Improving locality in out-of-core computations using data layout transformations

M. Kandemir[1] A. Choudhary[2] J. Ramanujam[3]

[1] EECS Dept., Syracuse University, Syracuse, NY 13244 (mtk@ece.nwu.edu)
[2] ECE Dept., Northwestern University, Evanston, IL 60208 (choudhar@ece.nwu.edu)
[3] ECE Dept., Louisiana State University, Baton Rouge, LA 70803 (jxr@ee.lsu.edu)

Abstract. Programs accessing disk-resident arrays, called out-of-core programs, perform poorly in general due to an excessive number of I/O calls and insufficient help from compilers. In order to alleviate this problem, we propose a data layout optimization in this paper. Experimental results provide evidence that our method is effective for out-of-core nests whose data sizes far exceed the size of memory.

1 Introduction

An important characteristic of out-of-core computations is that the large data structures they access do not fit in main memory. Consequently, the data has to be stored on disk, and brought into memory only when they are to be processed. The time spent on these disk accesses is the primary determinant of the performance of out-of-core programs. Two main issues related with these large data structures are (1) when and how to transfer them between different levels of memory hierarchy, and (2) how to operate on them. It is certainly true that over the years, the I/O subsystems of parallel machines have evolved from simple-minded architectures that reserve a single processor for I/O to sophisticated systems where a number of I/O nodes can collectively perform I/O on behalf of all compute nodes. However, in general, it has been difficult to exploit these capabilities.

Several aspects of of the software support system for out-of-core computations have been addressed so far. Most of the previous research has focused on operating systems, parallel file systems, run-time libraries and applications themselves. In particular, run-time libraries and parallel file systems have received a lot of attention recently, resulting in a number of powerful and fast run-time libraries as well as a few commercial parallel file systems. In spite of these advances, it is very important for the user to exploit this capacity; inserting calls to library functions is tedious, error-prone and results in programs whose performance varies widely from machine to machine. Even in cases where a user is allowed to convey semantic information to the library, the programmer faces the burden of ensuring that what is provided is indeed correct.

In this paper we offer a less widely studied alternative to deal with the I/O problem at the software level. Instead of relying on programmer-supplied information to the run-time libraries and parallel file systems, we use techniques based on compiler analysis to obtain high-level information about the data access behavior of the scientific codes. After the information is collected, the compiler can plug it into a run-time system and/or parallel file system. Thus our approach is based on the idea of dividing responsibility between compiler and run-time system. This, in principle, should work as current optimizing compilers for shared and distributed memory machines are successful in detecting the data access behavior of scientific programs. Using the relevant information, a compiler can restructure the code such that I/O will be less of a bottleneck. In low-level terms, this restructuring of code should result in two improvements: first, the number of accesses to the disk subsystem should be reduced; secondly, the volume of data transferred between memory and disk system should be reduced. Overall, these reductions will lead to reduced I/O overhead, which in turn, hopefully, will reduce the execution time of the codes.

The rest of this paper is organized as follows. In Section 2, we elaborate on why cache memory oriented techniques may not work well for I/O intensive applications, and motivate the need for data-oriented techniques to improve the performance. In Section 3 we present an overview of our data restructuring framework. In Section 4, we give preliminary experimental results on three example kernels. In Section 5 we discuss related work and conclude the paper in Section 6.

2 Existing techniques

Our main goal is to improve the file locality exhibited by out-of-core programs. Of course, a simple way of looking at this problem is to think of the disk subsystem as yet another level in the memory hierarchy, and apply the code restructuring techniques [7] available for optimizing the performance of cache–main memory hierarchy. While in some cases this approach can work reasonably well, the observation of the following fact is important: The techniques developed for optimizing cache–main memory performance are access pattern oriented; that is the main goal there is to change the access pattern of the program (specifically the order of the loops in programs) to obtain an optimized program such that the majority of data accesses is satisfied from cache instead of memory. Improving data locality *indirectly* by changing the access pattern is problematic for main memory–disk hierarchy because of two points:

(1) The applications that use memory–disk hierarchy frequently are I/O intensive. They do I/O either because the data that they handle are too big to fit in the memory as in out-of-core computations or the data should be written into (read from) disk in regular intervals as in checkpointing, or visualization or a combination of both. The main issue here is the data itself; that is, the restructuring techniques should focus more on the data (instead of the access pattern) and seek ways to improve the transfer of data between disk and memory.

