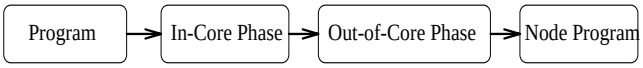


**Figure 1. Flow-Chart for Out-of-Core Compilation.**

The rest of the paper is organized as follows. Section 2 introduces our model and section 3 explains out-of-core compilation strategy. Section 4 discusses how I/O optimizations can reduce the I/O cost of loop nests. Section 5 presents experimental results. Section 6 discusses related work and concludes.

## 2. Model for Out-of-Core Compilation

In SPMD model, parallelism is achieved by partitioning data among processors. To achieve load balance, express locality of accesses and reduce communication, several distribution and alignment strategies are often used. Many parallel languages or language extensions provide directives that enable the expression of mappings from the problem domain to the processing domain. The compiler uses the information provided by these directives to compile global name space programs for distributed memory computers. Examples of parallel languages which support data distributions include Vienna Fortran [9] and HPF [4].

Explicit or implicit distribution of data to each processor results in each processor having a *local array* associated with it. For large data sets, local arrays cannot entirely fit in local memory and parts of them have to be stored on disk. We refer such local arrays *out-of-core local array*s. The out-of-core local arrays of each processor are stored in separate

1063 7133/97 $10.00 © 1997 IEEE

files called *local array file*s. We assume that each processor has its own logical disk with the local array files stored on that disk. If a processor needs data from any of the local array files of another processor, the required data will be first read by the owner processor and then communicated to the requesting processor.

## 3. Compilation Strategy

In order to translate out-of-core programs, the compiler has to take into account the data distribution on disks, the number of disks used for storing data etc. The portions of local arrays currently required for computation are fetched from disk into memory. These portions are called (*data*) *tiles*. Each processor performs computation on its tiles.

Figure 1 shows various steps involved in translating an out-of-core program consisting a single nest. The compilation consists of two phases. In the first phase, called *in-core phase*, the arrays in the source program are partitioned according to the distribution information and bounds for local arrays are computed. The second phase, called *out-of-core phase*, involves adding appropriate statements to perform I/O and communication. The local arrays are first tiled according to the node memory available in each processor. The resulting tiles are analyzed for communication. The loops are then modified to insert necessary I/O calls.

Consider the loop nest shown in Figure 2:(A) where $lb_k$, $ub_k$ and $s_k$ are the lower bound, upper bound and step size respectively for loop $k$. This nest will be translated by compiler into the node program shown in Figure 2:(B). In this translated code loops $IT$ and $JT$ are called *tiling loops*, and loops $IE$ and $JE$ are called *element loops*. Note that communication is allowed only at tile boundaries (outside the element loops). For sake of clarity, we will write this translated version as shown in Figure 2:(C). All communication statements and element loops will be omitted, and in **computation** part each reference will be replaced by its sub-matrix version.

## 4. I/O Optimizations

We first consider the example shown in Figure 3:(A), assuming that $A$, $B$ and $C$ are column-major out-of-core arrays.[1]

The compilation is performed in two phases as described before. In the in-core phase, using the array distribution information, the compiler computes the local array bounds and partitions the computation. In the second phase, tiling of the data is carried out using the information about available node memory size. The I/O calls to fetch necessary data tiles for $A$, $B$ and $C$ are inserted and finally the node program is generated. Figure 3:(B) shows the *straightforward* node program. Suppose that every processor has a memory of size $M$ and that the compiler works on square tiles of size $S \times S$. The overall I/O cost of the nest shown in Figure 3:(B) considering file reads alone is

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

under the memory constraint $3S^2 \leq M$. Here $C_{I/O}$ is the startup cost for a file access, $t_{I/O}$ is the cost of accessing an element from file and $p$ is the number of processors. We have assumed that cost of accessing $l$ consecutive elements from file can be approximated as $C_{I/O} + l t_{I/O}$. $T_A$, $T_B$ and $T_C$ denote the I/O costs for the arrays $A$, $B$ and $C$ respectively.

