Data Access Reorganization in Compiling Out-of-Core Programs*

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

Abstract

This paper describes optimization techniques for translating out-of-core programs written in a data parallel language like HPF to message passing node programs with explicit parallel I/O. We first discuss how an out-of-core program can be translated by extending the method used for translating in-core programs. 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 improve the performance of 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 accesses, and (3) partitioning the available in-core node memory among references based on I/O cost estimation. Our analytical approach, simulation results and experimental results indicate that these optimizations can reduce the amount of time spent in I/O by as much as an order of magnitude. Performance results on IBM SP-2 and Intel Paragon are presented and discussed.

Keywords: Compilation techniques, out-of-core computations, parallel I/O, file layouts, loop optimizations

---

*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. The work of J. Ramanujam was supported in part by an NSF Young Investigator Award CCR-9457768, by an NSF grant CCR-9210422 and by the Louisiana Board of Regents through contract LEQSF(1991-94)-RD-A-09.

†EECS Dept., Syracuse University, Syracuse, NY 13244, e-mail: mtk@ece.nwu.edu
‡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, e-mail: jxr@ee.lsu.edu
¶CACR, Caltech, Pasadena, CA 91125, e-mail: rajesh@cacr.caltech.edu
1 Introduction

The use of parallel machines to solve large-scale computational problems in science and engineering has increased tremendously in recent years. This is primarily due to the considerable improvements in the potential performance of parallel computers in the last few years. Many of these applications—referred to as Grand Challenge Applications [15]—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 applications usually deal with large quantities of data. At present, a typical Grand Challenge Application could require between one gigabyte and four terabytes of data per run [10]. 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 parallel computers has not kept pace with the processing and communication capabilities of these machines. Thus the time to perform disk I/O has become a performance bottleneck. del Rosario and Choudhary [10] present an overview of the issues involved in high performance I/O.

Data parallel languages like HPF [18] have recently been developed to provide support for portable high performance programming on parallel machines. In order to use these languages for large-scale scientific computing, support for performing large-scale I/O from programs written in these languages is necessary. It is therefore essential to provide compiler support for these languages so that the programs can be translated automatically and efficiently.

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) estimating I/O costs associated with different access patterns in out-of-core computations, (2) reorganization of data storage in files to reduce I/O costs, (3) reordering of computations based on the reorganization of data, and (4) allocation of memory among multiple arrays accessed in a computation to minimize I/O cost. Experimental results demonstrate that these techniques can reduce I/O costs by as much as an order of magnitude compared to the costs of I/O when in-core compilation methods are naively extended for out-of-core computations.

The rest of the paper is organized as follows. Section 2 introduces our model and Section 3 explains out-of-core compilation strategy. How I/O optimizations can reduce the I/O cost of loop nests is discussed in Section 4. Experimental results are presented in Section 5. Section 6 presents a global I/O optimization algorithm. Section 7 discusses related work, and Section 8 concludes the paper.

2 Model for Out-of-Core Compilation

2.1 Programming Model

The most widely used programming model for large-scale scientific and engineering applications on distributed memory machines is the Single Program Multiple Data (SPMD) model. In this model, parallelism is achieved by partitioning data among processors. To achieve load balance, express locality of accesses and reduce communication, several distribution and data alignment strategies are often used, e.g., BLOCK, CYCLIC etc. Many parallel program-
Programming languages or language extensions provide directives that enable the expression of mappings from the problem domain to the processor domain and allow the user to align and distribute arrays in the most appropriate fashion for the underlying computation. 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 [30], Fortran D [14] and High Performance Fortran (HPF) [18]. While we discuss the compilation of out-of-core HPF programs in this paper, the discussion is applicable to programs written in other data parallel languages as well.

Explicit or implicit mapping of an array among processors results in each processor having an associated local array. For large data sets local arrays may not entirely fit in local memory. In such cases parts of the local arrays have to be stored on disk. We refer to such local arrays as out-of-core local arrays. Parts of the out-of-core local arrays need to be swapped between main memory and disk during the course of computation.

### 2.2 Data Storage Model

The data storage model shown in Figure 1 specifies how the out-of-core arrays are placed in files and how they are accessed by the processors. The out-of-core local arrays of each processor are stored in separate files called local array files. The local array files can be assumed to be owned by that processor. 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. If a processor needs data from the local array files of another processor (the owner of the data), the required data will be first transferred by the owner processor from its local disk to its memory and then sent to the requesting processor. Since data sharing is performed by explicit message-passing, this system is a natural extension of distributed-memory paradigm. The details of the data storage model can be found in the last author’s thesis [3].
We believe that this straightforward translation can be improved substantially by being more careful about choosing file layouts, loop ordering, and tile allocations. The proposed techniques, to be explained shortly, transform the loop nest shown in Figure 4(b) to the nest shown in Figure 4(c), and associates row-major file layout for arrays \( A \) and \( C \) and column-major file layout for array \( B \), and then allocates data-tiles of size \( S_d \times n \) for \( A \) and \( C \) and a data-tile of size \( n \times S_d \) for \( B \) as shown in Figure 5(d).\(^2\) The overall I/O cost of this new loop order and allocation scheme is:

\[
T_{overall}^d = \frac{n}{pS_d} \left( S_d C_{io} + S_d n t_{io} + n S_d t_{io} + \frac{n^2 C_{io} + n^2 n t_{io} + n^2 S_d + n^2 t_{io}}{pS_d} \right)
\]

\[
= \frac{n C_{io}}{p T_A} + \frac{n^2 t_{io}}{p T_D} + \frac{n C_{io}}{p S_d} + \frac{n^2 C_{io} + n^2 t_{io}}{p T_D}
\]

\[
= C_{io} \left( \frac{2n}{p} + \frac{n^2}{p S_d} \right) + t_{io} \left( \frac{2n^2}{p} + \frac{n^3}{p S_d} \right)
\]

provided that \( 3nS_d \leq M \). Note that under the assumption that when the maximum available memory is used, the cost for Figure 5(d) is much better than that of Figure 5(a), the original version. Note also 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 a good combination of file layout, loop order and memory allocation, given a naive translation. Our approach consists of three steps:

- Determination of the most appropriate file layouts for all arrays referenced in the loop nest (Section 4.3);
- Permutation of the loops in the nest in order to maximize spatial and temporal locality (Section 4.4);
- Partitioning the available memory among references based on I/O cost estimation (Section 4.5).

Throughout the paper 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. To be precise, we are assuming only one Uniformly Generated Reference Set (UGRS) [11] per array. For our purposes, all references in a UGRS can be treated as a single reference. Unless otherwise stated, the term loop in this paper refers to tiling loop. First we need a few definitions.

**Definition 1** 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 \( R, \text{layout} \) 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 \( ICost(IT, R, r, \text{layout}) \) where layout can be either row-major (rowmajor) or column-major (colmajor).\(^3\)

---

\(^2\)Assuming that the trip counts are equal to array sizes in relevant dimensions. Otherwise, either trip count, if it is known at compile time, or an estimation of it based on the array size should be used instead of \( n \). Notice also that since the compiler reads tiles of size \( S_d \times n \) and \( n \times S_d \), the tiling loops \( JT \) and \( KT \) disappear.

