Compilation Techniques for Out-of-Core Parallel Computations*

M. Kandemir
EECS Dept.
Syracuse University
Syracuse, NY, 13244
mtk@ece.nwu.edu

A. Choudhary
ECE Dept.
Northwestern University
Evanston, IL, 60208-3118
choudhar@ece.nwu.edu

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

R. Bordawekar
CACR
Caltech
Pasadena, CA 91125
rajesh@cacr.caltech.edu

Abstract

The difficulty of handling out-of-core data limits the performance of supercomputers as well as the potential of the parallel machines. Since writing an efficient out-of-core version of a program is a difficult task and virtual memory systems do not perform well on scientific computations, we believe that there is a clear need for compiler directed explicit I/O approach for out-of-core computations. In this paper, we first present an out-of-core compilation strategy based on a disk storage abstraction. Then we offer a compiler algorithm to optimize locality of disk accesses in out-of-core codes by choosing a good combination of file layouts on disks and loop transformations. We introduce memory coefficient and processor coefficient concepts to characterize the behavior of out-of-core programs under different memory constraints. We also enhance our algorithm to handle data-parallel programs which contain multiple loop nest. Our initial experimental results obtained on IBM SP-2 and Intel Paragon provide encouraging evidence that our approach is successful at optimizing programs which depend on disk-resident data in distributed-memory machines.

Keywords: out-of-core, data-parallelism, I/O, optimizing compilers, distributed-memory machines.

*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, an NSF grant CCR-9210422 and by the Louisiana Board of Regents through contract LEQSF (1991-94)-RD-A-09.

1 Introduction

It is well known that the I/O subsystems of the supercomputers and massively parallel machines constitute one of the major bottlenecks limiting the performance of those architectures. Many applications processing large quantities of data cannot obtain the desired performance due to disparity between the computational and I/O speeds of those machines. In order to alleviate this problem, over the decades, the computer scientists and engineers concentrated on I/O subsystem hardware, the virtual memory (VM) and related operating system (OS) concepts, and finally parallel file systems. Although each of those approaches contributed somewhat to alleviating the I/O problem, they all lacked one important parameter which limited their effectiveness: a global view of application's behavior.

In the architecture arena, the need for high-performance has prompted several manufacturers to develop parallel I/O subsystems. Although these subsystems have increased the I/O capabilities of the parallel machines significantly, a lot of improvement is still needed to balance the CPU performance. The problem has become more severe since the size and complexity of applications, both in scientific and commercial worlds, have increased tremendously [14]. As the data requirements of those complex applications exceed the capacity of main memories, the data needs to be stored on a variety of storage media, which include disks and tape drives. As the performance of other system components tends to improve rapidly, it is very likely that the storage subsystem performance of parallel machines will become increasingly important in the future.

In the software arena, the first related studies came from OS designers who offered the handling of I/O activity via a technique called virtual memory which also assumes a considerable amount of help from the hardware. The techniques on this line generally fall into two groups: (1) techniques which consider smart virtual memory implementations and replacement policies [2, 11]; and (2) techniques which consider re-shaping the data reference patterns in order to exploit the given hardware facilities and system software [23, 32]. The latter group then paved the way for automatic program restructuring techniques like loop distribution and page-indexing [1]. In general, these techniques apply to already written programs and consist of re-arranging the code and data to make program's access pattern more local.

Another system software based technique is built upon the file systems and run-time libraries. Several approaches considered extending the traditional Unix file I/O interface for handling the parallel accesses to parallel disk subsystems [18]. The problem with those approaches was that although the Unix-like semantics is convenient for the portability of the existent user applications, the file I/O interfaces derived from the Unix do not aim to get high I/O performance. More recently a number of parallel file systems both from the academia and industry have emerged [10]. Apart from presenting the user convenient interface for indicating parallel I/O accesses from within her program, those file systems also support several I/O modes which take care of the commonly seen file access patterns [28] in explicitly or implicitly parallelized scientific codes. These patterns can be characterized as regular and non-consecutive I/O accesses [18], and can be described by *strided* interfaces and their variants. Recently a number of run-time libraries for out-of-core computations and a few file interfaces have been proposed [31, 30, 8, 29]. SIO initiative [29] leaded by Caltech proposed a parallel file system programming interface which is based on separation of programmer convenience functions from high performance enabling functions. MPI-IO [8], on the other hand, is an attempt to provide a portable interface for parallel machines. Instead of defining common I/O access modes, the designers of MPI-IO chose to express data partitionings in terms of derived datatypes. Although most of the I/O libraries suffer from the portability problem, they still fill, to some degree, the need for software for out-of-core computations. In spite of the fact that the parallel file systems and run-time libraries for out-of-core computations provide considerable I/O performance, they require a considerable effort from the user as well. As a result, the user-optimized parallel I/O-intensive applications both consume precious time of the programmer who instead should focus on higher aspects of her program and also are not portable across a wide variety of parallel machines with different disk subsystems.

In this paper, we concentrate on compiler techniques to optimize the I/O performance of scientific applications. In other words, we give the responsibility of keeping track of data transfers between disk subsystems and memory to the compiler. The main rational behind this approach is the fact that the compiler is, sometimes, in position to examine the overall access pattern of the application, and can perform I/O optimizations which conform to application's behavior. Moreover, a compiler can establish a coordination with the underlying architecture, native parallel file system of the machine and I/O-libraries available so that the optimizations can obtain good speedups and execution times. An important challenge for the compiler approach to I/O on parallel machines is that the disk use, parallelism and communication (synchronization) need to be considered together to obtain a satisfying I/O performance. A compiler-based approach to the I/O problem should be able to restructure the out-of-core data and computations, insert calls to the parallel file systems and/or libraries, and perform some low-level I/O optimizations.

The remainder of this paper is organized as follows: In Section 2, we describe out-of-core computations. In Section 3 we present our data storage model used by compiler. We explain an out-of-core compilation strategy in Section 4, and in Section 5 we discuss several compilation issues for I/O and communication. We present a file locality optimization algorithm in Section 6. In Section 7, we report results of our experiments and in Section 8 we enhance our file locality algorithm to handle multiple loop nests. In Section 9, we summarize the related work and we conclude the paper in Section 10.

2 Out-of-Core Computations

A computation is called an *out-of-core* computation if the data that is used in the computation can not fit in the main memory. Thus, the primary data structures reside in files and this data is called *out-of-core* data. Processing out-of-core data, therefore, requires staging data in smaller granules that can fit in the main memory of a system. That is, the computation is carried out in several phases, where in each phase, part of the data is brought into memory, processed, and stored back onto secondary/tertiary storage media (if necessary).

Out-of-core computations can easily be classified as *physically* out-of-core or *algorithmically* out-of-core [3]. In a physically out-of-core computation, data required for the entire computation has to be fetched from files because the available memory is too small to hold the data structures. On the other hand, in an algorithmically out-of-core computation, the overall computation is performed in phases because the input data is received in parts at runtime. Examples of algorithmically out-of-core applications include real-time rendering application that processes a stream of images or a scientific application accessing data over a network. In this paper we concentrate on the compilation of physically out-of-core computations only.

Both types of out-of-core applications access data from files stored either in secondary (e.g., disk drives) or tertiary memory (e.g., data tapes). For the purposes of this paper, we only consider files stored in secondary memory. Moreover, we assume that the way a file is stored on disks is dependent on the underlying parallel file system, and the connection between the parallel file system and compiler is provided by our data store model which is explained in the following section.

3 Data Storage Model

Many new high performance multicomputers employ aggressive secondary storage subsystems. These subsystems may contain private disks, shared disks or a combination of both. This variety in the I/O architectures makes it difficult to design optimization techniques to reduce the time spent in I/O. We believe that in order to design techniques for achieving reasonable performance for programs using disk-resident data on multicomputers, the following questions should be addressed.

- Can a simple storage model abstracting out the details of the underlying disk subsystems be designed?
- On such a model, can the common locality optimizations be applied to programs manipulating the disk-resident data? What kinds of program and/or data restructurings are needed?
- How should the effectiveness of such locality optimizations be measured? What is the success criterion?
- What are the architectural bottlenecks preventing these optimization from succeeding on different disk subsystem architectures?

To address some of these problems, we designed and implemented an abstract storage subsystem called *local placement model* (LPM) [3] extending the distributed-memory paradigm to account for disk subsystem related issues. Our data storage model abstracts out the peculiarities of the underlying disk subsystem and presents the user or the compiler a system where the disk-resident arrays are divided into local disk-resident arrays, each of which is stored on separate logical private disk assigned to a processor. Within this framework, access to data residing in a non-local disk is performed via message passing. One of the advantages of such a model is that it enables the user and/or compiler to apply optimizations in order to improve the disk-locality characteristics of programs.