The proposed techniques transform the loop nest shown in Figure 3:(B) to the nest shown in Figure 3:(C), and associates row-major file layout for arrays $A$ and $C$ and column-major file layout for array $B$, and then allocates tiles of size $Sn$ for $A$ and $C$ and a tile of size $nS$ for $B$. The overall I/O cost of this new loop order and allocation scheme is

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

```
DO i = lb_i,ub_i,s_i
  DO j = lb_j,ub_j,s_j
    computation
  ENDDO j
ENDDO i
```

(A)

```
DO IT = lb_IT,ub_IT,s_IT
  DO JT = lb_JT,ub_JT,s_JT
    read tile from local file
    computation
    write tile into local file
  ENDDO JT
ENDDO IT
```

(C)

```
DO IT = lb_IT,ub_IT,s_IT
  DO JT = lb_JT,ub_JT,s_JT
    read tile from local file
    handle communication
    DO IE = lb_IE,ub_IE,s_IE
      DO JE = lb_JE,ub_JE,s_JE
        perform computation on tile
      ENDDO JE
    ENDDO IE
    handle communication
    write tile into local file
  ENDDO JT
ENDDO IT
```

(B)

**Figure 2. (A) Example Loop Nest. (B) Resulting Node Program. (C) Simplified Node Program.**

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

Proceedings of the 11th International Parallel Processing Symposium (IPPS '97)
1063-7133/97 $10.00 © 1997 IEEE

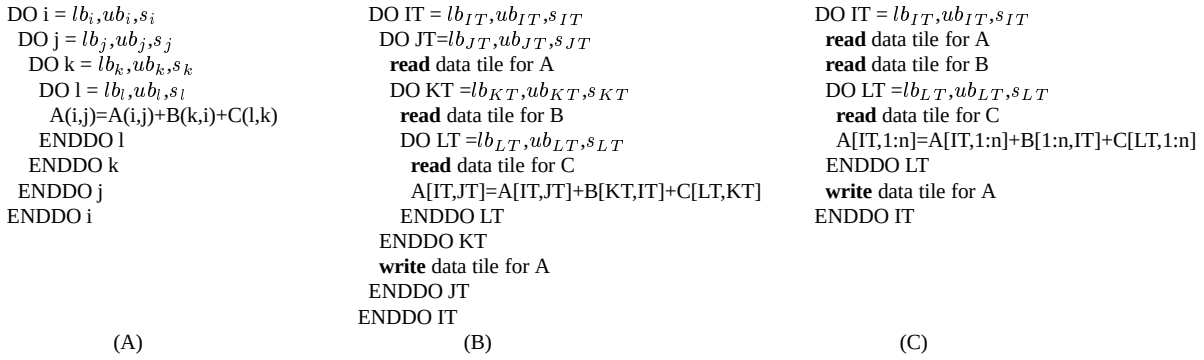


**Figure 3. (A) An out-of-core loop nest. (B) Straightforward translation. (C) I/O optimized translation.**

provided that $3nS \leq M$. Notice that this cost is much better than that of the original. Also note that in order to keep calculations simple we have assumed that at most $n$ elements can be requested in a single I/O call.

The rest of the paper explains how to obtain I/O optimized node programs. Our approach consists of three steps: (1) Determination of the most appropriate file layouts for all arrays referenced in the nest, (2) Permutation of the loops in the nest in order to maximize locality, and (3) Partitioning the available memory across references based on I/O cost.

We assume that the file *layout* for any out-of-core array may be either *row-major* or *column-major* and there is only one distinct reference per array.

*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 $ICost(IT, R, r, layout)$ [3].

*Definition:* The *Basic I/O Cost* 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 $R$. Mathematically speaking,

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

*Definition:* The *Array Cost* of an array reference $R$ is the sum of $BCost$ values for all loop indices with respect to reference $R$. In other words,

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

### 4.1. Determining File Layouts

Our heuristic for determining file layouts for out-of-core local arrays first computes the $ACost$ values for all arrays under possible layouts. It then chooses the combination that will allow the compiler to perform efficient file access. Consider the assignment statement in Figure 3:(B). The term by term additions of $ACost$ values for different combinations are shown in Table 1.

*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. For our example, combination 6 is an optimum combination, since it contains 2 *S-order* terms ($S$ and $2S$). Notice that there may be more than one optimum combination.

### 4.2. Deciding Loop Order

Our technique next determines an optimal loop order for efficient file access.

*Definition:* The *Total I/O Cost* of a loop index $IT$ is the sum of the Basic I/O costs (*BCost*) of $IT$ with respect to each distinct array reference it surrounds. Mathematically,

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

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 (1) calculates $TCost(IT)$ for each tiling loop $IT$, (2) permutes the tiling loops from outermost to innermost according to non-increasing values of $TCost$, and (3) applies necessary loop

**Table 1. Possible file layout combinations for our example.**

| Combination | Array A | Array B | Array C | Cost |
|---|---|---|---|---|
| 1 | colmajor | colmajor | colmajor | (S+n/p)+n+(S+n)+S |
| 2 | colmajor | colmajor | rowmajor | (S+n/p)+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/p+S+(S+n)+S |
| 6 | rowmajor | colmajor | rowmajor | 2n/p+S+2S+n |
| 7 | rowmajor | rowmajor | colmajor | (n/p+S)+S+2n+S |
| 8 | rowmajor | rowmajor | rowmajor | (n/p+S)+S+(n+S)+n |

interchange(s) to improve the temporal locality for the tile being updated.