\(^3\)As will be explained later, the approach can be easily extended to handle all \( k! \) possible layouts for a \( k \)-dimensional array.
Table 1: BCost values for the program shown in Figure 4.(b).

<table>
<thead>
<tr>
<th>Index</th>
<th>Reference</th>
<th>Layout</th>
<th>BCost</th>
<th>Index</th>
<th>Reference</th>
<th>Layout</th>
<th>BCost</th>
</tr>
</thead>
<tbody>
<tr>
<td>IT</td>
<td>B[IT, JT]</td>
<td>colmajor S</td>
<td>IT</td>
<td>B[IT, JT]</td>
<td>colmajor n</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IT</td>
<td>C[IT, KT]</td>
<td>colmajor S</td>
<td>IT</td>
<td>C[IT, KT]</td>
<td>colmajor n</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KT</td>
<td>A(IT, JT)</td>
<td>colmajor S</td>
<td>KT</td>
<td>A[IT, JT]</td>
<td>colmajor n</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KT</td>
<td>B[KT, IT]</td>
<td>colmajor S</td>
<td>KT</td>
<td>B[KT, IT]</td>
<td>colmajor n</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KT</td>
<td>C[KT, KT]</td>
<td>colmajor S</td>
<td>KT</td>
<td>C[KT, KT]</td>
<td>colmajor n</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Definition 2** Assuming a file layout, 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. That is,

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

Notice that 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 local 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 4.4.

**Definition 3** Array Cost of an array reference R, assuming a layout, is the sum of BCost values for all loop indices with respect to reference R. In other words,

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

### 4.3 Determining File Layouts

Our heuristic for determining file layouts for out-of-core local arrays first computes the ACost values for all arrays under all possible layouts. It then chooses the combination that will allow the compiler to perform efficient file accesses. Consider the statement: \( A[IT, JT] = A[IT, JT] + B[KT, IT] + C[LT, KT] + 1 \) in Figure 4(b). The Basic I/O costs for this statement are given in Table 1. Using these BCost values, array costs (ACost values) for A, B and C are shown 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, KT and LT in that order) under the given layout of the file. Next, our heuristic considers all possible layout combinations by summing up the ACost values for the components of each combination term by term. Since, in our example, there are eight possible combinations, the term by term additions of ACost values are as shown in Table 3.

**Definition 4** 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.
Table 2: ACost values for the program shown in Figure 4:(b).

<table>
<thead>
<tr>
<th>Array</th>
<th>Layout</th>
<th>ACost</th>
<th>Array</th>
<th>Layout</th>
<th>ACost</th>
<th>Array</th>
<th>Layout</th>
<th>ACost</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>colmajor</td>
<td>S+p0+S+0</td>
<td>B</td>
<td>colmajor</td>
<td>S+p0+S+0</td>
<td>C</td>
<td>colmajor</td>
<td>0+S+0+nsS</td>
</tr>
<tr>
<td>A</td>
<td>rowmajor</td>
<td>S+p0+S+0</td>
<td>B</td>
<td>rowmajor</td>
<td>S+p0+S+0</td>
<td>C</td>
<td>rowmajor</td>
<td>S+p0+S+ns</td>
</tr>
</tbody>
</table>

Table 3: ACost values for the program shown in Figure 4:(b).

<table>
<thead>
<tr>
<th>Combination</th>
<th>Array A</th>
<th>Array B</th>
<th>Array C</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>colmajor</td>
<td>colmajor</td>
<td>colmajor</td>
<td>(S+p0+S)+(S+p0+S)+S</td>
</tr>
<tr>
<td>2</td>
<td>colmajor</td>
<td>colmajor</td>
<td>rowmajor</td>
<td>(S+p0+S)+(S+p0+S)+S+\n</td>
</tr>
<tr>
<td>3</td>
<td>colmajor</td>
<td>rowmajor</td>
<td>colmajor</td>
<td>S+p0+S+2p0+2S+\n</td>
</tr>
<tr>
<td>4</td>
<td>colmajor</td>
<td>rowmajor</td>
<td>rowmajor</td>
<td>S+1+S+2(S+p0)+\n</td>
</tr>
<tr>
<td>5</td>
<td>rowmajor</td>
<td>colmajor</td>
<td>colmajor</td>
<td>2p0+S+S(S+p0)+\n</td>
</tr>
<tr>
<td>6</td>
<td>rowmajor</td>
<td>colmajor</td>
<td>rowmajor</td>
<td>2p0+S+S(S+p0)+\n</td>
</tr>
<tr>
<td>7</td>
<td>rowmajor</td>
<td>rowmajor</td>
<td>colmajor</td>
<td>(nS+p0+S+\n S)+\n</td>
</tr>
<tr>
<td>8</td>
<td>rowmajor</td>
<td>rowmajor</td>
<td>rowmajor</td>
<td>(nS+S+S+\n nS)+\n</td>
</tr>
</tbody>
</table>

After creating a table of all possible layout combinations term by term, our layout determination heuristic chooses the combination with maximum number of constant-order and/or S-order terms. 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 (symbolic) cost. It is easy to see from Table 3 that, for our example, row-major layout for arrays A and C and column-major layout for array B (Combination 6) is a good (minimum I/O cost) combination, since it contains 2 S-order terms (S and 2S). Notice that there may be more than one such combination. Essentially, what our approach does is to attempt to read as much data as possible with a low I/O cost. For example, suppose that an S × S tile can be read by issuing S I/O calls. If we can also read an n × S tile by S calls this means that the additional (nS – S^2) elements need no extra I/O call. With the combination costs given in Table 3, our heuristic seeks the solution with the minimum number of I/O calls, and chooses the combination with the maximum number of constant-order and/or S-order terms. This choice will enable the compiler to read all the data along the relevant dimension(s) with a low I/O cost.

4.4 Deciding Loop Order

After selecting file layouts for each array, our compiler next determines an optimum (tiling) loop order that will enable efficient file accesses.

**Definition 5** 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 encloses. Generally speaking, $TCost(IT)$ is the estimated I/O cost caused by loop $IT$ when all array references are considered, i.e.,

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

where $R$ is the array reference and $layout_R$ is the layout of the associated file as determined in the previous step (Section 4.3).

Our algorithm for desired loop permutation is as follows:
• Calculate $T\text{Cost}(IT)$ for each tiling loop $IT$;

• If loop permutation is legal, then permute the tiling loops from outermost to innermost position according to non-increasing values of $T\text{Cost}$; if loop permutation is not legal, then the algorithm proceeds to the memory allocation step (Section 4.5); (and)

• Apply necessary loop interchange(s) to improve the temporal locality for (the tile of) the reference being updated. This step prevents the file-write operation from occurring at innermost tiling loops.\footnote{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.}

Returning to our example under the chosen file layouts (Combination 6), $T\text{Cost}(IT) = 2n/p, T\text{Cost}(JT) = S$, $T\text{Cost}(KT) = 2S$, and $T\text{Cost}(LT) = n$. So, the desired loop permutation from outermost to innermost position is $LT, IT, KT, JT$, assuming $p \geq 2$. But considering the temporal locality for the array being written to, the compiler interchanges $LT$ and $IT$, and reaches the order $IT, LT, KT, JT$.