Based on this abstract model, we introduce control and data transformations to exploit the locality in files. Our control transformations change the access patterns to files and are based on a general loop transformation theory [34]. Our data transformations, on the other hand, involve assigning appropriate file layouts for different disk-resident arrays. In this way, we remove the impact of fixing layouts for all arrays as in conventional compiler and file systems. However, we should make a distinction between file and disk layouts. Depending on the storage style used by the underlying file system, a file can be striped across several disks. Accordingly, an I/O call in the program can correspond

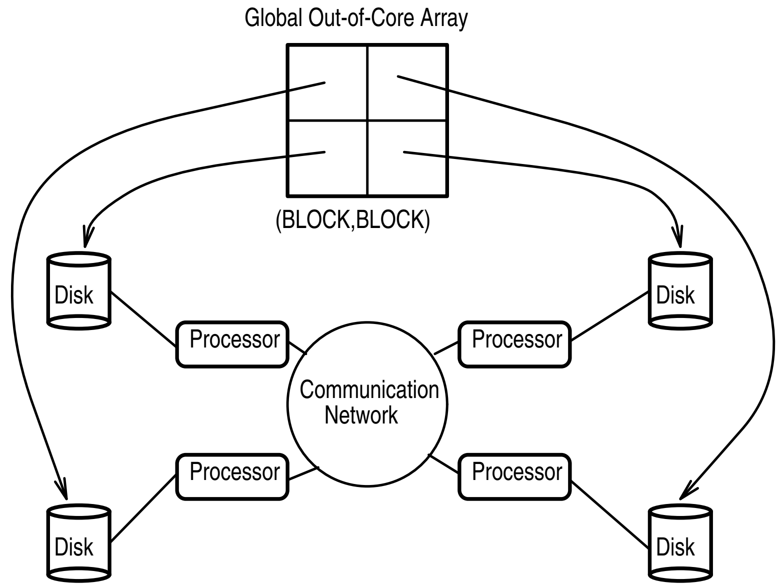

Figure 1: **Our storage subsystem abstraction - LPM.**

to several system calls to disk(s). The techniques described in this paper attempt to reach the optimized file layouts and to minimize the number of I/O calls to the files. Of course, reduction in I/O calls to files lead, in general, to reduction in calls to disks. The relation, though, is system dependent and is not addressed in this paper. Additionally, when more than one array is involved in a computation, we offer a memory allocation strategy such that the total I/O time is minimized.

Since our main objective is to minimize the time spent in I/O, commonly termed as *I/O cost*, we must have a measure to quantify the effectiveness of our optimizations statically for different programs. Practically, the cost of an I/O call can be modeled by two parameters: *startup cost*, which is the time taken to start a file I/O operation; and *read/write cost* of a datum. Since the startup cost is the dominating term, it should be amortized by transferring data between node memory and disk in large chunks. On the other hand, we can restructure the program so that once a data block is transferred into node memory, it will be reused as much as possible. Our optimizations achieve both of these goals. The reduction in number of I/O calls is computed, the overall effect of our approach at reducing the number of I/O calls and at reusing data is shown analytically. Our experimental results confirm these results.

In principle, our storage subsystem model can be implemented on any distributed-memory message passing machine. The data storage subsystem shown in Figure 1 specifies how the out-of-core arrays are placed on disks and how they are accessed by the processors. The local arrays of each processor are stored in separate files called *local array file*s. The local array file can be assumed to be *owned* by that processor. The node program explicitly reads from and writes into the local array file when required. Figure 2 shows how data in an out-of-core local array is tiled into memory.

In other words we assume that each processor has its own logical disk with the local array files stored on that disk. The mapping of the logical disk to the physical disks is system dependent. If a processor needs data from

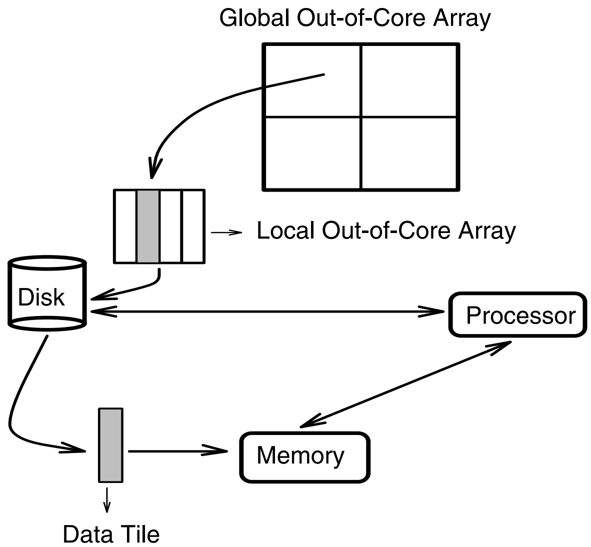

Figure 2: **Data tiling for a single processor.**

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. Since data sharing is performed by explicit message passing, this system is a natural extension of distributed-memory paradigm.

Designing I/O optimizations for architectures with different disk subsystems can be a very difficult task. Instead by building a storage subsystem abstraction, we isolate the peculiarities of underlying systems. An architecture with local disks is a straightforward extension of distributed-memory message-passing machine. The main advantages of such a system are scalability, ease of I/O programming and predictability of I/O optimizations. To elaborate more on the difficulty of designing efficient software optimizations, let us consider an I/O intensive data parallel program running on a multicomputer. The primary data sets of the program will be accessed from files stored on disks. Assume that the files will be striped across several disks. We can define four different *working spaces* [3] in which this I/O intensive parallel program operates: a *program space* which consists of all the data declared in the program, a *processor space* which consists of all the data belonging to a processor, a *file space* which consists of all the data belonging to a local file of a processor and finally a *disk space* which contains some subset of striping units belonging to a local file. An important challenge before the compiler writers for out-of-core computations is to maintenance the maximum degree of locality across those spaces. During the execution of I/O intensive programs, data needs to fetched from external storage into memory. Consequently, performance of such a program depends mainly on the time required to access data. In order to achieve reasonable speedups, the compiler or user needs to minimize the number of I/O accesses. One way to achieve this goal is to transform the program and data sets such that the localities between those spaces are maintained. This problem is similar to that of finding appropriate compiler optimizations to enhance the locality characteristics of in-core programs; but due to the complex interaction between working spaces it is more difficult. To improve the I/O performance, any application should access as much consecutive data as possible from disks. In other words, the program locality should be translated into spatial locality in disk space. Since maintaining the locality in

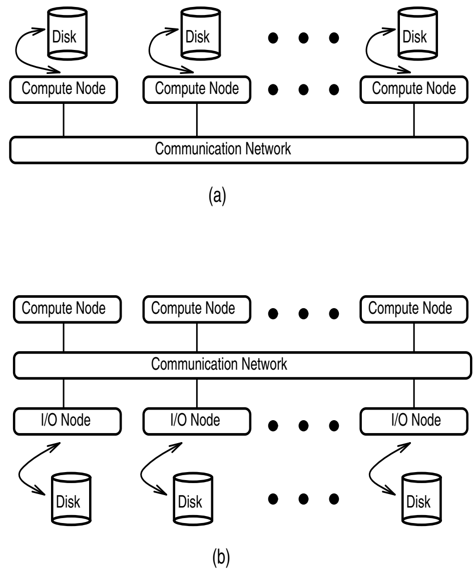

Figure 3: **Storage subsystems of SP-2 (a) and Paragon (b) as seen by LPM.**

disk space is very difficult in general, our compiler optimizations try to maintain the locality in the file space.

Throughout the paper we present out-of-core compilation methods and I/O optimization techniques that work on LPM and evaluate the performance of those techniques on two different distributed-memory machines: IBM SP-2 and Intel Paragon.

In order to implement the LPM on SP-2 we used the locally attached disk space to store the local array files as shown in Figure 3:(a). For Paragon, on the other hand, we created on the disk subsystem a separate file for each local array (Figure 3:(b)). We should note that under those implementations there is a strong parity between LPM and the underlying I/O architecture on the SP-2 whereas there is a disparity between those on the Paragon. Of course, those implementations are only representative, and the LPM can be implemented in a variety of ways on both Paragon and SP-2.

4 Out-of-Core Compilation Strategy

In this section we discuss the general issues involved in compilation of out-of-core codes on distributed-memory message-passing machines. Our programming model is inspired by the data-parallel programming paradigm. In essence, data-parallel programs apply the same conceptual operations to all elements of large data structures. This form of parallelism occurs naturally in many scientific and engineering applications such as partial differential equation solvers and linear algebra routines. In these programs, a decomposition of the data domain exploits the inherent parallelism and adapts it to a particular machine. Compilers can use programmer-supplied decomposition patterns such as block and cyclic to partition computation, generate communication and synchronization, and guide optimization of the program. Different data alignment and distribution strategies used in the decomposition affect the computational load balance, and amount of interprocessor communication, and allow other optimizations.

Several new languages provide directives that permit the expression of mappings from the problem domain to the processing domain to allow a user to express precisely these alignments and distributions. The compiler uses the information provided by these directives to optimize the programs. Languages based on this principle are called data-parallel languages and include High Performance Fortran (HPF) [17], Vienna Fortran [35], and Fortran D [13]. In this paper, we refer to all those languages as HPF-like languages, and to directives provided by them as HPF-like directives.

Our main assumption throughout this section is that HPF-like language directives are used to distribute the data across the processors. In out-of-core computations, however, these directives apply to data on disk(s). For example a directive such as `DISTRIBUTE X(BLOCK,BLOCK) ON TO P(2,2)` results in distribution of an out-of-core array `X` in a block-block manner across the local disks of four processors (see Figure 1). An alternative approach [27] is to employ two types of directives: one type for the decomposition of out-of-core data into in-core chunks; and another type for the distribution of an in-core chunk across the local memories of processors. The main disadvantage of this two-level mapping is that it specifies the tile access pattern of a node program completely, and hinders high-level optimizations to improve I/O performance. As will be shown in the rest of the paper, the determination of tile access pattern can be successfully achieved by a compiler analysis; so, our approach uses only one-level mapping.