Returning to our example, for combination 6, $TCost(IT) = 2n/p$, $TCost(JT) = S$, $TCost(KT) = 2S$, and $TCost(LT) = n$. The desired loop permutation from outermost to innermost is $LT$,$IT$,$KT$,$JT$, assuming $p \geq 2$.[2] Considering the temporal locality for the array being written to, the compiler interchanges $LT$ and $IT$, and obtains the order $IT$,$LT$,$KT$,$JT$.

### 4.3. Memory Allocation Scheme

Since each node has a limited memory capacity and in general a loop nest may contain a number of arrays, the memory should be partitioned optimally.

*Definition:* The *column-conformant* (*row-conformant*) position of an array reference is the *first* (*last*) index position of it.

Our scheme starts with tiles of size $S$ for each dimension in each reference. For example, if a loop nest contains a one-dimensional array, and a three-dimensional array, it first allocates a tile of size $S$ for the one-dimensional array, and a tile of size $S \times S \times S$ for the three-dimensional array. This allotment scheme implies the memory constraint $S^3 + S \leq M$ where $M$ is the size of the node memory. It then divides array references in the nest into two disjoint groups depending on the file layouts. For row-major (column-major) group, the compiler considers all loop indices in turn. For each loop whose index appears in at least one row-conformant (column-conformant) position *and* does not appear in any other position of any reference in this group, it increases the tile size, in the row-conformant (column-conformant) position(s) to full array size. Of course, the memory constraint should be adjusted accordingly. [3]

For our running example, the compiler first allocates a tile of size $S \times S$ for each reference. It then divides the array references into two groups: $A[IT, JT]$ and $C[LT, KT]$ in the first group, and $B[KT, IT]$ in the second group. Since $JT$ and $KT$ appear in the row-conformant positions of the first group and do not appear elsewhere in this group, our algorithm allocates data tiles of size $Sn$ for $A[IT, JT]$ and $C[LT, KT]$. Similarly, since $KT$ appears in the column-conformant position of the second group and does not appear elsewhere in this group, the algorithm allocates a data tile of size $nS$ for $B[KT, IT]$. After these tile allocations tiling loops $KT$ and $JT$ disappear and the node program shown in Figure 3:(C) is obtained.

If we assume fixed column-major file layout for all arrays, then $TCost(IT) = S + n/p$, $TCost(JT) = n$, $TCost(KT) = S + n$, and $TCost(LT) = S$ (from the first row of Table 1). So, from outermost to innermost position $KT$,$JT$,$IT$,$LT$ is the desirable loop permutation. Considering the temporal locality for the array being written to, the compiler interchanges $KT$ and $IT$, and the order $IT$,$JT$,$KT$,$LT$ is obtained. If, on the other hand, we assume a fixed row-major layout for all arrays, then $TCost(IT) = n/p + S$, $TCost(JT) = S$, $TCost(KT) = n + S$, and $TCost(LT) = n$. From outermost to innermost position $KT$,$LT$,$IT$,$JT$ is the desirable loop permutation. Considering the temporal locality, our compiler takes $IT$ to the outermost position. So, the final loop order is $IT$,$KT$,$LT$,$JT$. It should be emphasized that although for reasonable values of $M$ the costs obtained under the assumption of fixed disk layouts are better than that of the unoptimized version, they are much worse than the one obtained by our approach.

The complexity of our heuristics is $\theta(lmd+2^m+llog(l))$ where $l$ is the number of loops, $m$ is the number of distinct array references, and $d$ is the maximum number of array dimensions considering all references. The $log$ term comes from sorting once the $TCost$ value for each loop index has been computed. Since in practice $l$, $m$ and $d$ are very small (e.g. 2,3,etc.), all the steps are inexpensive and the approach is efficient.

It should also be noted that if the desired loop permutation is not legal (semantic-preserving), then the compiler keeps the original loop order and applies only the memory allocation algorithm[4].

## 5. Experimental Results

The technique introduced in this paper was applied on IBM SP-2 by hand using PASSION [7], a run-time library for parallel I/O. PASSION routines can be called from C and Fortran, and an out-of-core array can be associated with different layouts. All the reported times are in seconds. The experiments were performed for different values of *slab ratio* (SR), the ratio of available node memory to the size of

[2] Notice that, in some cases, the actual value of $p$ can change the preferred loop order.