### 4.5 Memory Allocation Scheme

Since each node has a limited amount of memory and in general a loop nest may contain a number of out-of-core arrays, the node memory should be partitioned among these out-of-core arrays suitably so that partitioning of the node memory should result in minimum I/O cost.

**Definition 6** We assume that the size of every array along each dimension is greater than one. Given an array and an associated layout, we define the *layout-conformant* position as the index of the fastest changing dimension of the array in the layout. For column- and row-major layouts, these positions are called the *column-conformant* and *row-conformant* positions respectively.

Our memory allocation scheme is as follows. The compiler divides the 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 row-major (column-major) layout group, the compiler considers all row-conformant (column-conformant) positions in turn. If a (filing) loop index appears in the conformant position of a reference and does not appear in any other position (except the conformant) of any reference in that group, then it sets the tile size for the conformant position to $n$; otherwise it sets the tile size to $S$. For all other index positions of that reference the tile size is set to $S$. After all the tile sizes for all dimensions of all array references are determined, our approach takes the size of the available memory ($M$) into consideration and computes the actual value for $S$.

As an example suppose that in a four-deep nest in which four two-dimensional arrays $A, B, C$ and $D$ are referenced, our layout determination algorithm has assigned row-major file layout for the arrays $A, B$ and $C$, and column-major file layout for the array $D$. Also assume that the references to those arrays are $A[IT, KT], B[JT, KT], C[IT, JT]$ and $D[KT, LT]$. Our memory allocation scheme divides those references into two groups: $A[IT, KT], B[JT, KT], C[IT, JT]$ in the row-major group, and $D[KT, LT]$ in the column-major group. Since $KT$ appears in
the row-conformant positions of \( A[IT, KT] \) and \( B[JT, KT] \), and does not appear any other position of any reference in this group, the tile sizes for \( A \) and \( B \) are determined as \( S \times n \). Notice that \( JT \) also appears in a column-conformant position (of the reference \( C[IT, JT] \)). But since it also appears in other positions of some other references (namely in the first position of \( B \)) in this group, the compiler determines the tile size for \( C[IT, JT] \) as \( S \times S \). Then it proceeds with the other group which contains the reference \( D[KT, LT] \) alone. Since \( KT \) is in the column-conformant position, and does not appear any other index position of \( D \), the compiler allocates a data tile of size \( n \times S \) for \( D[KT, LT] \). After those allocations the final memory constraint is determined as \( 3 \times n \times S + S \times S \leq M \). Given a value for \( M \), the value of \( S \) that utilizes all of the available memory can easily be determined by solving the second order equation \( S^2 + 3nS = M \) for positive \( S \), i.e., \( S = \left\lfloor \frac{-\sqrt{9n^2+4M} - 3n}{2} \right\rfloor \).

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 running example, the compiler 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 \( S \times n \) 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, our algorithm allocates a data-tile of size \( n \times S \) for \( B[KT, IT] \) as shown in Figure 5:(d). Notice that after these tile allocations tiling loops \( KT \) and \( JT \) disappear and the node program shown in Figure 4:(c) is obtained.

In what follows we show that neither a fixed column-major file layouts nor fixed row-major file layouts for all arrays results in optimal I/O cost.

- **Figure 5(b):** Assume a fixed column-major file layout for all arrays. In that case, \( T\text{Cost}(IT) = S_b + n/p \), \( T\text{Cost}(JT) = n, T\text{Cost}(KT) = S_b + n \), and \( T\text{Cost}(LT) = S_b \) (using the first row of Table 3). So, from outermost to innermost position \( KT, JT, IT, LT \) is the desirable loop permutation. Again 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. 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 \( n \times S_b \) for \( C \), and tiles of size \( S_b^2 \) for each of \( A \) and \( B \) as shown in Figure 5(b). The reason for assigning tiles of size \( S_b^2 \) is that although \( IT \) and \( KT \) appear in column-conformant positions, they appear in other positions as well. The overall I/O cost of this approach is

\[
T_{\text{overall}}^b = \frac{n^2}{pS_b} \left( S_bC_{i0} + S_b^2 t_{i0} + \frac{n(S_bC_{i0} + S_b^2 t_{i0})}{S_b} + \frac{n(S_bC_{i0} + nS_b t_{i0})}{S_b} \right)
= \frac{n^2}{pS_b} C_{i0} + \frac{n^2 t_{i0}}{p} + \frac{n^3 C_{i0}}{pS_b^2} + \frac{n^3 t_{i0}}{pS_b^2} + \frac{n^4 t_{i0}}{pS_b^2}
\]

---

Note that for this and the following case (fixed row-major layouts), during memory allocation, there is only one group, and all array references belong to it.
under the memory constraint \(2S_c^2 + nS_b \leq M\).

- **Figure 5(c):** Assume a fixed row-major layout for all arrays. In that case, \(TCost(IT) = n/p + S_c, TCost(JT) = S_c, TCost(KT) = n + S_c\), and \(TCost(LT) = n\) (from the last row of Table 3). From outermost to innermost position \(KT, LT, IT, JT\) is the desirable loop permutation. Once more considering the temporal locality, our compiler takes \(JT\) to the outermost position. So, the final loop order is \(IT, KT, LT, JT\). The compiler allocates a tile of size \(S_c n\) for \(A\), and tiles of size \(S_c^2\) for each of \(B\) and \(C\) as shown in Figure 5(c). The overall I/O cost of this approach is

\[
T_{\text{overall}}^c = \frac{n}{pS_c} \left( \frac{S_c C_{io} + S_c n t_{io}}{S_c} + \frac{n(S_c C_{io} + S_c^2 t_{io})}{S_c} + \frac{n^2(S_c C_{io} + S_c^2 t_{io})}{S_c^2} \right)
\]

\[
= \frac{nC_{io}}{p} + \frac{n^2 t_{io}}{p} + \frac{n^2 C_{io}}{pS_c} + \frac{n^2 t_{io}}{pS_c^2} + \frac{n^3 C_{io}}{pS_c^2} + \frac{n^3 t_{io}}{pS_c^3}
\]

under the memory constraint \(2S_c^2 + nS_c \leq M\).

It should be noted that although for reasonable values of \(M\) these costs are better than that of the original version, they are much worse than the one obtained by our approach. These analytical results were also confirmed by our experiments as will be shown in Section 5.

### 4.6 The Overall Algorithm

Figure 6 shows the overall algorithm for optimizing file layouts and access pattern for a single loop nest. The steps marked with \(\ast\), \(\ast\) and \(\odot\) belong to **layout determination**, **loop permutation** and **memory allocation** parts of the algorithm respectively.

### 4.7 Matrix multiplication example

As a second example we consider a nest that is taken from matrix multiplication program (see Figure 7(a)). Suppose that \(j^{th}\) column of a matrix \(X\) is denoted by \(x_j\) and let \(A, B\) and \(C\) be \(n \times n\) matrices. The algorithm for computing \(C = A \times B\) is \(c_j = \sum_{k=1}^{n} b_{kj}a_{yk}\), for \(j = 1, \ldots, n\).