The compilation of an out-of-core data-parallel program consists of two distinct phases. In the first phase, a global program analysis is performed. This part involves lexical analysis, file/data distribution analysis, dependence/data-flow analysis and a preliminary analysis of `DO` loops and `FORALL` statements. In fact, the out-of-core compilation in this phase proceeds in the same way as an in-core compilation would. The second phase operates on the local name space. Using the distribution information supplied by the user directives, the out-of-core arrays are partitioned across the local disks of the processors. Then the compiler performs work distribution which involves computing the local array sizes for each processor, scalarizing the `FORALL` statements and generating corresponding `DO` loops. Also in this step, the compiler detects necessary communication and generates the communication sets. The next step in this phase is to perform *tiling*. Tiling (also known as blocking) is a technique to improve the locality and is a combination of strip-mining and loop permutation [33, 34]. It can be used to automatically create blocked versions of programs, and when it is applied it replaces the original loop with two new loops: a tiling loop and an element loop. In an out-of-core compilation strategy based on explicit file I/O, tiling of out-of-core data into memory is mandatory, and compiler uses the results of the dependence analysis [34] performed in the first phase to determine whether or not tiling is legal. All necessary loop transformations are performed in order to ensure the legality of tiling. Notice that the fetch order of tiles in out-of-core computations determines the order of computation as well. What essentially being performed during tiling is strip-mining the local computation according to the available memory. The next step performs some layout and loop optimizations in order to minimize the time spent in I/O. This re-ordering of computation and file layout transformations may not be trivial and we discuss this issue in detail in Section 6. After the I/O optimizations are applied, several communication optimizations are performed. The main task of this step is to determine the communication points for specific data tiles within the program. As will be explained in the following section, apart

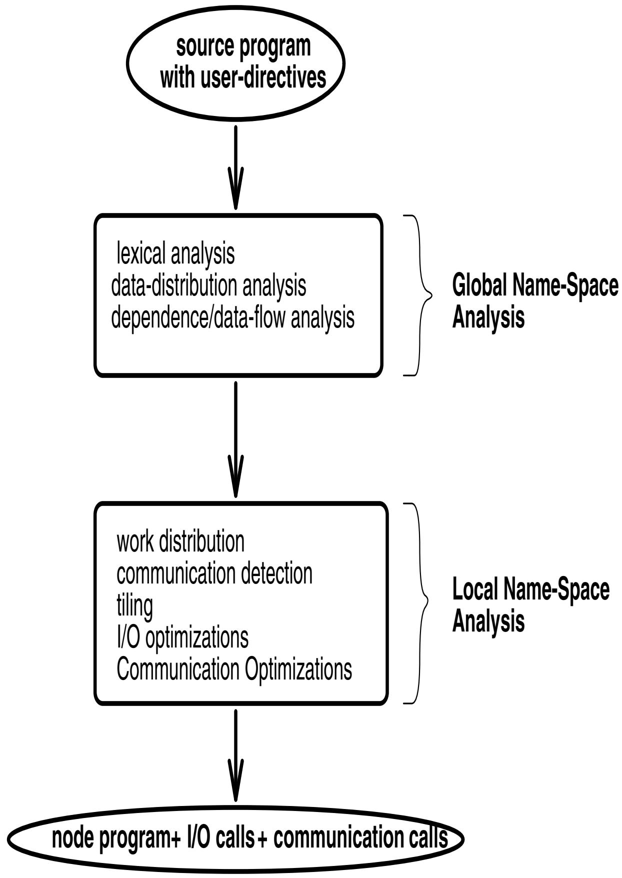

Figure 4: **Out-of-core compilation process.**

from reducing the time spent in communication, the communication optimizations also help to reduce the overall I/O time. This is especially true for the `FORALL`-type computations as they involve anti-dependence relations only. This gives the compiler the freedom to re-order the tile fetches such that the communication will only be performed for the data tiles currently residing in memory. The final step of this phase deals with the low-level I/O optimizations. This may involve pipelining of writes, aggregation of multiple I/O requests, and selection of the disk access strategy and I/O mode(s) taking into account the striping-type of the local data across disks. The strategies like two-phase access [3] and disk-directed I/O [19] can be used at this step, but in this paper we do not consider this last step anymore. The Figure 4 shows the overall compilation process.

5 Issues in Out-of-Core Compilation

In this section we summarize some of the previous work on out-of-core compilation and discuss several issues unique to out-of-core compilation.

An optimizing compiler based on explicit file I/O for out-of-core data parallel applications takes a data-parallel program (such as one written in HPF) accessing out-of-core arrays as input and generates the corresponding node

```
1         REAL A(1024,1024), B(1024,1024)
     ..........
2    !HPF$ PROCESSORS P(2,2)
3    !HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO P ::A,B
     ...........
4         FORALL (I=2:N-1, J=2:N-1)
5            A(I,J) = (B(I,J-1) + B(I,J+1) + B(I+1,J) + B(I-1,J))/4
6         END FORALL
```

Figure 5: **A program fragment for two-dimensional Jacobi computation.**

program with calls to runtime routines for I/O and communication. The compiler *strip-mines* the computation so that only the data which are in memory are operated on, and handles the required buffering. Computation on in-core data often requires data which is not present in a processor's memory requiring I/O access as well as communication. Since the data is stored on disks, communication often results in disk accesses. The disk access costs are normally several orders of magnitude greater than the inter-processor communication costs. Therefore, it is very important to reduce the disk access costs. In this section we summarize strategies to perform communication when data is stored on disks.

Consider the HPF program fragment from Figure 5. The HPF program achieves parallelism using data and work distribution. Work distribution is performed by the compiler during the compilation of parallel constructs like `FORALL` or array assignment statements (lines 4-6, Figure 5). A commonly used paradigm for work distribution is the *owner-computes* rule [34].

In this example, it can be observed that for the execution of the `FORALL` statement, each processor requires data from two neighboring processors. This communication requirement can be easily and efficiently satisfied for an in-core computation by collective communication routines such as *overlap shift*. In out-of-core computations, however, the compiler should make an important decision as how to perform the communication. The communication in our example can be satisfied by either a collective communication routine as in in-core compilation, or by considering the individual communication requirements of the data tiles in memory [3, 4, 5]. The latter performs communication only for the data tiles currently being processed in memory. Although the former communication method is attractive from compiler's point of view since it allows the compiler to easily identify and optimize collective communication patterns, it has been shown in [4] that in out-of-core computations involving transfer of the boundary data it is inefficient. In order for the latter method to be effective however, the compiler should be able to re-order processing of data tiles such that whenever a data item is required it should be available in some processor's memory rather than somewhere on disk. The details of the methods to obtain desired processing order can be found in [3, 5, 15].

During all this effort however, we have considered the I/O optimizations which are required because of the *communication requirements* of the out-of-core program. In the following section, we will concentrate on I/O optimizations which are required because of the *computation requirements* of the out-of-core programs. We will show that the main issue is to select the appropriate layouts, loop order and to allocate the available node memory such that the overall I/O time will be minimized.

6 I/O Optimizations for File Locality

Since accessing data on disks is usually orders of magnitude slower than accessing data on memory, the optimizing compilers must reduce the number as well as volume of the disk accesses. In this section we present:

- we present an algorithm based on explicit file I/O to reduce the time spent in disk I/O. Our algorithm automatically transforms a given loop nest to exploit spatial locality on files, assigns appropriate file layouts for arrays, and partitions the available memory among the data tiles of different out-of-core arrays, all in a unified framework.
- we present performance results for several kernels on an IBM SP-2 and on an Intel Paragon. These results provide enough evidence that the algorithm can be very useful for compilation of out-of-core codes.

As has been explained in the previous sections, 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. Abstractly, the compilation of an out-of-core loop nest can be thought of as consisting of two distinct phases. In the first phase, several analyses are performed, the out-of-core arrays in the source HPF program are partitioned according to the distribution information and bounds for local out-of-core arrays are computed. The second phase, on the other hand, adds appropriate statements to perform I/O and communication, and performs some I/O and communication optimizations. The local out-of-core arrays are first tiled according to the node memory available in each processor and the resulting tiles are analyzed for communication. The loops are then modified to insert necessary I/O calls. The techniques explained so far have not taken the file layouts into consideration. As will be shown in this section, however, some out-of-core computations perform best when different out-of-core arrays have different file layouts. This optimization is vital for generating an output code with satisfying I/O performance on large number of processors. The domain of our techniques is dense numerical out-of-core codes. This domain is appropriate because it is very important in practice.

Our research [16] has identified three major issues to be exploited in order to generate efficient code for out-of-core computations: access pattern, storage layouts on files, and memory allocation. The access pattern is generally a function of distribution directives and control constructs such as loops, conditional statements etc. Since in scientific computations most of the execution time is spent in loop nests, we can consider loops as the sole factor determining the access pattern along with the data distribution directives. On the other hand, the file layout for an h-dimensional array can be in one of the $h!$ forms, each of which corresponds to layout of data in file linearly by a nested traversal of the axes in some predetermined order. The innermost axis is called *the fastest changing dimension*. As an example, for row-major storage layout of a two-dimensional array, the second axis is the fastest changing dimension. And lastly, since node memory is a limited resource, it should be divided optimally among competing out-of-core arrays such that the total I/O time is minimized. A compiler for out-of-core codes should optimize the access pattern, storage layout and memory allocation together in order to exploit the locality as much as possible.

A naive approach can extend the compilation methodology of in-core data-parallel programs for out-of-core computations as follows: after the node program is determined, the loops are tiled (by considering data dependences) and

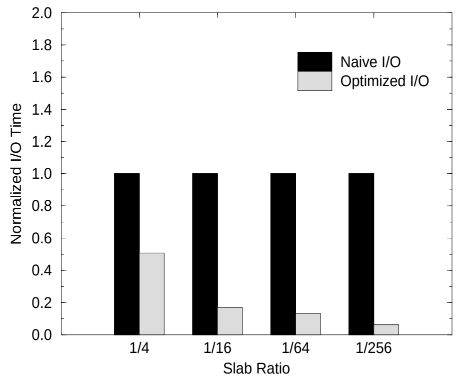

Figure 6: **Optimized I/O vs. Unoptimized (naive) I/O on IBM SP-2.**

appropriate I/O calls are inserted between tiling loops. If no layout optimization is performed, *data tiles* with sides of equal length in each dimension are fetched from disk, and the available memory is divided as evenly as possible across the arrays involved. We believe that the code produced by this naive method may not perform well due to the following reasons.