(2) Restructuring techniques for cache memories are in general limited by data dependences. That is, data dependences may prevent some locality optimizations from taking place. In the case of cache memories, this is fine for many programs. But for I/O intensive programs, traditional loop restructuring techniques are not sufficient since the penalty for going to the disk instead of memory can be very severe.

Thus, it is important for restructuring techniques for I/O intensive programs to be "data-oriented" rather than "control-oriented" (e.g., program dependences). With these issues in mind, we offer a program restructuring technique based on data layout transformations for out-of-core dense matrix computations. Specifically, our technique detects the optimal layouts in files and memory for out-of-core arrays referenced in a program. The overall effect is a much better I/O performance in terms of the number of I/O calls, the I/O volume and hence the execution time.

In order to explain the problem with loop transformation based techniques in detail, we consider *loop permutation* [7], a well-known transformation to improve locality in loop nests. Assuming column-major layouts, the locality of the code in Figure 1(a) will be poor, because successive iterations of the innermost loop touch different columns of the array. Loop permutation can improve the performance by *interchanging* the order of the loops; the resulting code shown in Figure 1(b) has good locality. Unfortunately, loop interchange may not be possible in every case. Consider the code in Figure 1(a) again, this time assuming that there is another right hand side reference `A(i-1,j+1)`. In this case, the data dependence [7] due to this and the left hand side reference prevents loop interchange.

Our objective is to derive an optimized out-of-core version of a program from its in-core version. Specifically, we would like to determine optimal file layouts for the arrays referenced in the program. The input to our framework is a program annotated (by user or compiler) using compiler directives that indicate which arrays are out-of-core. The output is a layout–optimized program which includes I/O calls to a run-time system as well as communication calls (for message-passing machines).

Since the size of an out-of-core array is larger than the memory available, the array should be divided into chunks such that each chunk can fit into memory. At a time, for a given array, only a single data chunk can be brought from file into memory, can be operated on and (if it is modified) stored back in file. In handling a data chunk, there are two main issues: (1) how to read/write it from (into) file efficiently; and (2) how to process it efficiently. The first issue is very important as file I/O is much more expensive as compared to memory or processor operations in terms of latency. An efficient compilation framework should minimize the file access time. This can be achieved by reducing the number of I/O calls as well as the volume of the transferred data. Reducing the number of I/O calls is more important, because it is the dominant factor. The second issue is related to the efficient use of memory which is a valuable resource in out-of-core computations. In essence, the elements of a data chunk that are brought into memory should be reused as much as possible before they are stored back on file. There is no point in bringing a data chunk when the ongoing computation will use only a part of it.

To see how the compilation of out-of-core arrays is handled, we again consider the program in Figure 1(a), assuming that the reference `A(i-1,j+1)` is also there. Suppose that `A` is an $n \times n$ out-of-core array and resides in a file with a column-major layout and brought into memory in chunks whose sizes and orientations are dictated by the innermost loop. Assuming that `M` is the size of memory allocated for this array and at most `n` consecutive items can be read/written in a single I/O call, to read a chunk of size $(M/n) \times n$ from file into memory (as shown in Figure 2(a)) will entail `n` I/O calls, each of which is for a sub-column of data of size `M/n`. For the entire execution of the code, n^3/M I/O calls should be issued. Of course, one might think of bringing a chunk of size $n \times (M/n)$ (instead of size of $(M/n) \times n$) by issuing only `M/n` I/O calls (instead of `n`) for it (see Figure 2(b)). Unfortunately, this is not very useful, as due to the access pattern, most of the data in this chunk will not be assigned new values as they are themselves dependent on the data which will be residing in file. The source of the problem here is that the data should not be stored as column-major in file. Rather, if it is stored as row-major as illustrated in Figure 2(c), then an $(M/n) \times n$ size data chunk can be brought into memory and operated on by issuing only `M/n` I/O calls. Therefore, it is possible to minimize the number of I/O calls by changing the file layout of the data.