As before the compilation is performed in two phases. The resulting straightforward node program is shown Figure 7(b). The local out-of-core arrays with square data-tiles are shown in Figure 8(a).

Next we demonstrate how our three-step approach optimizes the I/O cost of this program. We concentrate on only \(A\) and \(B\), as they are used more frequently than array \(C\). The Basic I/O costs for this nest are given in Table 4 and the array costs for \(A\) and \(B\) 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. It is clear that the Combination 2 which associates column-major layout with \(A\) and row-major layout with \(B\) is the most appropriate one, since it contains two \(S\)-terms.
input: a tiled loop nest with a number of out-of-core array references

output: for each array a file layout, for the nest a loop order, and tile sizes for each dimension of each reference

notation:

\( IT \): a (tiling) loop index

\( R \): an array reference (may represent a UGRS)

\( \text{layout} \): file layout for an out-of-core array (in this paper \( \text{layout} \in \{ \text{row-major, column-major} \} \))

\( \text{layout}_R \): assigned file layout for the reference \( R \)

\( \text{group} \): a group of references with the same file layout

\( n \): array size (and upper bound for the loops)

\( S \): a parameter that satisfies \( S \ll n \)

- For all \( IT, R, r \) and layout compute \( I\text{Cost}(IT, R, r, \text{layout}) \)
- For all \( IT, R \) and layout compute \( B\text{Cost}(IT, R, \text{layout}) = \sum_r I\text{Cost}(IT, R, r, \text{layout}) \)
- For all \( R \) and layout compute \( A\text{Cost}(R, \text{layout}) = \sum_{IT} B\text{Cost}(IT, R, \text{layout}) \)
- Consider all possible layout combinations and choose the one with maximum number of constant-order and/or \( S \)-order terms
- According to the chosen combination in the previous step, associate each array reference \( R \) with a \( \text{layout}_R \)
- For all \( IT \) compute \( T\text{Cost}(IT) = \sum_{R, \text{layout}_R} B\text{Cost}(IT, R, \text{layout}_R) \)
- Permute the tiling loops from outermost to innermost position according to non-increasing values of \( T\text{Cost} \)
- Apply necessary loop interchange(s) to improve temporal locality
- Divide the array references into groups according to the file layouts of the associated files
- For all groups
  - if \( R \in \text{group} \)
    - if a loop index appears in the conformant position of this reference and does not appear in any other position (except conformant) of any reference in this group, then set the tile size for the conformant position to \( n \); otherwise set the tile size to \( S \)
    - if there is any conflict with the previous group(s), set the tile size of the conformant position to \( S \)
    - for all other index positions set the tile size to \( S \)
  - Determine the value of \( S \) by considering the value of \( M \)

Figure 6: Overall algorithm for optimizing locality.

Table 4: BCost values for the program shown Figure 7:(b).

<table>
<thead>
<tr>
<th>Index</th>
<th>Reference</th>
<th>Layout</th>
<th>BCost</th>
<th>Index</th>
<th>Reference</th>
<th>Layout</th>
<th>BCost</th>
</tr>
</thead>
<tbody>
<tr>
<td>IT</td>
<td>A(K,L,T)</td>
<td>colmajor</td>
<td>np</td>
<td>IT</td>
<td>A(K,L,T)</td>
<td>colmajor</td>
<td>0</td>
</tr>
<tr>
<td>IT</td>
<td>A(K,L,T)</td>
<td>rowmajor</td>
<td>S</td>
<td>IT</td>
<td>A(K,L,T)</td>
<td>rowmajor</td>
<td>0</td>
</tr>
<tr>
<td>IT</td>
<td>B(T,L,T)</td>
<td>colmajor</td>
<td>S</td>
<td>IT</td>
<td>B(T,L,T)</td>
<td>colmajor</td>
<td>a</td>
</tr>
<tr>
<td>IT</td>
<td>B(T,L,T)</td>
<td>rowmajor</td>
<td>np</td>
<td>IT</td>
<td>B(T,L,T)</td>
<td>rowmajor</td>
<td>S</td>
</tr>
</tbody>
</table>

Table 5: ACost values for the program shown Figure 7:(b).

<table>
<thead>
<tr>
<th>Army</th>
<th>Layout</th>
<th>ACost</th>
<th>Army</th>
<th>Layout</th>
<th>ACost</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>colmajor</td>
<td>np+S</td>
<td>B</td>
<td>colmajor</td>
<td>S+np</td>
</tr>
<tr>
<td>A</td>
<td>rowmajor</td>
<td>S+np</td>
<td>B</td>
<td>rowmajor</td>
<td>np+S</td>
</tr>
</tbody>
</table>
PARAMETER (n=..., p=...)  
REAL A(n,n),B(n,n),C(n,n),TEMP(n,n)  
IHPFS PROCESSORS P(p)  
IHPFS TEMPLATE D(a)  
IHPFS DISTRIBUTE D(BLOCK) ON P  
IHPFS ALIGN (*,:) WITH D::A,C,TEMP  
IHPFS ALIGN ([,]) WITH D::B  
DO J = 1,n  
P/FALL (k=1:n)  
TEMP(1:n,k)=B(k,j)*A(1:n,k)  
ENDDO  
C(1:n,)=SUM(TEMP,2) !Sum Intrinsic  
END

DO JT = L_JT,U_JT,S_JT  
DO IT = L_IT,U_IT,S_IT  
read data-tile for B  
DO KT = L_KT,U_KT,S_KT  
read data-tile for A  
TEMP[KT,IT]=A[KT,IT]*B[IT,]  
ENDDO KT  
ENDDO IT  
participate in global sum  
if (owner) store the jth column  
ENDDO JT

Figure 7: (a) Out-of-core matrix multiplication nest. (b) Resulting straightforward node program.

Table 6: Possible file layout combinations for the program shown Figure 7:(b).

<table>
<thead>
<tr>
<th>Combination</th>
<th>Array A</th>
<th>Array B</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>column</td>
<td>column</td>
<td>(n+p)!+n!</td>
</tr>
<tr>
<td>2</td>
<td>column</td>
<td>row</td>
<td>2n+p+8+n!</td>
</tr>
<tr>
<td>3</td>
<td>row</td>
<td>column</td>
<td>2n+n!</td>
</tr>
<tr>
<td>4</td>
<td>row</td>
<td>row</td>
<td>(2n+p)!+n!</td>
</tr>
</tbody>
</table>

Under these layouts, $TCost(IT) = 2n/p$, $TCost(JT) = S$, and $TCost(KT) = S$. From outermost to innermost position $IT$, $JT$, $KT$ and $IT$, $KT$, $JT$ are two desirable loop permutations. Going on with the former, the compiler allocates a tile of size $nS$ for array $A$ and a tile of size $Sn$ for array $B$ as illustrated in Figure 8:(b).

4.8 Discussion

The complexity of our heuristics is $\Theta(kmd + 2^m + k \log k)$ if layout determination algorithm is used, otherwise it is $\Theta(kmd + k \log k)$ where $k$ 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 the $TCost$ values. Since in practice $k$, $m$ and $d$ are very small (e.g. 2, 3 etc.), all steps are inexpensive and the approach is efficient.