(1) In this method, the order of tiling loops is the same as that of the original loops. As will be shown later, for many nests encountered in practice that order of tiling loops may not be the best to exploit the data locality in files. In other words, the best loop order for file locality may be different from that for the register/cache locality.

(2) Assuming a fixed file layout such as row-major or column-major for all arrays involved in the computation may not be a good idea. In out-of-core computations, fixed file layout strategy can lead to poor results on secondary storage.

(3) When more than one array is involved in computation, during memory allocation it might be better to favor (by allocating more memory) the frequently accessed arrays over the others.

6.1 Why File Layout Optimizations?

Figure 6 shows the normalized I/O times of unoptimized and optimized versions of a loop nest on IBM SP-2. The loop nest (shown in Figure 9:(a)) accesses three out-of-core arrays, size of each is 4096×4096 double elements. The experiment was performed for different values of slab ratio (SR)[1], and in order to eliminate the effects of parallelism and isolate the improvement originating from the layout optimizations, we ran both versions of the program on a single node of SP-2. The following subsections will give detailed analysis in order to obtain appropriate file layouts for given a loop nest. For now, we should emphasize that the optimized version has different file layouts, access pattern and memory allocation than the unoptimized version, the combined effect of which is $50\% - 90\%$ improvement in the total I/O time.

[1] Slab ratio is the ratio of size of the in-core node memory to the total size of the out-of-core local arrays.

6.2 What is an optimized array access?

We refer to an array access as optimized if it can be performed such that all the data along a specific dimension will be read by a low I/O cost. Practically, the situation is depicted in Figure 7 for three different cases using a two-dimensional array. The shaded portions denote data tiles. The case in Figure 7:(a) corresponds to unoptimized case where all the available memory is utilized for accessing a square tile. In that case the file layout for the array does not matter as in order to read an $S_a \times S_a$ data tile, S_a I/O calls should be issued no matter what the file layout is. The cases shown in Figures 7:(b) and Figures 7:(c), on the other hand, correspond to the optimized accesses for row-major and column-major file layouts respectively. In Figures 7:(b), with S_b I/O calls it is possible to read $S_b \times n$ elements from the file, and in Figures 7:(c), $n \times S_c$ elements are read by issuing only S_c I/O calls (assuming for both the cases at most n elements can be read by a single I/O call). The technique described in the rest of the paper attempts to reach the optimized array accesses for all references. The following points should be noted. First, an optimized array access is only meaningful with the corresponding optimized file layout. For example, reading $S_b \times n$ elements from the array shown in Figures 7:(b) would cost n separate I/O calls (i.e. would not be optimized) if the array was stored in file as column-major. Second, in order to have a fair comparison we fix the available memory size (M) no matter how the arrays are optimized. As an example, for the cases shown in Figures 7, assuming this is the only array referenced in the nest, the equality ${S_a}^2 = S_b n = nS_c = M$ should hold.

6.3 Array Reference Matrices and Loop Transformation Basics

Our focus is on loops where both array subscripts and loop bounds are affine functions of enclosing loop indices. Given a loop nest with indices i_1,i_2,...,i_n that can be represented by a column vector $\vec{I}$, we define an *array reference matrix* $\mathcal{L}$ for each reference in that nest such that the array reference can be expressed in the form $\mathcal{L}\vec{I} + \vec{b}$ where $\vec{b}$ is an offset vector. In general there are as many array reference matrices as there are references. For our purposes, however, we consider only the array reference matrices which are distinct. On applying a transformation T to a loop nest denoted by $\vec{I}$, the transformed loop index becomes $T\vec{I}$ and the transformed array reference matrix $\mathcal{L}T^{-1}$. Similarly if $\vec{d}$ is the original distance/direction vector, after applying T, $T\vec{d}$ is the new distance/direction vector. A transformation is legal if and only if $T\vec{d}$ is lexicographically positive for every $\vec{d}$. We denote T^{-1} by Q. An important characteristic of our approach is that using the array reference matrices, the entries of Q are derived systematically. For the rest of the paper, the reference matrix for array X will be denoted by $\mathcal{L}^X$ whereas the i^{th} row of the reference matrix for array X will be denoted by $\vec{\ell_i}^X$.

6.4 Algorithm for Optimizing File Locality

Let i_1,i_2,...,i_n be loop indices of the original nest, and j_1,j_2,...,j_n be the loop indices of the transformed nest, starting from outermost loop. An explanation of our algorithm for a single statement follows. The modifications necessary for multiple LHSs and multiple nests are discussed in Section 8.

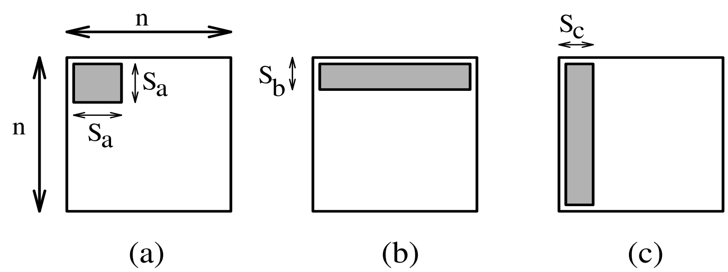

Figure 7: **(a) Unoptimized access. (b)-(c) Optimized accesses.**

Handling the LHS. At the first step, our technique determines a suitable transformed array reference matrix for the LHS reference. The transformation matrix should be such that the LHS array of the transformed loop should have the innermost index as the only element in one of the array dimensions and that index should not appear in any other dimension for this array. In other words, after the transformation, the LHS array C should be of the form $C(*, *, ..., j_n, ..., *, *)$ where j_n (the new innermost loop index) is in the r^{th} dimension and $*$ indicates a term independent of j_n. This means that the r^{th} row of the transformed reference matrix for C is $(0, 0, ..., 0, 1)$ and all entries of the last column, except the one in r^{th} row, are zero. After that process the LHS array *can* be stored in file such that the r^{th} dimension will be the fastest changing dimension. This helps to exploit the spatial locality in file, for this reference. Notice that after this step the out-of-core array is not stored in file immediately according to the determined layout. Instead, the final layout for the LHS array is decided after considering all alternatives.

Handling the RHSs. Then the algorithm works on one reference from the RHS at a time. If a row s in the data reference matrix is identical to r^{th} row of the original reference matrix of the LHS array, then this RHS array is attempted to be stored in file such that the s^{th} dimension will be the fastest changing dimension. We note however that having such a row s does not guarantee that the array will be stored in the file such that the s^{th} dimension will be the fastest changing dimension.

If the condition above does not hold for a RHS array A, in that case the algorithm attempts to transform the reference to $A(*, *, ..., \mathcal{F}(j_{n-1}), ..., *, *)$ where $\mathcal{F}(j_{n-1})$ is an affine function of j_{n-1} and other indices except j_n, and $*$ indicates a term independent of both j_{n-1} and j_n. This helps to exploit the spatial locality at the second innermost loop. If no such transformation is possible, the j_{n-2} is tried and so on. If all loop indices are tried unsuccessfully, then the remaining entries of Q are determined considering the data dependences and non-singularity. Li [21] discusses the completion of partial transformations derived from the data access matrix of a loop nest. The modified versions of his completion algorithms can be used for our purpose.

Choosing the best alternative. After a transformation and corresponding file layouts are found, they are recorded and the next alternative layout for the LHS is considered and so on. Among all feasible solutions, the best one is chosen. Although several approaches can be taken to select the best alternative, we found the following scheme both accurate and practical: Each loop in the nest is numbered with its level (depth), the outermost loop getting the number 1. Then, for each reference in the nest, the level number of the loop whose index sits in the fastest changing dimension for this reference is checked. The number for all references in the nest are summed up, and the alternative with the maximum sum is chosen. As an example, if, for a two-deep nest with three references, an alternative exploits the locality for the first reference in the outer loop, and for the other references in the inner loop, the sum for this alternative is $1 + 2 + 2 = 5$.

Memory allocation. After choosing the best alternative, our technique applies the following heuristic to partition the available memory among the competing array references: The algorithm divides the array references in the nest into groups according to file layouts of the associated files (i.e. the arrays with the same file layout are placed in the same group). The heuristic then handles the groups one by one. For each group, the compiler considers *all* fastest changing positions in turn. If a (tiling) loop index appears in the fastest changing position of a reference and does not appear in any other position (except the fastest changing) of any reference in that group, then it sets the tile size for the fastest changing position to n; otherwise it sets the tile size to S. The tile sizes for the remaining dimensions are also 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 node memory (M) into consideration and computes the actual value for S. For example, suppose that in a four-deep nest in which four two-dimensional arrays are referenced, the previous steps have 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 fastest-changing positions of $A[IT, KT]$ and $B[JT, KT]$, and does not appear in 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 the fastest-changing 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 algorithm 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 $S^2 + 3nS - M = 0$ for the positive S values. Note that any inconsistency between the groups (due to a common loop index) should be resolved by setting the tile size for the conflicting dimension(s) to S.

Our algorithm also takes care of the following points:

- After an alternative transformation matrix T (in fact, inverse of it) is computed, it is checked for *legality*. If there is at least one dependence distance/direction vector $\vec{d}$ such that $T\vec{d}$ is not lexicographically positive, then it is discarded.

- We also note that our approach is general and considers all possible layouts ($h!$ of them) for an h-dimensional array. The technique, however, does not consider *chunking*-a method by which the rectilinear blocks are stored in file consecutively- as it is very difficult to choose suitable chunks for multiple loop nests, given the current state of the optimizing compiler technology. Moreover, for singly-nested dense matrix codes we did not run into

Step 1 Initialize $i = 1$.