To see the difference in performance between column-major and row-major file layouts quantitatively, consider Figure 3. The figure on the left shows (on a log scale) the number of I/O calls issued by the two versions (column-major and row-major file layouts) for different input sizes (n) between 2K and 10K elements. The memory size (M) is fixed at 4M elements for illustrative purposes. We note that when the input size is increased, the gap between the number of I/O calls in the two versions widens. The figure on the right illustrates the number of I/O calls when the memory size is varied and the input size is fixed at 4K elements. As one would expect, 16M elements is the minimum memory size to accommodate all the data. These figures indicate that with *a careful choice of file layouts,* huge savings can be obtained in the number of I/O calls issued in an application.

More importantly, file layout transformations, in most cases, do not get affected by data dependences; because they do not change the access order, they just rename the data points (array elements). This gives the file layout transformations a capability to optimize some loop nests where iteration space based optimization techniques fail due to existing data dependences.

We note that what we have done for this nest is *tiling*, a well-known transformation technique [7]. However, instead of tiling the iteration space first and then focusing on the data requirements of individual tiles, we first tile the data space and then execute all the iterations that assign new

```
DO i = 2, n
  DO j = 2, n
    A(i,j) = A(i-1,j) + A(i,j-1)
  END DO
END DO
```

(a)

```
DO j = 2, n
  DO i = 2, n
    A(i,j) = A(i-1,j) + A(i,j-1)
  END DO
END DO
```

(b)

Fig. 1. (a) Original program. (b) Transformed program. [The transformed program exhibits good spatial locality for column-major memory layout].

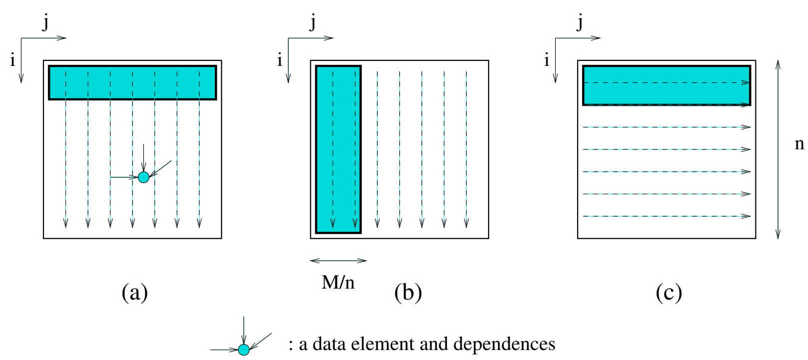

Fig. 2. Different memory layouts of the array accessed in Figure 1. [The shaded block in each figure denotes a data tile (chunk). The dashed arrows indicate the storage order of the array. Except near the boundaries, each data element is dependent on three others. An example data element and its dependences are also shown].

values to the elements of the tile currently stored in memory. Our file layout transformation allows us to read an $(M/n) \times n$ data tile by issuing just `M/n` I/O calls. In this example, a simple column-major to row-major layout conversion (i.e., a dimension permutation) is sufficient to optimize the layout. In general, a compiler may have to deal with more complex data layouts. In the next section, we focus on detecting optimal file layouts automatically by using a simple linear algebra techniques using hyperplanes.

3 Layout restructuring framework

In this section, we outline a method using which an optimizing compiler can restructure array layouts. The method was originally developed for in-core computations to determine memory layouts for optimal cache performance; but with appropriate modifications it can be adapted for out-of-core programs. In the out-of-core computation domain, this approach reduces the number of I/O calls as well as the volume of data transferred between disk and memory. The details of the approach can be found in [4]. In order to keep the presentation simple, we focus only on two-dimensional out-of-core arrays. However, our method works with arrays of any dimensionality.