[3] 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.

[4] Another option is to try the next most desirable loop permutation. Our choice is simpler and guarantees that the optimized program will be at least as good as the original one.

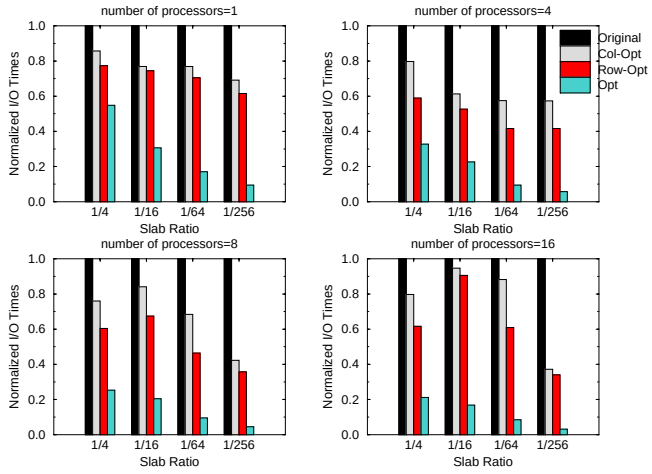


**Figure 4. Normalized I/O times for our example with $4K \times 4K$ (128 MByte) double arrays.**

out-of-core local arrays combined.

Figure 4 presents the normalized I/O times of four different versions of our first example (Figure 3) with $4K \times 4K$ (128 MByte) double arrays : unoptimized version (Original), optimized version using column-major layout for all arrays (Col-Opt), optimized version using row-major layout for all arrays (Row-Opt), and the version that is optimized by our approach (Opt).

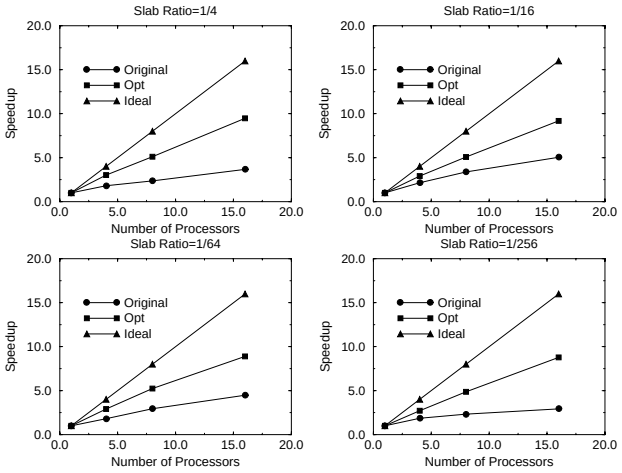


**Figure 5. Speedups for unoptimized and optimized versions of our example with $4K \times 4K$ double arrays.**

Figure 5 illustrates the speedups for Original and Opt versions. We define two kinds of speedups: speedup that is obtained for each version by increasing the number of processors, which we call $S_p$, and speedup that is obtained by using Opt version instead of the Original when the number of processors is fixed. We call this second speedup *local speedup* ($S_l$), and product $S_p \times S_l$ is termed as *combined speedup* (see Figure 6:(A)). We conclude the following:

(1) The Opt version performs much better than all other versions.

(2) When the slab ratio is decreased, the effectiveness of our approach increases (see Figure 4).

(3) As shown in Figure 5, the Opt version also scales better than the Original for all slab ratios.

(4) It is clear to see from Figure 6:(A) that combined speedup is much higher for small slab ratios. Note that the combined speedups are super-linear as the algorithm (loop order) is changed in the Opt version.

(5) When the slab ratio is very small, the optimized versions with fixed layouts for all files also perform much better than the Original.

### 5.1. Processors Coefficient and Memory Coefficient

The I/O optimizations introduced in this paper can be evaluated in two different ways:

(1) First, a problem that is solved by the Original version using a fixed slab ratio on $p$ processors can, in principle, be solved in the same or less time on $p'$ processors with the same slab ratio using the Opt version. The ratio $p/p'$ is termed as *processor coefficient* (PC).

(2) Second, a problem that is solved on a fixed number processors with a slab ratio $sr$ by the Original version can, in principle, be solved in the same or less time on the same number of processors with a smaller slab ratio (less memory) $sr'$ by the Opt version. We call the ratio $sr/sr'$ *memory coefficient* (MC).

The larger these coefficients are, the better as they indicate reduction in processor and memory requirements of the application program respectively.