Step 2 Set $\vec{\ell_i}^{C}.Q = (0, 0, ..., 0, 1)$ and $\vec{\ell_k}^{C}.Q = (\delta, \delta, ..., \delta, 0)$ for each $k \neq i$, where δ denotes *don't-care*. The solution of those results in determination of some values of T^{-1}.

Step 3 Set the file layout of C for this alternative such that i^{th} index position will be the fastest changing position.

Step 4 For each array reference A on the RHS that has $\vec{\ell_l}^{A} = \vec{\ell_i}^{C}$ for some l, set the file layout of A for this alternative such that the l^{th} dimension will be the fastest changing dimension.

Step 5 Choose an array reference A for which the equality in **Step 4** does not hold. Initialize $j = 1$.

Step 6 Set $\vec{\ell_j}^{A}.Q = (0, 0, ..., 1, 0)$ and $\vec{\ell_k}^{A}.Q = (\delta, \delta, ..., \delta, 0, 0)$ for each $k \neq j$. If this step is consistent with the previous steps go to **Step 7**, otherwise increment j and go to the beginning of this step. If there exist inconsistencies for all j values, then initialize $j = 1$, and set $\vec{\ell_j}^{A}.Q = (0, 0, ..., 1, 0, 0)$ and $\vec{\ell_k}^{A}.Q = (\delta, \delta, ..., \delta, 0, 0, 0)$ for each $k \neq j$, and repeat **Step 6** and so on. If no T^{-1} is found then fill the remaining entries arbitrarily observing the dependences and non-singularity.

Step 7 Repeat **Step 6** for all reference matrices of a particular A (Of course, all reference matrices for a particular A should have the same distribution).

Step 8 Repeat **Step 6** for all distinct array references.

Step 9 Record the obtained transformation matrix. Also record, for each array, the loop index position which appears in the fastest changing position for that array.

Step 10 Increment i and go to **Step 2** (try a different layout for the LHS array C).

Step 11 Compare all the recorded transformation matrices and their associated layouts, and choose the best alternative (see the explanation in Section 6.4).

Step 12 Determine the memory allocations for the all out-of-core arrays in the nest and obtain the memory constraint in terms of S, n, p (number of processors) and M (size of the available node memory).

Step 13 Solve the memory constraint for S.

Figure 8: **Algorithm for optimizing locality in out-of-core computations.**

any case in which chunking could have produced better results than our approach as far as the number of I/O calls are concerned.

- As the reader might notice, our method is LHS-based. In other words, it first optimizes the LHS reference and then proceeds with the RHS references. Although, theoretically, this order is not necessary, in practice we found it to be useful as the data tiles for the LHS reference are both read and written whereas the data tiles for the RHS references are read only.

The algorithm is shown in Figure 8. In the algorithm, C is the array reference on the LHS whereas A represents an array reference from RHS. The symbol δ denotes the “don't care” condition. It should be emphasized that the **Step 2** and **Step 6** involve solving matrix equations.

6.5 Layout Constraints

The approach presented so far implicitly assumes that the file layouts for all out-of-core arrays can be set according to the output of the file locality algorithm. Unfortunately, this assumptions applies only to the case for which the compiler is to create arrays on disk from scratch as in the case of temporary array allocation or out-of-core version of an in-core program. In general, however, an out-of-compiler should be able to work with out-of-core arrays which have already been created on disk. To be specific, during the compilation of an out-of-core program either or both the following may be true: (a) the compiler does not have complete knowledge about the access pattern or storage layouts; (b) the compiler, due to data dependences or other constraints, is not able to change the access pattern or storage layout. Each unknown or unmodifiable information about access pattern or storage layouts constitutes a constraint for the compiler.

We now focus on the problem of optimizing locality when some or all array layouts are fixed, as this case frequently occurs in practice. We note that each *fixed layout* requires that the innermost loop index should be in the appropriate (corresponding) array index position (dimension), depending on the file layout of the array. For example, suppose that the file layout for a h-dimensional array is such that the dimension k_1 is the fastest changing dimension, the dimension k_2 is the second fastest changing dimension, k_3 is the third etc. The algorithm should first try to place the new innermost loop index j_n only to the ${k_1}^{th}$ dimension of this array. If this is not possible, then it should try to place j_n only to the ${k_2}^{th}$ dimension and so on. If all dimensions up to and including k_h are tried unsuccessfully, then j_{n-1} should be tried for the ${k_1}^{th}$ dimension and so on.

In general, given a loop nest, the layouts for some of the arrays may have already been determined, and the question is to determine the layouts of the remaining arrays, and find a loop order accordingly. This problem can easily occur, for example, in a program in which multiple nests access (overlapping) subsets of the out-of-core arrays declared in the program. It should be noted that solving this constrained-layout problem is less expensive than solving the general problem in which all possible layouts for all arrays should be evaluated. As we will show later, this modified algorithm with the constrained layouts is very important for global I/O optimization.

6.6 Example

In order to illustrate the performance improvement that can be obtained by compiler optimizations aiming file layouts over the naive explicit I/O approach, we consider the nest shown in Figure 9:(a), assuming that arrays A, B and C are $n \times n$ out-of-core arrays. In the *naive* translation, after obtaining the node program, the compiler tiles all four loops and inserts I/O statements between tiling loops. Of course, it should also check for legality of tiling; but, for sake of this explanation we do not consider legality as an issue and assume that tiling is always legal. A sketch of the resulting code is given in Figure 9:(b), assuming n is an exact multiple of S, the tile size; and p is the number of processors. In this example, using HPF-like distribution 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. We remind the reader that in out-of-core computations, compiler directives apply to data on (logical) disks. In the translated code the loops u, v, w and y are called *tiling loops* and the computation inside the tiling loops is performed on data tiles (sub-matrices) rather

```
DO i = 1, n
  DO j = 1, n
    DO k = 1, n
      DO l = 1, n
        A(i,j)+=B(k,i)+C(l,k)
      ENDDO l
    ENDDO k
  ENDDO j
ENDDO i
```

(a)

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

(b)

```
DO u = 1, n/p, S
  read data tile for A[u,1:n]
  read data tile for B[1:n,u]
  DO v = 1, n, S
    read data tile for C[v,1:n]
    A[u,1:n]+=B[1:n,u]+C[v,1:n]
  ENDDO v
  write data tile for A[u,1:n]
ENDDO u
```

(c)

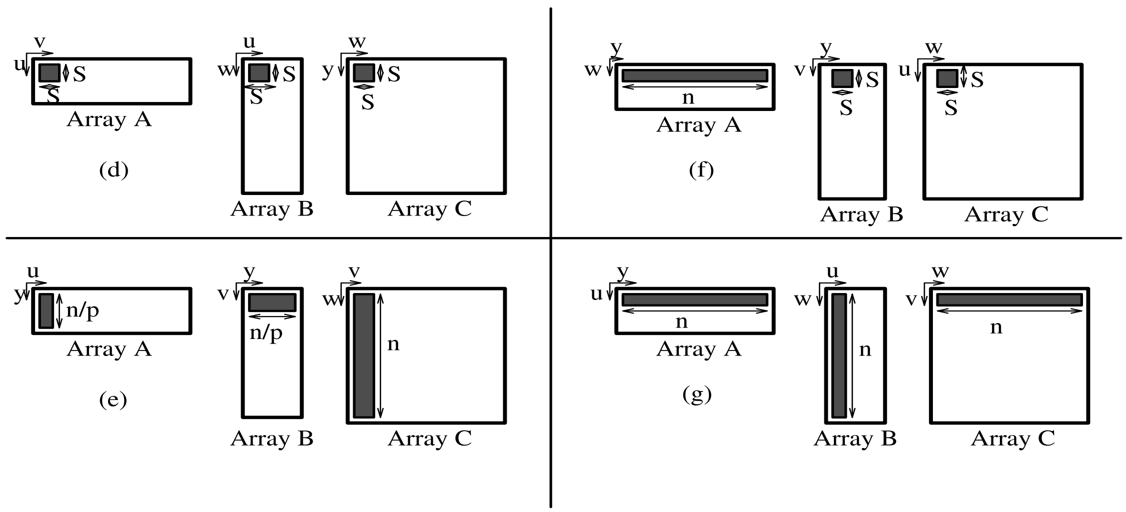

Figure 9: **(a) An out-of-core loop nest. (b) Straightforward translation. (c) I/O optimized translation. (d)-(g) Different tile allocations.**

than individual array elements. In other words, there are four more loops called *element loops* (not shown for sake of clarity) that iterate over the individual elements of data tiles of A, B and C. It should be emphasized that a reference such as $A[u, v]$ denotes a data tile of size $S \times S$ from file coordinates (u, v) as upper-left corner to $(u+S-1, v+S-1)$ as lower-right corner. A reference like $A[u, 1:n]$, on the other hand, denotes a data tile of size $S \times n$ from $(u, 1)$ to $(u + S - 1, n)$, i.e. a block of S consecutive rows of the out-of-core matrix A.

With Figure 9:(b), during the execution, square tiles of size $S \times S$ (shown as shaded blocks in Figure 9:(d)) are read from files. Note that this tile allocation scheme implies the memory constraint $3S^2 \leq M$ where M is the size of the available node memory.

The array reference matrices are as follows:
$L^A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$, $L^B = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$ and $L^C = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$. The algorithm works as follows: First it considers column-major disk layout for A.

$L^A.Q = \begin{bmatrix} 0 & 0 & 0 & 1 \\ \delta & \delta & \delta & 0 \end{bmatrix}$. Therefore $q_{11} = q_{12} = q_{13} = q_{24} = 0$ and $q_{14} = 1$.