3.1 Preliminaries

We focus on programs where array subscript expressions and loop bounds are affine functions of enclosing loop indices or constants. Under this assumption, each array reference to an `m`-dimensional array that occurs in a `k`-deep loop nest can be represented by $L\bar{I} + \bar{o}$, where the access (or reference) matrix L is of size `m` $\times$ `k` and the offset vector $\bar{o}$ is of size `m`. For example, the reference `A(i-1,j)` in Figure 1(a) can be represented by $L\bar{I} + \bar{o}$, where

$$L = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \bar{I} = \begin{pmatrix} \mathtt{i} \\ \mathtt{j} \end{pmatrix}, \bar{o} = \begin{pmatrix} -1 \\ 0 \end{pmatrix}$$

Notice that access pattern in the innermost loop is captured by the last column of L.

3.2 Hyperplanes and file layouts

In an m-dimensional data space, a *hyperplane* can be defined as a set of tuples

$$\{(a_1, a_2, ..., a_m) \mid g_1 a_1 + g_2 a_2 + ... + g_m a_m = c\}$$

where g_1, g_2,...,g_m are rational numbers called hyperplane coefficients and c is a rational number called hyperplane constant [6]. For convenience, we use a row vector $g^T = (g_1, g_2, ..., g_n)$ to denote an hyperplane family (for different values of c) whereas $\bar{g}$ corresponds to the column vector representation of the same hyperplane family. At least one of the hyperplane coefficients should be non-zero.

In a two-dimensional data space, the hyperplanes are defined by (g_1, g_2). We can think of a hyperplane family as parallel lines for a fixed coefficient set and different values of c. An important property of the hyperplanes is that two data points (array elements) (a, b) and (c, d) belong to the same hyperplane if

$$(g_1, g_2)\begin{pmatrix} a \\ b \end{pmatrix} = (g_1, g_2)\begin{pmatrix} c \\ d \end{pmatrix} \tag{1}$$

For example, $(g_1, g_2) = (0, 1)$ indicates that two elements belong to the same hyperplane as long as they have the same value for the column index (i.e., the second dimension); the value for the row index does not matter.

It is important to note that a hyperplane family can be used to partially define the file layout of an out-of-core array. In a two-dimensional case $(0, 1)$ is sufficient to indicate that the elements in a column of the array (i.e., the elements in a hyperplane with a specific c value) will be stored *consecutively* in file and will have *spatial locality*. The relative order of these columns are not important provided the array is large enough compared to the memory size which is the case in out-of-core computations. Similarly, the hyperplane vectors $(1, 0)$ and $(1, -1)$ correspond to row-major and diagonal file layouts, respectively.

3.3 Determining optimal file layouts

The following claim gives us a simple method to determine optimal file layout for a given reference to have spatial locality in the innermost loop (see [4] for the proof).

Claim 1. *Consider a reference $L\bar{I} + \bar{o}$ to a two-dimensional array inside a loop nest of depth k where*

$$L = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1k} \\ a_{21} & a_{22} & \cdots & a_{2k} \end{pmatrix}$$

In order to have spatial locality in the innermost loop, this array should have a layout represented by a hyperplane (g_1, g_2) such that $(g_1, g_2) \in Ker\{(a_{1k}, a_{2k})^T\}$.

Returning to our example in Figure 1(a), using this result, we have $(g_1, g_2) \in Ker\{(0,1)^T\}$ which means $(g_1, g_2) = (1, 0)$ is the spanning vector for that Ker set (null set). This vector corresponds to the row-major layout; therefore, in order to have spatial locality in the innermost loop, the array should have a row-major file layout

4 Preliminary results

In this section, we present performance results for three example programs: an out-of-core matrix transpose nest, an out-of-core matrix multiplication nest, and the Fast Fourier Transform (FFT). For each case, we report the I/O times on an Intel Paragon message-passing machine. The machine that we used had 56 compute nodes, 3 service nodes, and one HIPPI node. Each compute node is an Intel i860 XP microprocessor with 32 MBytes of memory.

Matrix transpose: Figure 4(a) shows the I/O times for an out-of-core matrix transpose nest which transposes an out-of-core array into another. Each array is of size 2048×2048 double elements and only 256 KBytes of the memory of each compute node is used. Note that this small amount of memory makes the problem out-of-core. We conducted experiments with 4, 8, and 16 processors. The results show that optimizing file layouts can lead to huge savings in I/O times. The super-linear speedups in this case are due to memory effects.