Figures 6:(B) and (C) show the PC and MC curves respectively for our example with $4K \times 4K$ (128 MByte) double arrays. It can be observed that there is a slab ratio, called *critical slab ratio*, beyond which the shape of the PC curve does not change. In Figure 6:(B) the critical slab ratio is $1/64$. Below this ratio, independent of the node memory capacities, for a given $p$ it is possible to find the corresponding $p'$ where $p$ and $p'$ are as defined above. Similarly it can be observed that there is a number of processors beyond which the shape of the MC curve does not change. In Figure 6:(C) that number is $8$. This result means that beyond that number of processors, given an $sr$ it is possible to find the corresponding $sr'$ where $sr$ and $sr'$ are as defined above.

We believe that the final PC and MC curves give enough information about the performance of I/O optimizations.

Proceedings of the 11th International Parallel Processing Symposium (IPPS '97)
1063-7133/97 $10.00 © 1997 IEEE

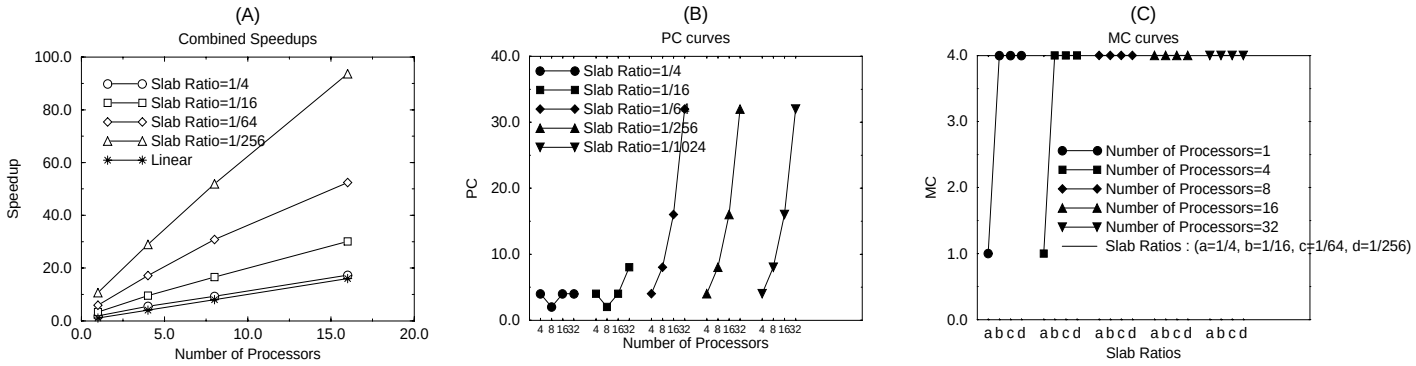


**Figure 6. (A) Combined Speedups for our example with $4K \times 4K$ double arrays. (B) PC curves. (C) MC curves.**

## 6. Related Work and Conclusions

Previous works on compiler optimizations to improve locality have concentrated on iteration space tiling. In [8] and [5], iteration space tiling is used for optimizing cache performance.

There is some work on compilation of out-of-core programs. In [2], the functionality of ViC*, a compiler-like preprocessor for out-of-core C* is described. In [6], the compiler support for handling out-of-core arrays on parallel architectures is discussed. In [1], a strategy to compile out-of-core programs on distributed-memory message-passing systems is offered. It should be noted that our optimization technique is general in the sense that it can be incorporated to any out-of-core compilation framework for parallel and sequential machines.

In this paper we presented how basic in-core compilation method can be extended to compile out-of-core programs. However, the code generated using such a straightforward extension may not give good performance. We proposed a three-step I/O optimization process by which the compiler can improve the code generated by the above method.

Our work is unique in the sense that it combines data transformations (layout determination) and control transformations (loop permutation) in a unified framework for optimizing out-of-core programs on distributed-memory message-passing machines.

## References

[1] 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*, July 1995.

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

[3] M. Kandemir, R. Bordawekar and A. Choudhary. I/O Optimizations for Compiling Out-of-Core Programs on Distributed-Memory Machines. To appear in *Eighth SIAM Conference for Parallel Processing for Scientific Computing*, March, 1997.

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

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

[6] M. Paleczny, K. Kennedy, and C. Koelbel. Compiler Support for Out-of-Core Arrays on Parallel Machines., CRPC Technical Report CRPC-TR94509-S, Rice University, Houston, TX, 1994.

[7] R. Thakur, A. Choudhary, R. Bordawekar, S. More and K. Sivaramkrishna. PASSIONate Approach to High-Performance Parallel I/O, In *IEEE Computer*, June 1996.

[8] 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.

[9] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. Vienna Fortran - a language specification. ICASE Interim Report 21, MS 132c, ICASE, NASA, Hampton VA 23681, 1992.