Notice that our approach is applicable to loop nests that have arbitrary reuse spaces as well [28]. Consider the statement $A[IT, JT] = A[IT, JT] + B[JT + LT, KT]$. Assuming a uniprocessor and column-major layout for all arrays, $TCost(IT) = S$, $TCost(JT) = n + S$, $TCost(KT) = n$ and $TCost(LT) = S$.

It should also be noted that our three-step approach can be used in different ways. If the way data arrives (eg. 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 technique, or
- redistributes data based on the result of the layout determination algorithm, and then applies the loop permutation and memory allocation heuristics. Of course, this redistribution involves some additional overhead which can be amortized if the array is used several times.
If on the other hand the compiler has to create the out-of-core arrays from scratch, then it can apply all three steps of the proposed technique.

As an example suppose that for the first example of this paper (Figure 4), the file layouts for the arrays $A$ and $B$ are fixed to row-major and column-major respectively. In that case our algorithm needs to consider only columns 5 and 6 of the Table 3 to find the optimal layout combination. We call this version of the algorithm Constrained(); in contrast to Unconstrained() which can choose any combination without any layout constraints. The next section (Global I/O Optimization) shows that the Constrained() is very important for global I/O optimization.

As we have indicated before, if the desired loop permutation is not legal (i.e., dependence-preserving), then the compiler keeps the original loop order and applies only the memory allocation algorithm\(^6\).

To summarize, an important characteristic of our scheme is that all three steps are independent from each other and depending on the specific requirements of the out-of-core application in hand, some combination of them can be applied. Finally, it should be emphasized that our I/O optimization heuristic reduces the code size in general by eliminating as many tiling loops as possible. For instance in Figure 4 two tiling loops ($JT$ and $KT$) are eliminated, due to the optimized file accesses.

### 4.9 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, it considers two possible file layouts only: row-major and column-major. In fact, an $m$-dimensional array can be stored in file in one of the $m!$ forms, each of which corresponding to layout of data linearly by a nested traversal of the axes in some order. Our algorithm can easily be modified to incorporate all possible layout combinations. The necessary changes should be made to (1) file layout determination part. This will increase the cost of this part substantially, as that step performs exhaustive search over all possible file layout combinations, and (2) memory allocation scheme. Now, the references should be divided into $m!$ groups, and each group should be handled separately.

---

\(^6\)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.
Table 7: I/O times for the first example (Figure 4) with $2K \times 2K$ (32 MByte) double arrays on IBM SP-2.

<table>
<thead>
<tr>
<th>Number of Processors = 1</th>
<th>Number of Processors = 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/4</td>
<td>152</td>
</tr>
<tr>
<td>1/16</td>
<td>623</td>
</tr>
<tr>
<td>1/64</td>
<td>3254</td>
</tr>
<tr>
<td>1/256</td>
<td>25087</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of Processors = 8</th>
<th>Number of Processors = 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/4</td>
<td>71</td>
</tr>
<tr>
<td>1/16</td>
<td>103</td>
</tr>
<tr>
<td>1/64</td>
<td>1917</td>
</tr>
<tr>
<td>1/256</td>
<td>8841</td>
</tr>
</tbody>
</table>

5 Experimental Results

The optimizations introduced in this paper were performed by hand using the PASSION runtime library for parallel I/O [27]. PASSION routines can be called from C and Fortran, and an out-of-core array can explicitly be associated with a data layout (e.g. row-major or column-major). In the experiments C language was used.

All experiments were run on an IBM SP-2 and Intel Paragon. 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 8MB/s. The nodes on Paragon, on the other hand, are divided into three groups: compute nodes, HIPPI nodes and service nodes. The platform where the experiments were conducted has 3 service nodes each with a RAID SCSI disk array attached to it. The total disk capacity of the system is 14.4 GBytes. All the reported times are in seconds. The experiments were performed for different values of Slab Ratio (SR), the ratio of available in-core node memory to the size of out-of-core local arrays combined.

Tables 7 and 8 present the I/O times of four different versions of our first example (Figure 4) with $2K \times 2K$ (32 MByte) and $4K \times 4K$ (128 MByte) double arrays on IBM SP-2: Original 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). Figures 9 and 10 show the I/O times normalized with respect to the Original version within each slab ratio on SP-2. Figures 11 and 12 illustrate 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 ($S_l$); the product $S_pS_l$ is called the combined speedup. The combined speedup is a parameter which shows the combined effect of the I/O optimizations and the increasing number of processors. In our work, its primary use is in determining the slab ratio(s) for which the I/O optimizations are most effective. Figure 13 illustrates the combined speedup curves of our example for different slab ratios for $2K \times 2K$ and $4K \times 4K$ double arrays.

Tables 9 and 10 and Figures 14 and 15 show the improvement in I/O time for the same example on Intel Paragon. Due to lack of space we do not present the speedup and combined speedup curves for Intel Paragon, but they are quite
Figure 9: Normalized I/O times for the first example (Figure 4) with $2K \times 2K$ (32 MByte) double arrays on IBM SP-2.
Figure 10: Normalized I/O times for the first example (Figure 4) with $4K \times 4K$ (128 MByte) double arrays on IBM SP-2.
Figure 11: Speedups for unoptimized and optimized versions of the first example (Figure 4) with $2K \times 2K$ double arrays on IBM SP-2.
Figure 12: Speedups for unoptimized and optimized versions of the first example (Figure 4) with $4K \times 4K$ double arrays on IBM SP-2.
Table 8: I/O times for the first example (Figure 4) with \(4K \times 4K\) (128 MByte) double arrays on IBM SP-2.

<table>
<thead>
<tr>
<th>Number of Processors = 1</th>
<th>Number of Processors = 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/4</td>
<td>383</td>
</tr>
<tr>
<td>1/16</td>
<td>1441</td>
</tr>
<tr>
<td>1/64</td>
<td>8384</td>
</tr>
<tr>
<td>1/256</td>
<td>48880</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of Processors = 8</th>
<th>Number of Processors = 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/4</td>
<td>54</td>
</tr>
<tr>
<td>1/16</td>
<td>277</td>
</tr>
<tr>
<td>1/64</td>
<td>2858</td>
</tr>
<tr>
<td>1/256</td>
<td>2026</td>
</tr>
</tbody>
</table>

Figure 13: Combined speedups for the first example (Figure 4) with \(2K \times 2K\) (left) and \(4K \times 4K\) (right) double arrays on IBM SP-2.

Figure 16 and 17 demonstrate the performance of our optimizations for the second running example of this paper (matrix multiplication nest). The experiments were performed with \(4K \times 4K\) double arrays on SP-2. As shown in Figure 16, the improvement in the I/O time is highly significant, especially with smaller slab ratios. As noted before, the effectiveness of our optimizations gets stronger as number of processors is increased. Figure 17 clearly shows that the optimized version scales better than the original. The combined speedups in Figure 18 indicate the importance of I/O optimizations with smaller slab ratios.