Matrix multiplication: Figure 4(b) shows the I/O times for an out-of-core matrix multiplication routine that computes `C = A` $\times$ `B`, where `A`, `B`, and `C` are 2048×2048 out-of-core matrices. The results here are not as impressive as those of the matrix-transpose example. The reason is that due to the access pattern of the loop nest, only two of the three arrays could be layout-optimized. The improvements are between 6% and 26%.

FFT: The fast Fourier Transform (FFT) is widely used in several scientific and engineering problems. The 2-D out-of-core FFT consists of three steps: (1) 1-D out-of-core FFT, (2) 2-D out-of-core transpose, and (3) 1-D out-of-core FFT. The 1-D FFT steps consist of reading data from a two-dimensional out-of-core array and applying 1-D FFT on each of the columns. Of course, in order to perform 1-D out-of-core FFTs the data on disk should be strip-mined into processors memory. After this, the processed columns are written to the file. In the transpose step, the out-of-core array is staged into memory, transposed and written to file. The innermost loop of the transpose routine uses two files that are accessed by all processors. In the original program, file layout for these two arrays is column-major. The transpose is performed by reading a rectangular data tile from one of the files, transposing it in the local memory, and writing it in the other file. Since both the files are column-major, optimizing the block dimension for one array has a negative impact on I/O performance of the other array, resulting in the poor I/O performance observed in Figure 5(a) and the poor overall performance shown in Figure 5(b) (for 2048×2048 double arrays). If we store one of the arrays in row-major order the I/O performance of both the arrays improve. This is evident from Figures 5(a) and (b).

5 Related work

Due to lack of space, we discuss related work on compilation of dense matrix codes which access out-of-core arrays. Brezany et al. [1] perform I/O optimizations in out-of-core compilation in a compilation framework with a runtime system called VIPIOS. The user provides hints for the

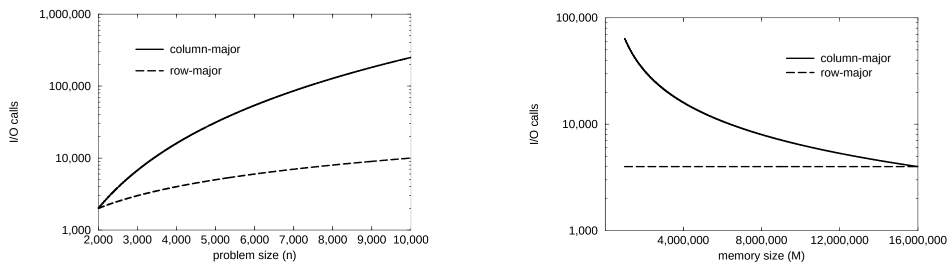

Fig. 3. The number of I/O calls in example shown in Figure 1 as a function of the input size (left) and the memory size (right).

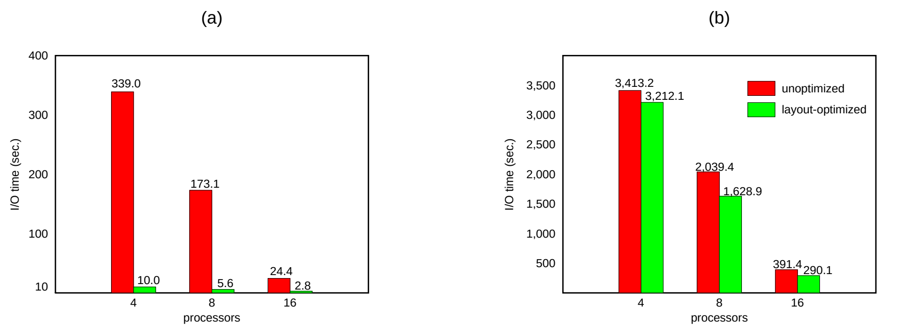

Fig. 4. I/O times (in seconds) on Intel Paragon for (a) matrix transpose and (b) matrix multiplication.