$L^B.Q = \begin{bmatrix} \delta & \delta & \delta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$. Therefore $q_{34} = 0$.

$L^C.Q = \begin{bmatrix} \delta & \delta & 1 & 0 \\ \delta & \delta & 0 & 0 \end{bmatrix}$. Therefore $q_{33} = q_{44} = 0$ and $q_{43} = 1$. At this point $T^{-1} = Q = \begin{bmatrix} 0 & 0 & 0 & 1 \\ q_{21} & q_{22} & q_{23} & 0 \\ q_{31} & q_{32} & 0 & 0 \\ q_{41} & q_{42} & 1 & 0 \end{bmatrix}$.

We set the unknowns to the following values: $q_{22} = q_{23} = q_{31} = q_{41} = q_{42} = 0$ and $q_{21} = q_{32} = 1$, and obtain $T^{-1} = Q = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$.

The resulting code is as follows:

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

Arrays A and C are column-major whereas the array B is row-major. By using our memory allocation scheme explained earlier, a tile of size $n/p \times S$ is allocated for A, of size $n \times S$ for C, and of size $S \times n/p$ for B. The final memory constraint is $2nS/p + nS \leq M$. [2] Tile allocations are shown in Figure 9:(e).

Next the algorithm considers the other alternative layout (row-major) for A.

$L^A.Q = \begin{bmatrix} \delta & \delta & \delta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$. Therefore $q_{14} = q_{21} = q_{22} = q_{23} = 0$ and $q_{24} = 1$.

$L^B.Q = \begin{bmatrix} \delta & \delta & 1 & 0 \\ \delta & \delta & 0 & 0 \end{bmatrix}$. Therefore $q_{13} = q_{34} = 0$ and $q_{33} = 1$.

$L^C.Q = \begin{bmatrix} \delta & \delta & 0 & 0 \\ \delta & \delta & 1 & 0 \end{bmatrix}$. Therefore $q_{43} = q_{44} = 0$. At this point $T^{-1} = Q = \begin{bmatrix} q_{11} & q_{12} & 0 & 0 \\ 0 & 0 & 0 & 1 \\ q_{31} & q_{32} & 1 & 0 \\ q_{41} & q_{42} & 0 & 0 \end{bmatrix}$. By

[2] Notice that the value of S for each of the four cases in Figure 9 is different. It is computed by taking into account the memory constraint.

setting $q_{12} = q_{31} = q_{32} = q_{41} = 0$ and $q_{11} = q_{42} = 1$, $T^{-1} = Q = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$.

The resulting code is as follows.

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

The arrays A and C are row-major whereas the array B is column-major. Tiles of size $S \times n$ are allocated for A and C, and a tile of size $n \times S$ is allocated for B. The final memory constraint is $3 \times n \times S \leq M$. Tile allocations are shown in Figure 9:(g). Notice that since tile sizes are equal to array sizes for some array dimensions, the loops w and y disappear and the optimized code given earlier in Figure 9:(c) is obtained. Also note that although we could have chosen one of these two alternatives as the best alternative by using our criterion based on loop levels as explained earlier, we have presented here the tile allocations for both the alternatives for illustrative purposes.

We now consider the example shown in Figure 9 once more, this time assuming fixed row-major file layouts for all arrays. Due to lack of space, we do not show the formulation; but after our constraint-based algorithm is run, the tiles of size $S \times S$ are allocated for B and C, and a tile of size $S \times n$ is allocated for A. The final memory constraint is $nS + 2S^2 \leq M$. Tile allocations for this constrained version are shown in Figure 9:(f). Note that the two array references in this constrained version are not optimized due to the fixed layout requirement. This is the main reason that the locality optimizations adopted by out-of-core compilers should take into account both file layouts and loop transformations.

6.7 Analytical Formulation

Essentially the reduction (as will be shown in the Experimental Results section) in the I/O costs of the optimized program come from the decrease in both the number of I/O calls issued from within the program and the volume of the data transferred between disk and memory. In this subsection, using an analytical model, we evaluate the effectiveness of the algorithm presented in this paper at reducing the number of I/O calls. In order to get a simple formulation, we assume that at most n elements can be accessed in a single I/O call, where n being the array size in one dimension and the loop upper bound. Let C_f, t_f and M be the file I/O startup cost, the cost of reading (writing) an element from

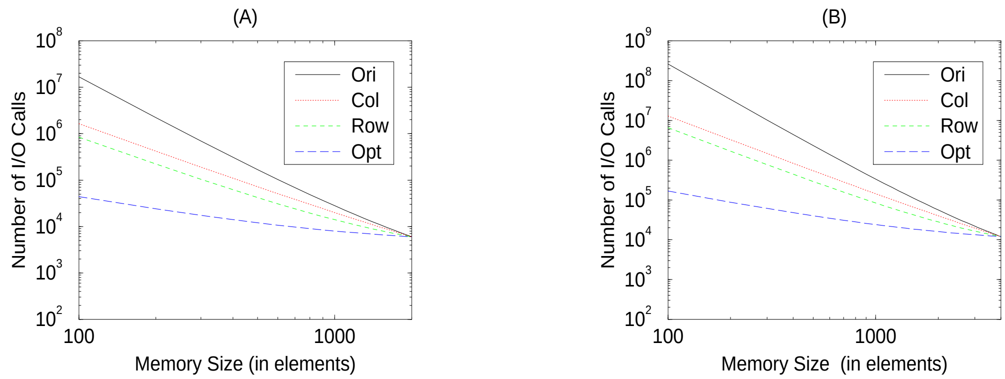

Figure 10: **(a) Number of I/O calls for the example shown in Figure 9 for $2K \times 2K$ double arrays. (b) Number of I/O calls for the example shown in Figure 9 for $4K \times 4K$ double arrays.**

(into) file, and the size of the available node memory respectively. If we assume that I/O cost of reading (writing) l consecutive array elements from (into) file can be approximated by $C_f + l \times t_f$, then the total I/O cost of the naive out-of-core translation shown in Figure 9:(b) (*excluding write costs*) is

$$T_{overall} = \underbrace{\frac{n^2 C_f}{S} + n^2 t_f}_{T_A} + \underbrace{\frac{n^3 C_f}{S^2} + \frac{n^3 t_f}{S}}_{T_B} + \underbrace{\frac{n^4 C_f}{S^3} + \frac{n^4 t_f}{S^2}}_{T_C}$$

under the memory constraint $3S^2 \leq M$. The T_A, T_B and T_C represent the I/O costs for the arrays A, B and C respectively. The overall I/O cost of the I/O optimized version (Figure 9:(c)), on the other hand, is

$$T_{overall} = \underbrace{nC_f + n^2 t_f}_{T_A} + \underbrace{nC_f + n^2 t_f}_{T_B} + \underbrace{\frac{n^2 C_f}{S} + \frac{n^3 t_f}{S}}_{T_C}$$

under the memory constraint $3nS \leq M$. The overall I/O costs for the versions with fixed layouts are computed similarly. The write costs can safely be neglected as the writes are performed very infrequently as compared to the reads and, in principle, they can always be pipelined. These formulae clearly show that, for reasonable values of M, our algorithm is very effective at reducing the number as well as the volume of disk accesses. We computed the number of I/O calls for different input sizes for the example in Figure 9. Figures 10:(a) and (b) illustrate the curves representing the number of I/O calls (coefficient of C_f) for $2K \times 2K$ and $4K \times 4K$ double arrays respectively on logarithmic scales. We consider four different versions: unoptimized (naive) version (Ori), optimized version assuming fixed column-major layout for all arrays (Col), optimized version assuming fixed row-major layout for all arrays (Row), and the version that is optimized by our approach (Opt). It should be emphasized that the curves for the optimized versions are overestimates, because of the fact that we assume at most n elements can be read in a single I/O call. The effectiveness of our approach at reducing the number of I/O calls is clearly reflected on these curves. For example, for $4K \times 4K$ double arrays with a memory size of 10^3 elements , the number of I/O calls required for Ori, Col, Row and Opt versions are approximately 5×10^5, 2×10^5, 1.5×10^5 and 1.5×10^4 respectively.

Table 1: **I/O times (in seconds) for the example shown in Figure 9 on SP-2 and Paragon.**

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

Table 2: **Number of bytes read and number of I/O calls issued for our example.**

SR=1/4	$2K \times 2K$ double arrays				$4K \times 4K$ double arrays			
	Ori	Col	Row	Opt	Ori	Col	Row	Opt
Number of bytes read	2.35×10^8	2.35×10^8	1.34×10^8	1.34×10^8	9.40×10^8	9.40×10^8	5.37×10^8	5.37×10^8
Number of I/O calls issued	28672	20480	14336	8192	57344	40960	28672	16384

7 Preliminary Results

The experiments were performed by hand using PASSION [30], a run-time library for parallel I/O. PASSION routines can be called from C and Fortran, and different out-of-core arrays can be associated with different file 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 total size of all out-of-core local arrays. Notice that the SR is both an abstract variable and a good indicator of the behavior of the out-of-core programs under different memory constraints.

Table 1 shows the I/O times of four different versions (Ori, Col, Row and Opt as explained above) of the example shown in Figure 9 on single nodes of IBM SP-2 and Intel Paragon.

The Table 2 shows the number of bytes read, and number of I/O calls issued for each of the four versions. It is easy to see that the Opt version minimizes both the number of I/O calls in the program and the number of bytes transferred from the disk subsystem, and that explains the reduction in the overall I/O time.

Tables 3 and 4 show the I/O times for the example shown in Figure 9 with $4K \times 4K$ (128 MByte) double arrays and different number of processors on SP-2 and Paragon respectively.