5.1 Additional examples

We have also applied our optimizations to a number of common kernels. The experiments were conducted on four nodes of SP-2 with \(4K \times 4K\) two-dimensional and \(4K\) one-dimensional double arrays and the results are shown in Figure 19 as normalized I/O times.
Figure 14: Normalized I/O times for the first example (Figure 4) with $2^k \times 2^k$ (32 MByte) double arrays on Intel Paragon.
Figure 15: Normalized I/O times for the first example (Figure 4) with $4K \times 4K$ (128 MByte) double arrays on Intel Paragon.
Figure 16: Normalized I/O times for matrix multiplication nest (Figure 7) with $4K \times 4K$ (128 MByte) double arrays on IBM SP-2.
Figure 17: Speedups for unoptimized and optimized versions of matrix-multiply nest with $4K \times 4K$ double arrays on IBM SP-2.

Figure 18: Combined speedups for matrix-multiply nest with $4K \times 4K$ double arrays on IBM SP-2.
Table 9: I/O times for the first example (Figure 4) with $2K \times 2K$ (32 MByte) double arrays on Intel Paragon.

<table>
<thead>
<tr>
<th>Number of Processors = 1</th>
<th>Number of Processors = 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>1/4</td>
<td>1359</td>
</tr>
<tr>
<td>1/16</td>
<td>2320</td>
</tr>
<tr>
<td>1/64</td>
<td>19201</td>
</tr>
<tr>
<td>1/256</td>
<td>9687</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of Processors = 8</th>
<th>Number of Processors = 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>1/4</td>
<td>682</td>
</tr>
<tr>
<td>1/16</td>
<td>634</td>
</tr>
<tr>
<td>1/64</td>
<td>621</td>
</tr>
<tr>
<td>1/256</td>
<td>48966</td>
</tr>
</tbody>
</table>

- Figure 19:(a) gives the performance improvement obtained on a matrix-transpose loop that contains the statement $B(i, j) = A(j, i)$. Our layout heuristic assigns column-major layout for array $A$ and row-major layout for array $B$. It then allocates a tile of size $nS$ to $A$, and a tile of size $Sn$ to $B$. Notice that no fixed file layout for all arrays can obtain that performance.

- The performance improvement on iterative-solver nest is given in Figure 19:(b). The innermost loop contains the statement $Y(k) = Y(k) - U(k, j) \times X(j)$. For this imperfectly nested example our heuristic associates column-major layout for all arrays.

- Figure 19:(c) shows the results of our optimizations on matrix-smoothing (four-point relaxation) nest. The optimized statement is $B(i, j) = 0.25 \times (A(i - 1, j) + A(i + 1, j) + A(i, j - 1) + A(i, j + 1))$. Our compiler associates row-major layout for all arrays (a fixed column-major layout for all arrays is also equally acceptable).

- Figure 19:(d) illustrate the performance improvement on matrix-vector multiplication ($Y=AX$). It is interesting to note that in this example all layout combinations are equally good as far as I/O cost is concerned. Going with column-major layout for all arrays, our compiler allocates a data-tile of size $nS$ for $A$, a tile of size $n$ for $Y$, and a tile of size $S$ for $X$.

- Finally, performance of the well-known $ijk$ matrix-matrix multiplication on SP-2 and Paragon is given in Figures 19:(e) and (f) respectively. The nest contains the statement $C(i, j) = C(i, j) + A(i, k) \times B(k, j)$. Six of the eight possible combinations are equally good as far as I/O cost is concerned, though none of them allows efficient array accesses for all arrays. Column-major layouts for $A$ and $B$, and row-major layout for $C$ is one of the worst combinations. Another worst combination associates row-major layouts for $A$ and $B$, and column-major layout for $C$. Among the remaining combinations, the following can be chosen: column-major layouts for $A$ and $C$, and row-major layout for $B$. This choice eliminates the tiling loop $IT$ (the tiling loop associated with $i$), but the file accesses for the array $B$ remain unoptimized. Because of this, the speedup is bounded by 1.65 on SP-2 and by 1.73 on Paragon.
Figure 19: Normalized I/O times for different nests. (a) Matrix transpose (on SP-2). (b) Iterative solver (on SP-2). (c) Matrix smoothing (on SP-2). (d) Matrix-vector multiplication (on SP-2). (e) Matrix-matrix multiplication (on SP-2). (f) Matrix-matrix multiplication (on Paragon).
Table 10: I/O times for the first example (Figure 4) with $4K \times 4K$ (128 MByte) double arrays on Intel Paragon.

<table>
<thead>
<tr>
<th>Number of Processors = 1</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SR</td>
<td>Original</td>
<td>Col-Opt</td>
<td>Row-Opt</td>
<td>Opt</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1/4</td>
<td>5172</td>
<td>4431</td>
<td>4001</td>
<td>1273</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1/16</td>
<td>7986</td>
<td>5683</td>
<td>5412</td>
<td>1344</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1/64</td>
<td>28864</td>
<td>23002</td>
<td>20357</td>
<td>2894</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1/256</td>
<td>92512</td>
<td>141370</td>
<td>117270</td>
<td>8193</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of Processors = 4</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SR</td>
<td>Original</td>
<td>Col-Opt</td>
<td>Row-Opt</td>
<td>Opt</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1/4</td>
<td>1840</td>
<td>1546</td>
<td>1220</td>
<td>410</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1/16</td>
<td>3989</td>
<td>2695</td>
<td>1094</td>
<td>556</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1/64</td>
<td>79251</td>
<td>12128</td>
<td>9577</td>
<td>1233</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1/256</td>
<td>123545</td>
<td>88246</td>
<td>80940</td>
<td>5043</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of Processors = 8</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SR</td>
<td>Original</td>
<td>Col-Opt</td>
<td>Row-Opt</td>
<td>Opt</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1/4</td>
<td>1186</td>
<td>956</td>
<td>786</td>
<td>221</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1/16</td>
<td>1862</td>
<td>1341</td>
<td>965</td>
<td>286</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1/64</td>
<td>11120</td>
<td>9976</td>
<td>8100</td>
<td>701</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1/256</td>
<td>79031</td>
<td>56780</td>
<td>51100</td>
<td>2692</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of Processors = 16</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SR</td>
<td>Original</td>
<td>Col-Opt</td>
<td>Row-Opt</td>
<td>Opt</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1/4</td>
<td>810</td>
<td>670</td>
<td>600</td>
<td>135</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1/16</td>
<td>1271</td>
<td>957</td>
<td>851</td>
<td>188</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1/64</td>
<td>7590</td>
<td>5698</td>
<td>5012</td>
<td>474</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1/256</td>
<td>32955</td>
<td>35330</td>
<td>30994</td>
<td>1701</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5.2 Observations

From these results we observe the following:

- The Opt version performs much better than all other versions. The reason for this result is that in the Opt version, all array accesses are optimized (as shown in Figure 5):(d)); in contrast to the other versions where accesses to at most one array could be optimized, and the accesses for the other arrays have remained as bottleneck (see Figure 5):(a), (b) and (c)).

- When the number of processors is increased, the effectiveness of our approach (Opt) increases (see Figures 9, 10, 14 and 15). This is because of the fact that more processors are now working on the out-of-core local array(s) I/O optimally.