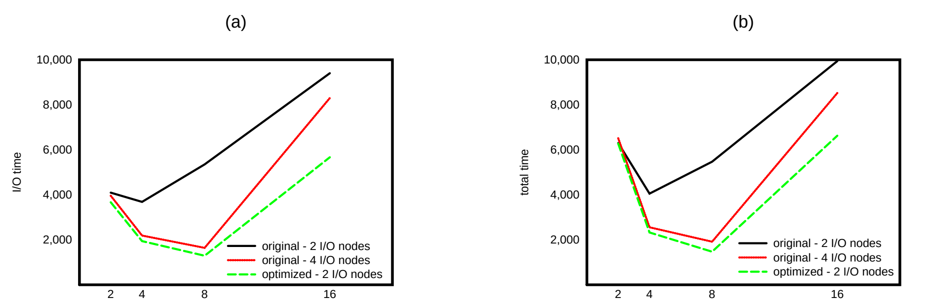

Fig. 5. Performance of the `FFT` on Intel Paragon (all times are in seconds).

I/O modes, I/O distribution etc., using the language constructs. Paleczny et al.[5] incorporate out-of-core compilation techniques with Fortran D. The main philosophy behind their approach is to choreograph I/O from disks along with the corresponding computation. Their idea, however, is based on computation re-ordering, and therefore is different from ours. Cormen and Colvin [2] introduce ViC* (Virtual C*), a preprocessor which transforms a C* program that uses out-of-core data structures into a program with appropriate library calls from the ViC* library in order to read/write data from/to disks. Finally, the previous work of the authors [3] considered unified loop and file layout transformations. However, the search space for possible file layout transformations was restricted to the dimension-wise transformations (e.g., from column-major to row-major). As against to that work, the approach presented in this paper uses a more powerful technique based on hyperplanes.

6 Summary and ongoing work

In this paper, we have presented an approach that can optimize file layouts of multi-dimensional out-of-core arrays. The approach can work with a large set of layouts which can be expressed using hyperplanes. In practice, the technique can optimize affine accesses to out-of-core arrays in two different yet complementary ways: (1) by reducing the number of file accesses, and (2) by reducing the data volume transferred between disk and main memory. The combined effect of these is a reduction in I/O as well as the overall execution time. The main issue that we are dealing with currently is to design a conflict resolution scheme that can decide near-optimal file layouts in cases where an out-of-core array is accessed with conflicting access patterns. This will allow us to apply our technique to longer programs. Another important issue is to combine computation reordering transformations with the layout optimization technique offered in this paper.

Acknowledgments A. Choudhary and M. Kandemir are supported in part by NSF Young Investigator Award CCR-9357840 and NSF CCR-9509143. J. Ramanujam is supported in part by an NSF Young Investigator Award CCR-9457768 and by an NSF grant CCR-9210422.

References

1. P. Brezany, T. Muck, and E. Schikuta. Language, compiler and parallel database support for I/O intensive applications, In *Proc. HPCN 95,* May 1995.
2. T.H. Cormen, and A. Colvin. ViC*: A preprocessor for virtual memory C*. Dartmouth College CS Tech. Rep. PCS-TR94-243, Nov. 1994.
3. M. Kandemir, A. Choudhary, J. Ramanujam, and R. Bordawekar. Compilation techniques for out-of-core parallel computations. *Parallel Computing,* 24(3-4):597–628, June 1998.
4. M. Kandemir, A. Choudhary, N. Shenoy, P. Banerjee, and J. Ramanujam. A data layout optimization technique based on hyperplanes. Tech. Rep. CPDC-TR-97-04, Northwestern Univ., Dec. 1997. A short version appears in *Proc. ACM International Conference on Supercomputing,* Jul. 1998.
5. M. Paleczny, K. Kennedy, and C. Koelbel. Compiler support for out-of-core arrays on parallel machines. CRPC TR 94509-S, Rice University, Dec. 1994.
6. J. Ramanujam, and P. Sadayappan. Compile-time techniques for data distribution in distributed memory machines. In *IEEE Trans. Par. & Dist. Sys.,* 2(4):472–482, Oct. 1991.
7. M. Wolfe. *High Performance Compilers for Parallel Computing,* Addison-Wesley, 1996.