Figure 11 gives the speedups for Ori (original, unoptimized) 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 Ori 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 12:(a)).

From these results we conclude the following:

(1) The Opt (I/O optimized) version performs much better than the Ori (naive) version. The reason for this result is that in the Opt version, array accesses are optimized as much as possible. For example as shown in Figure 9:(g), by using the I/O optimization technique presented in this paper all three array accesses are optimized.

(2) When the slab ratio is decreased, the effectiveness of our approach (Opt) increases. As the amount of node memory is reduced, the Ori performs many number of small I/O requests, and that in turn degrades the performance

Table 3: **I/O times for the example shown in Figure 9 with $4K \times 4K$ (128 MByte) double arrays on IBM SP-2.**

Number of Processors = 4				
SR	Original	Col-Opt	Row-Opt	Opt
1/4	202	161	119	66
1/16	672	412	354	152
1/64	5193	2411	2196	490
1/256	29504	16904	12291	1689

Number of Processors = 8				
SR	Original	Col-Opt	Row-Opt	Opt
1/4	154	117	93	39
1/16	427	359	288	87
1/64	2858	1955	1327	272
1/256	21026	8901	7505	941

Number of Processors = 16				
SR	Original	Col-Opt	Row-Opt	Opt
1/4	99	79	61	21
1/16	286	271	259	48
1/64	1874	1654	1141	160
1/256	16719	6221	5687	522

Table 4: **I/O times for the example shown in Figure 9 with $4K \times 4K$ (128 MByte) double arrays on Intel Paragon.**

Number of Processors = 4				
SR	Original	Col-Opt	Row-Opt	Opt
1/4	1840	1546	1220	410
1/16	2889	2063	1094	536
1/64	17251	12128	9877	1233
1/256	123545	88246	80940	5043

Number of Processors = 8				
SR	Original	Col-Opt	Row-Opt	Opt
1/4	1186	996	786	221
1/16	1862	1241	985	286
1/64	11120	9976	8100	701
1/256	79631	56780	51160	2695

Number of Processors = 16				
SR	Original	Col-Opt	Row-Opt	Opt
1/4	810	670	600	135
1/16	1271	998	851	188
1/64	7590	5698	5012	474
1/256	52995	35330	30994	1701

significantly. The Opt version, on the other hand, continues with the optimized I/O no matter how small the node memory is.

(3) As shown in Figure 11, the Opt version also scales better than the Ori (original, unoptimized) for all slab ratios.

(4) Experiments on two different platforms (Paragon and SP-2) with varying compile-time/run-time parameters such as available node memory, array sizes, number of processors etc., demonstrate that our algorithm is quite robust.

7.1 Memory Coefficient

Suppose that a problem of a specific size is solved in t_0 seconds with a slab ratio SR using the Ori version (i.e. without any optimization). In principle the same problem can be solved in $\leq t_0$ seconds with a smaller slab ratio SR' (i.e. less memory) using the Opt version on the same number of processors. We call the ratio SR/SR' *memory coefficient* (MC) [16]. The larger this coefficient the better it is, because it indicates the reduction in the memory requirements

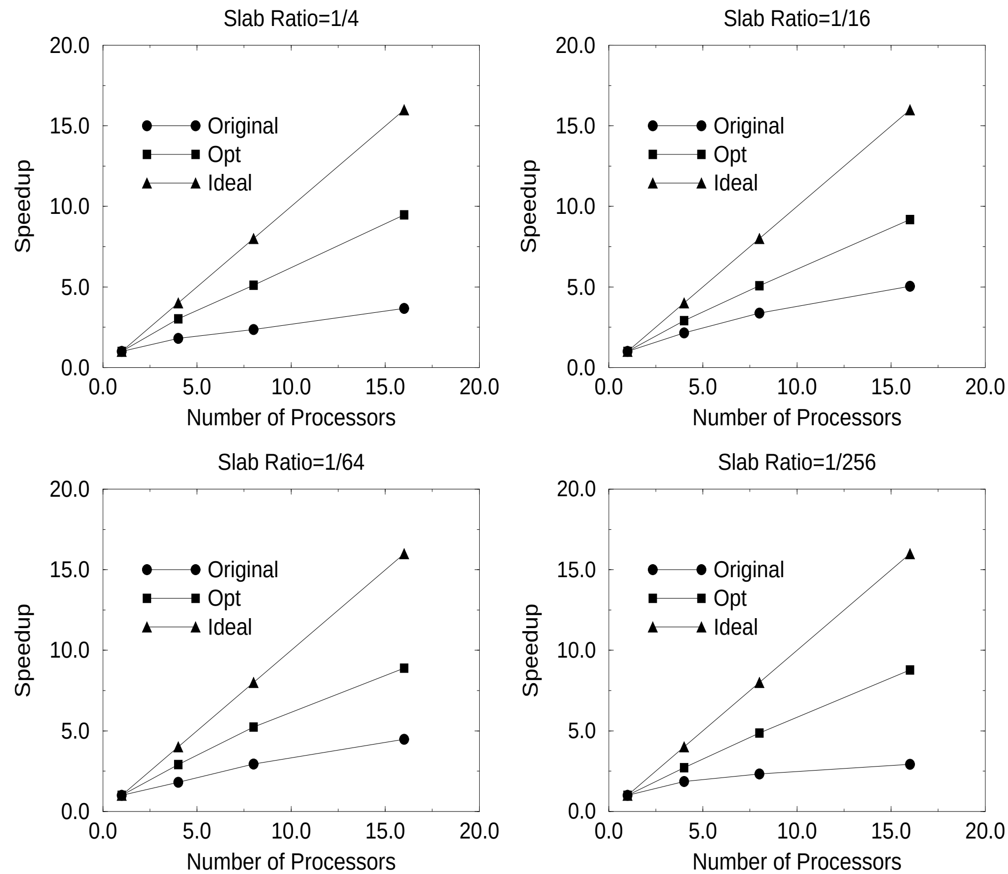

Figure 11: **Speedups for unoptimized and optimized versions of our example with $4K \times 4K$ double arrays on SP-2.**

of the application. During our experiments with different out-of-core nests, we noticed that the MC is usually a fixed value for a given loop nest. As an example, Figure 12:(c) presents the MC values for the example shown in Figure 9 for different slab ratio (SR) values. As can be seen, MC is almost always 4. Practically this indicates that it is possible to solve the same problem, at the same or less time, with $1/4$ of the original memory by using the Opt version instead of the Ori, the straightforward translation. Of course, the value of the MC depends on the computation under consideration and can be approximated by using the analytical formulation presented earlier. During the experiments, we found that if the original SR value is large, MC can be unpredictable in some cases (e.g. in Figure 12:(c), for SR=1/4 on Paragon). Since the Opt version can solve the same problem using less memory than Ori, we believe that our algorithm is especially useful for multiprogramming systems.

7.2 Processor Coefficient

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). Figure 12:(b) shows the PC curves for our running example with $4K \times 4K$ (128 MByte)

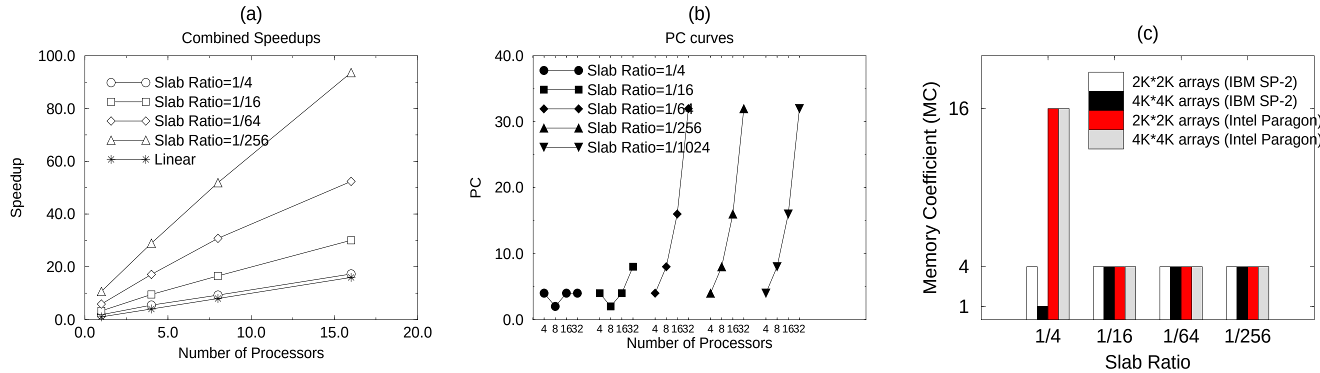

Figure 12: **(a) Combined Speedups for our example with $4K \times 4K$ double arrays. (b) PC curves. (c) MC values.**

double arrays 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 12:(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. An intuitive explanation for this is that, in the unoptimized case, beyond a slab ratio, the program performs so badly that increasing the number of processors does not help much; and the same execution time can easily be obtained by one or two processors using the Opt version.

We believe that the final PC curves and MC values give enough information about the performance of I/O optimizations in a distributed-memory environment.

8 Global I/O Optimization

In this section, we show how our algorithm can be extended to work on multiple nests. Notice that a single loop nest with multiple LHSs (i.e. multiple statements) can be handled in a similar manner if we think the nest as if it is distributed [34] over the individual statements.

Since a number of arrays can be accessed by a number of nests and these nests may require different layouts 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 we present sketch of a simple heuristic. Our approach is based on the concept of *most costly nest*, the nest which takes the most I/O time. The programmer can use *compiler directives* to give hints about this nest, or we can 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 costly nest. A more aggressive approach should consider all references in program for a particular array globally. Since it is unclear to us at this point whether compiler itself can or should select the most costly nest, we do not discuss this issue further. The rest of the algorithm is independent from how the most costly nest is determined.