- When the slab ratio is decreased, the effectiveness of our approach increases (see Figures 9, 10, 14 and 15). As the amount of node memory is reduced, the Original version performs many small I/O requests which degrades the performance dramatically. The Opt version on the other hand continues with optimized I/O no matter how small the node memory is.

- As shown in Figures 11 and 12, the Opt version also scales better than the Original for all slab ratios. This is due to poor I/O performance of all involved processors in the Original version. The speedups are limited because of the fact that the array C is not distributed.

- It is clear from Figure 13 that combined speedup is much higher for small slab ratios. That again points to the importance of our I/O optimizations in environments where the amount of in-core node memory is small compared to the size(s) of the out-of-core local array(s). Note that all of the combined speedups are superlinear as the loop order is changed in the Opt version.

- It should also be noted that when the slab ratio is very small, the optimized versions with fixed layouts also perform much better than the Original.
Figure 20: Number of I/O calls for the first example (Figure 4) with different problem sizes on a single node.

- Demonstrations on two different platforms with varying compile-time and run-time parameters, such as number of processors, available memory, array sizes etc., prove that our algorithm is quite robust.

We believe that the efficacy of our optimizations will increase with much larger data sets than our problem sizes since there will be more I/O to optimize. Note that a case with a unit slab ratio is interesting to consider as it is different from the case when the entire data set is stored in memory. Our experiments confirm that the I/O optimizations perform well even when the slab ratio is 1 or extremely small (e.g. 1/1024, 1/4096 etc.). However, due to lack of space, we do not present those results. In order to present a uniform compilation phase, the (in-core) case when all the data sets can fit in memory should be handled as well. For this case we offer an adaptive approach which checks the available memory and problem size at runtime and suppresses tiling loops and I/O requests if the problem can be fully in-core.

5.3 Results from analytic modeling

In order to further evaluate how effective our approach is at reducing the number of I/O calls and to explain the reduction observed in I/O times when our approach is applied, we have calculated the number of I/O calls (coefficient of $C_{io}$) for different problem sizes for the nest shown in Figure 4 on a single node. The resulting curves are presented on logarithmic scale in Figure 20. 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 in these curves. For example in Figure 20(c) with a memory size of \( 10^8 \) elements, the number of I/O calls required for Original, Col-Opt, Row-Opt and Opt versions are approximately \( 7 \times 10^6 \), \( 1.5 \times 10^6 \), \( 8 \times 10^5 \) and \( 10^5 \) respectively. These curves clearly show that our approach is successful in reducing the number of I/O calls.

5.4 Processor Coefficient and Memory Coefficient

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

- 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 (i.e. a memory of the same size) using the Opt version. The ratio \( p/p' \) is called the processor coefficient (PC).

- Second, a problem that is solved on a fixed number processors with a slab ratio \( s \) 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 (i.e. less memory) \( s' \) by the Opt version. The ratio \( s/s' \) is called the memory coefficient (MC).

It is better to have these coefficients as large as possible, since larger values indicate reduction in processor and memory requirements of the application program respectively.

Figures 21 and 22 shows the PC and MC curves for the first example with \( 2K \times 2K \) (32 MByte) and \( 4K \times 4K \) (128 MByte) double arrays respectively on SP-2. 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 21 and 22 the critical slab ratio is 1/64. The practical significance of this observation is that below the critical slab 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. For instance, suppose that in this example nest (Figure 4), a problem can be solved using a number of processors and the Original version in \( t_o \) seconds with a slab ratio which is equal to or smaller than the critical slab ratio. It was observed that the same problem can be solved within \( t_o \) seconds in a single processor using the Opt with the same slab ratio. The reason for this is that the I/O in the Original is performed unoptimally, and that after the critical slab ratio is reached, increasing the number of processors does not give an additional benefit at all. 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 22 that number is 8 while in Figure 21 it is 1. This result means that beyond that number of processors, given an \( s \) it is possible to find the corresponding \( s' \) where \( s \) and \( s' \) are as defined above. Since the Opt version can solve the same problem on the same number of processors using less memory than the Original, we argue that our optimizations are especially useful in environments where the individual nodes are multiprogrammed. And finally PC and MC values for this loop nest are given in Figure 23. It is easy to see that the critical slab ratio is 1/256.

We believe that the final PC and MC curves give enough information about the performance of our I/O optimizations in a multicomputer environment where large arrays are processed.
Figure 21: PC and MC curves for the first example (Figure 4) with $2K \times 2K$ double arrays on IBM SP-2.

Figure 22: PC and MC curves for the first example (Figure 4) with $4K \times 4K$ double arrays on IBM SP-2.

6 Global I/O Optimization

So far we applied our technique to one possibly imperfect nest. In this section we show how the algorithm can be extended to work on multiple (a sequence of) nests. Since several arrays may be accessed by each nest and since each of these nests may require a different (optimal) file layout for a specific out-of-core array, the algorithm should find a layout for that array that satisfies the majority of the nests. In the following two subsections we

- show that the global layout optimization problem is NP-complete, even in its very restricted form, and
- propose a heuristic that works well in practice.
6.1 General Problem

Let \( \{N_1, N_2, ..., N_m\} \) denote the different loop nests in the program and \( \{A_1, A_2, ..., A_r\} \) denote the different out-of-core arrays. In general each nest can access a subset of these arrays. We assume that our algorithm described before is run for each nest, and a number of possible optimized layout combinations are obtained for each nest. In the following, we show that the problem of finding a global array layout combination that satisfies all the nests is NP-complete even for the restricted case where only row-major and column-major arrays are considered.

When our algorithm described before is run for each nest in the program, we obtain possible layouts similar to shown in Table 11. We define the number of entries in the table as the size of the problem.

First, the problem belongs to the class NP. This is because a nondeterministic algorithm can guess a solution and check in polynomial time whether or not that solution (certificate) satisfies all the nests [7]. Next, we reduce the satisfiability problem [7, 12] to our problem as follows. First a given formulation is transformed to multiplications of sums (a polynomial-time operation). After that, each multiplicative term is associated with a nest, and each sub-term (clause) in a multiplicative term is associated with a layout combination. With each logical variable \( x \) we associate an out-of-core array \( X \). If the logical variable appears itself, we assign column-major file layout for \( X \); if \( \bar{x} \) (complement of \( x \)) appears, we assign row-major file layout for \( X \).

As an example the layout assignments shown in Table 11 correspond to the following formulation where \( a_i \) and \( \bar{a}_i \) are the logical variables associated with \( A_i \).

\[
(ad_1d_2d_3 + \bar{a}d_1a_2a_3). (a_2a_5 + \bar{a}d_2a_5). (a_3a_4)
\]

There might be however some special cases to handle. For example, after obtaining the desired form, a multiplicative term can contain terms such as in \((ac + b\bar{c})\). This expression should be transformed to \((a(b + \bar{b})c + (a + \bar{a})b\bar{c}) = abc + ab\bar{c} + ab\bar{c} + ab\bar{c} \) so that each sub-term contains logical variables \( a, b \) and \( c \) or complements of them.