The algorithm proceeds as follows: First, the most costly nest is fully optimized by using the algorithm presented in this paper. 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 6.5. After each nest is

```
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 13: **A global I/O optimization algorithm.**

optimized, new layout constraints will be obtained, and these will be propagated for optimization of the next nest.

Figure 13 shows an algorithm 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 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 costly than Nest_List(2) etc.). Then the most costly nest Nest_List(1) is optimized using Unconstrained() whereas the others are optimized inside the **while** loop using Constrained(). Notice that besides returning new_constraints both Unconstrained() and Constrained() also compute the loop transformation for each nest (not shown for clarity).

We note that the global I/O optimization problem is very similar to the problem of determining the appropriate data decomposition across the processors for multiple nests [20]. We are currently investigating whether or not the techniques developed for the data decomposition problem can be applied for the file layout optimization for multiple nests.

9 Related Work

Iteration space tiling has been used for optimizing the cache locality in several papers [21, 33]. McKinley *et al.* [24] proposes an optimization technique consisting of loop permutation, loop fusion and loop distribution. The assumption of a fixed layout strategy prevents some array references getting optimized as shown earlier in this paper, and that in turn may cause a substantial performance loss. But as indicated earlier, after our approach is applied, we strongly recommend the use of an in-core locality optimization technique for data tiles in memory.

In [7], a unified approach to locality optimization which employs both control and data transformations is presented for in-core problems in distributed shared-memory machines. This model can be adapted to out-of-core computations as well. But we believe our approach is better than that of [7] because of the following reasons:

- The approach given in [7] depends on a *stride vector* whose value should be guessed by the compiler beforehand. Our approach does not have such a requirement.

- Our approach is more accurate, as it does not restrict the search space of possible loop transformations whereas the approach in [7] does.

- Our extension to multiple nests (global optimization) is also simpler than the one offered by [7] for global optimization, because we use the same algorithm for each nest.

In [16], the authors present a heuristic for optimizing out-of-core programs consisting of a single loop nest. In the future, we intend to implement the algorithms in [7] and [16], enhance them by a suitable memory allocation scheme for out-of-core computations and compare them with the approach presented in this paper on different programs.

Another compiler-directed optimization, prefetching, is used by [25] for caches and by [26] 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 out-of-core compilation. Some of them consider 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). It should be emphasized that in principle, our file layout determination scheme can be applied for optimizing the performance of the VM as well (by changing tile sizes to take the page size into account). Recently Malkawi *et.al.* [22] and Gornish *et.al.* [12] have proposed compiler based techniques to obtain good performance from memory hierarchy.

In [9], the functionality of a ViC*, a compiler-like preprocessor for out-of-core C* is described. Several compiler methods for out-of-core HPF programs are presented in [30] and [4]. In [27], compiler techniques to choreograph I/O for applications based on high-level programmer annotations are investigated. The techniques in [30] and [27] are specifically designed for parallel machines whereas our approach can also be used on uniprocessors. To our knowledge, non of the previous work on out-of-core computations considers the compiler-directed file layout transformations.

10 Conclusions

The difficulty of efficiently handling out-of-core data limits the performance of supercomputers as well as the enormous potential of parallel machines. Since coding out-of-core version of a problem might be a very onerous task and virtual memory does not perform well in scientific programs, we believe that there is a need for compiler-directed explicit I/O approach for parallel architectures. Unfortunately, fixed layout strategies adopted by popular compilers prevent the potential spatial locality in files from being exploited.

In this paper, after discussing some important issues on out-of-core compilation, we presented a compiler algorithm to optimize the locality on files by changing the access pattern and file layouts. Our technique can easily be employed as a part of an out-of-core compilation framework like [9], [27] or [30]. An important characteristic of the approach is that it also optimizes the nests with arrays whose layouts are constrained. We show that this is important for global I/O optimization as well.

An area of future work is integrating this technique with the techniques designed to eliminate I/O costs originating from communication requirements of out-of-core parallel programs [4, 5, 6]. We also intend to employ the algorithm presented in this paper in in-core compilers, and investigate its effectiveness at determining memory layouts and in improving cache performance.

References

[1] W. Abu-Sufah, D. Kuck, and D. Lawrie. On the Performance Enhancement of Paging Systems Through Program Analysis and Transformations. *IEEE Transactions on Computers*, C-30(5):341–356, May 1981.

[2] L. A. Belady. A Study of Replacement Algorithms for a Virtual Storage Computer. In *IBM Systems Journal*, 5 (1996), pages 78-101.

[3] R. Bordawekar. Techniques for Compiling I/O Intensive Parallel Programs, Ph.D. Dissertation, ECE Dept., Syracuse University, Syracuse, NY, May 1996.

[4] R. Bordawekar, A. Choudhary, and J. Ramanujam. Compilation and Communication Strategies for Out-of-core programs on Distributed Memory Machines. In *Journal of Parallel and Distributed Computing*, 38:(2), November, 1996.

[5] R. Bordawekar, A. Choudhary, and J. Ramanujam. Automatic Optimization of Communication in Compiling Out-of-core Stencil Codes. In *Proc. 10th ACM International Conference on Supercomputiing,* Philadelphia, PA, May 1996.

[6] R. Bordawekar, A. Choudhary, and J. Ramanujam. A Framework for Integrated Communication and I/O Placement. In *Proc. EUROPAR'96,* Lecture Notes in Computer Science, Springer-Verlag, August 1996.

[7] M. Cierniak, and W. Li. Unifying Data and Control Transformations for Distributed Shared Memory Machines. Technical Report 542, Dept. of Computer Science, University of Rochester, NY, November 1994.

[8] P. Corbett, D. Fietelson, S. Fineberg, Y. Hsu, B. Nitzberg, J. Prost, M. Snir, B. Traversat, and P. Wong. Overview of the MPI-IO parallel I/O interface, In *Proc. Third Workshop on I/O in Parallel and Distributed Systems*, IPPS'95, Santa Barbara, CA, April 1995.

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

[10] T. H. Cormen, and D. Kotz. Integrating theory and practice in parallel file systems. In *Proceedings of the 1993 DAGS/PC Symposium*, pages 64-74, Hanover, NH, June 1993.

[11] P.J. Denning. The Working Set Model for Program Behavior. In *Comm. of the ACM*, Vol. 11, No. 5, May 1968.

[12] E. Gornish, E. Granston, and A V. Veidenbaum. Compiler Directed Data Prefetching in Multiprocessors with Memory Hierarchies. In *Proc. ACM Int'l Conf. on Supercomputing*, pp 354–368, Amsterdam, The Netherlands, 1990.

[13] S.Hiranandani, K.Kennedy, and C.Tseng. Compiling Fortran D for MIMD distributed-memory machines. In *Comm. of the ACM* 35,8, pages 66-80, Aug. 1992.

[14] High Performance Computing and Communications: Grand Challenges 1993 Report. *A Report by the Committee on Physical,Mathematical and Engineering Sciences*, Federal Coordinating Council for Science, Engineering and Technology.

[15] M. Kandemir, R. Bordawekar, A. Choudhary, and J. Ramanujam. A Unified Tiling Approach for Out-of-core Computations, *In the Sixth Workshop on Compilers for Parallel Computers*, December 1996.

[16] M. Kandemir, R. Bordawekar, and A. Choudhary, Data Access Reorganizations in Compiling Out-of-Core Data Parallel Programs on Distributed Memory Machines. In *Proc. the Int'l. Parallel Processing Symposium (IPPS)*, pp 559–564, Geneva, Switzerland, April 1997.

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

[18] D. Kotz. Multiprocessor file system interfaces. In *Proceedings of the Second International Conference on Parallel and Distributed Information Systems*, pages 194-201, 1993.

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

[20] U. Kremer. Automatic Data Layout for Distributed Memory Machines. Ph.D. thesis, Technical Report CRPC-TR95559-S, Center for Research on Parallel Computation, October 1995.

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

[22] M. Malkawi, and J. Patel. Compiler Directed Memory Management Policy for Numerical Programs. In *Proc. ACM Symposium on Operating Systems Principles (SOSP'85)*, 1985.

[23] A. C. McKellar, and E. G. Coffman. The Organization of Matrices and Matrix Operations in a Paged Multiprogramming Environment, In *Comm. ACM* 12, 3 (March 1969), pages 153-165.

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

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

[26] T. C. Mowry, A. K. Demke, and O. Krieger. Automatic Compiler-Inserted I/O Prefetching for Out-of-Core Applications. *Proc. Second Symposium on Operating Systems Design and Implementations*, Seattle, WA, October 1996, pages 3-17.

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

[28] A. Purakayastha, C. S. Ellis, D. Kotz, N. Nieuwejaar, and M. Best. Characterizing parallel file-access patterns on a large-scale multiprocessor. In *Proceedings of the Ninth International Parallel Processing Symposium*, pages 165-172, April 1995.

[29] The Scalable I/O Low-level API: A Portable Programming Interface for Parallel File Systems. Presentation in *Supercomputing'96*, Philadelphia, PA, 1996.

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

[31] S. Toledo, and F. G. Gustavson. The Design and Implementation of SOLAR, a Portable Library for Scalable Out-of-Core Linear Algebra Computations. In *Proc. of the Fourth Annual Workshop on I/O in Parallel and Distributed Systems*, May 27, 1996, Philadelphia, PA, pages 28-40.

[32] K. S. Trivedi. On the Paging Performance of Array Algorithms. *IEEE Transactions on Computers*, C-26(10):938-947, October 1977.

[33] 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.

[34] M. Wolfe. High Performance Compilers for Parallel Computing. Addison-Wesley Publishing Company, CA, 1996.

[35] 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.