It is easy to see that the formulation is satisfied if and only if there is a layout assignment that satisfies all the nests.
Table 11: Local layout assignments for three nests.

<table>
<thead>
<tr>
<th>N1</th>
<th>A1</th>
<th>A2</th>
<th>A3</th>
<th>A4</th>
<th>A5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>row-major</td>
<td>row-major</td>
<td>row-major</td>
<td>column-major</td>
<td>column-major</td>
</tr>
<tr>
<td>N2</td>
<td></td>
<td>column-major</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>row-major</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N3</td>
<td>column-major</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Since the reduction can be achieved in polynomial time, the problem is NP-hard; and since it belongs to the class NP as well, it is NP-complete.

Note that since this problem is a restricted version of the most general problem of finding suitable layout assignments such that the value of a cost function does not exceed \( r \); we argue that the general problem is also NP-complete.

6.2 A Simple Heuristic

It should be emphasized that in the preceding subsection we have assumed that all the nests are separately optimized, i.e., the optimal array layouts and loop orders are determined for each nest. In the following we outline a simple heuristic which does not assume this. Our approach is based on the concept of most expensive nest. Intuitively, this is the nest which takes the most I/O time and should be optimized. Different methods can be adopted to choose this nest. For example the programmer can use compiler directives to give hints about this nest. We can also use a metric such as multiplication of the number of loops and the number of arrays referenced in the nest. The nest which has the largest resulting value can be marked as the most expensive nest. Then the algorithm proceeds as follows: First, the most expensive nest is fully optimized by using the algorithm presented in this paper (Unconstrained()). After this step, file layouts for some of the arrays will be determined. Then each of the remaining nests can be optimized using the approach presented for the constrained layout case in Section 4.8 (Constrained()). After each nest is optimized, new layout constraints will be obtained, and these will be propagated for optimization of the next nest. Note that the order of processing for the remaining nests may 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 expensive nest.

Figure 24 presents an implementation for global I/O optimization. The functions Unconstrained() and Constrained() implement the algorithms for the unconstrained and constrained layout cases respectively; constraints refers to a set that holds the array layout constraints and is updated after each nest is processed. Before the main loop, estimated cost of each nest \( i \) is computed using a metric and the nests are sorted according to non-increasing values of Cost(i) into Nest.List (e.g. Nest.List(1) is more expensive than Nest.List(2) etc.). Then the most expensive nest Nest.List(1) is optimized using Unconstrained() whereas the others are optimized inside the while loop using Constrained(). Note that besides returning new constraints both Unconstrained() and Constrained() also compute the necessary tiling loop transformation for the nest being processed; this is not shown in the figure.
for (each loop nest i)
    compute Cost(i);
endfor;
sort nests according to non-increasing values of Cost(i) into Nest.List;
Unconstrained(Nest.List(1),&new.constraints);
constraints = new.constraints;
while (there is a nest in the Nest.List)
    Current.Nest = next nest in the Nest.List;
    Constrained(Current.Nest,constraints,&new.constraints);
    constraints = constraints ∪ new.constraints;
endwhile;

Figure 24: A global I/O optimization algorithm.

7 Related Work

Previous works on compiler optimizations to improve locality have concentrated on iteration space tiling. Wolf and Lam [28] and Li [19] use iteration space tiling for optimizing cache performance of in-core programs. McKinley et al. [20] develop a simple model for optimizing cache locality. Schreiber and Dongarra [26] present reorganization of loops in order to create sub-matrix versions of algorithms automatically. In most of the previous works, locality enhancements for sequential programs are targeted. Our work deals with locality optimizations for out-of-core programs running on distributed-memory machines. Iteration space tiling was also used for purposes other than optimizing locality. In [16], an iteration space partitioning technique based on hyper-planes is introduced. In [25], the problem of compiling perfectly nested loops for distributed-memory message-passing machines is addressed. In [2] a solution to the problem of determining loop and data partitions automatically for programs with multiple loops and arrays is presented.

Prefetching, another compiler-directed optimization, is used by [21] and [22] for caches and by [23] for main memories. We believe that the compiler-directed prefetch is complimentary to our work in the sense that once the I/O time is reduced by our optimization, the remaining I/O time can be hidden by prefetching.

There has been a few papers on compilation of out-of-core programs. The approaches can be divided into two groups: The first group considers optimizing the performance of virtual memory (VM). The most notable work is from Abu-Sufah et al. [1], which deals with optimizations to enhance the locality properties of programs in a VM environment. Among the program transformations used are loop fusion, loop distribution and tiling (page indexing). In principle, our file layout and loop order determination schemes can be used for optimizing the performance of the VM as well (by changing tile sizes to take the page size into account). But we believe that the impact of the I/O optimizations based on VM will be limited as compared to that of the optimizations based on the explicit file I/O because of the following reasons:

- the fixed page sizes present a problem. Even if the computation requires a small portion of a data-tile, a full page containing the data is brought into memory. Or, conversely, even if there is enough bandwidth for fetching a number of pages, the VMs generally bring one or two pages after every page fault, wasting the bandwidth.

- the performance of the VM depends mostly on the page replacement policy of the operating system and this
policy is out of control of the compiler. In particular, even if a chunk of data will not be used any more, the replacement policy can keep that chunk in memory for a long time [23].

We should emphasize, however, that the performance of the a compilation technique based on the VM can be improved substantially by using compiler-hints to the operating system [23]. In UNIX, for example, the system calls like madvise, mmap etc. can be used for this purpose. The comparison of compilation techniques based on VM and explicit file I/O, however, is beyond the scope of this paper.

The second group of techniques is based on the explicit I/O approach. In [9], 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. Several issues that arise in the design and implementation of virtual-memory systems for data-parallel computing are investigated in [8]. In [24], the compiler support for handling out-of-core arrays on parallel architectures is discussed. In [4], a strategy to compile out-of-core programs on distributed-memory message-passing systems is offered. In [5], a methodology to optimize the communication in out-of-core computations is presented. It should be noted that our optimization techniques are general in the sense that they can be incorporated to any out-of-core compilation framework for parallel and sequential machines. We note that while the techniques offered in [4], [5] and [24] are specifically oriented for parallel machines, our technique can be used in uniprocessors as well.

Finally, in [6], a unified approach like ours that uses both data and control transformations for optimizing locality of in-core programs on distributed shared-memory machines is offered.

8 Conclusions

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 result in good performance. We proposed a three-step I/O optimization process by which the compiler can improve the code generated by the above method. The compiler estimates the I/O costs associated with different I/O access patterns and selects the method with the least I/O cost. This can reduce the amount of I/O by as much as an order of magnitude. It has been shown in this paper that instead of dividing the available memory equally among all arrays, the best performance is obtained when the most frequently accessed array(s) are allocated larger tiles.

Our work is unique in the sense that it combines data transformations (layout determination) and control transformations (loop permutations) in a unified compilation framework for optimizing out-of-core programs on distributed-memory message-passing machines. Our three-step approach however can also be used in uniprocessors as well. It has been shown in this paper that the combination of these two transformations leads to good memory allocations for different out-of-core arrays, and that in turn, minimizes the overall I/O time.
References


