Optimizing Out-of-Core Computations in Uniprocessors*

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

1 Introduction and Motivation

Due to the wide gap between processor and disk speeds in current parallel architectures, optimizing I/O performance presents a special challenge for designers of parallel systems. Current multicomputers either have virtual memory (VM) on each node as in Intel Paragon or do not have VM on nodes. Unfortunately, the studies of performance of VM using OSF/1 on Paragon have shown that the Paragon does not perform well on scientific programs due to frequent paging in and out of data [4]. Our experiments confirm this observation as shown in Figure 1 for a typical out-of-core loop nest accessing three out-of-core arrays of the same size. Figure 1 shows the performance improvement (in terms of normalized execution times) obtained by explicitly-optimized I/O against VM on a single node of an Intel Paragon. Since the Intel Paragon on which the experiments have been conducted has 32 MBytes node memory, three double-precision floating-point arrays each of which is $1K \times 1K$ represent the largest amount of data in our input set that can fit into memory. So, as expected, for arrays of size 512×512 and $1K \times 1K$, VM performs better. But starting from $2K \times 2K$ arrays, explicit I/O outperforms VM and the performance improvement is highly significant. Note that with the optimized approach, even for 512×512 and $1K \times 1K$ arrays, explicit I/O has been performed. This indicates that there is a clear need for optimized explicit I/O approach.

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

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

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

†EECS Dept., Syracuse University, Syracuse, NY 13244.
‡ECE Dept., Northwestern University, Evanston, IL 60208-3118
§ECE Dept., Louisiana State University, Baton Rouge, LA 70803.
¶CACR, Caltech, Pasadena, CA 91125.

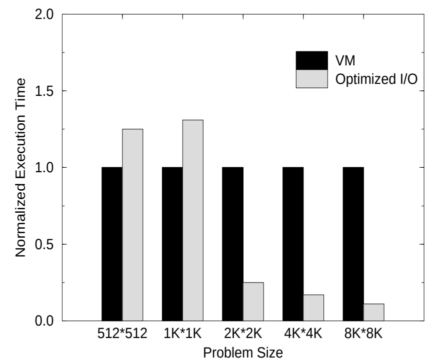

Figure 1: **Optimized I/O vs. VM on a single node of Intel Paragon.**

2 I/O Optimizations

This section presents the details of our three-step algorithm. Our objective is to find file layout(s) and a loop order that will enable us to partition the memory among references such that the total I/O cost is minimized. The problem is that finding the best order requires an exhaustive search of all possible permutations of tiling loops. Moreover, for all of these permutations all possible memory allocations should be evaluated. Instead we propose a three-step heuristic to solve this problem [1, 2]. First we need a few definitions.

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

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

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

Note that the $BCost$ function can be used in two different ways: (1) If IT is fixed and (R,$layout$) pair is changed, it gives the I/O cost induced by loop index IT for different array layouts. (2) If (R,$layout$) pair is fixed and IT is changed over all loop indices in the nest, then it gives the I/O cost induced by R with the associated $layout$. The following definition employs the latter usage, while the former is used in Section 2.2.

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

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

2.1 Layout Determination Heuristic

This heuristic first computes the $ACost$ values for all arrays under all the possible layouts. It then chooses the combination that will allow the compiler to perform efficient file access. $ACost$ values are listed term by term and each term corresponds to the $BCost$ of a loop index under the given file layout. Next our heuristic considers all possible layout combinations by summing up the *ACost* values for the components of each combination term by term.

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

After listing all possible layout combinations term by term, our layout determination algorithm chooses the combination with greatest number of *constant-order* and/or *S-order* terms. The reason for this choice is that each loop index with only *S-order* and/or *constant-order* terms will enable the out-of-core compiler to read all the data along the relevant dimension(s) with low I/O cost. If there are more than one combination with maximum number of constant-order and/or S-order terms, our heuristic chooses the one with the minimum cost. It should be noted that the optimum combination may not be unique.

2.2 Loop Permutation Heuristic

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

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

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

where R is the array reference and $layout_R$ is the layout of the associated file as determined in the previous step (Section 2.1).

Our algorithm for desired loop permutation is as follows:

- Calculate $TCost(IT)$ for each tiling loop IT,
- Permute the tiling loops from outermost to innermost position according to non-increasing values of $TCost$. It should be noted that the desired loop permutation should observe the existing dependences. If it does not, our heuristic keeps the loop order as it is, and applies the memory allocation scheme (Section 2.3) only. Another option is to try the next desirable loop permutation. Our choice is simpler and guarantees that the optimized nest will be at least as good as the original one.
- Apply necessary loop interchange(s) to improve the temporal locality for (the tile of) the reference being updated. This step prevents write operation from occurring in inner tiling loops. (Loop indices that do not have any n expression should be placed into innermost positions. Other loop indices however can be interchanged with one another if doing so promotes the temporal locality for the reference(s) being updated.)

2.3 Memory Allocation Heuristic

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

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

Our memory allocation heuristic works as follows.

The compiler divides the array references in the nest into two disjoint groups: a group whose associated files have row-major layout, and a group whose associated files have column-major layout. For row-major (column-major) layout group, the compiler considers *all* row-conformant (column-conformant) positions in turn. If a (tiling) loop index appears in the conformant position of a reference and does not appear in any other position (except the conformant) of any reference in that group, then it sets the tile size for the conformant position to n; otherwise it sets the tile size to S. After all the tile sizes for all dimensions of all array references are determined, our approach takes the size of the available memory (M) into consideration and computes the actual value for S.

As an example suppose that in a four-deep nest in which four two-dimensional arrays are referenced, our layout determination algorithm has assigned row-major disk layout for the arrays A, B and C, and column-major disk layout for the array D. Also assume that the references to those arrays are $A[IT, KT]$, $B[JT, KT]$, $C[IT, JT]$ and $D[KT, LT]$. Our memory allocation scheme divides those references into two groups: $A[IT, KT]$, $B[JT, KT]$, $C[IT, JT]$ in the row-major group, and $D[KT, LT]$ in the column-major group. Since KT appears in the row-conformant positions of $A[IT, KT]$ and $B[JT, KT]$, and does not appear any other position of any reference in *this group*, the tile sizes for A and B are determined as $S \times n$. Notice that JT also appears in a column-conformant position (of the reference $C[IT, JT]$). But since it also appears in other positions of some other references (namely in the first position of B) in this group, the compiler determines the tile size for $C[IT, JT]$ as $S \times S$. Then it proceeds with the other group which contains the reference $D[KT, LT]$ alone. Since KT is in the column-conformant position, and does not appear any other index position of D, the compiler allocates a data tile of size $n \times S$ for $D[KT, LT]$. After those allocations the final memory constraint is determined as $3 \times n \times S + S \times S \leq M$. Given a value for M, the value of S that utilizes all of the available memory can easily be determined by solving the second order equation $S^2 + 3nS - M = 0$ for the positive S values.

2.4 Empirical Results

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

Figure 3 shows the normalized I/O times for four different versions of a four-deep loop nest accessing three out-of-core arrays: Original version (Ori), I/O optimized version using column-major file layouts for all arrays (Col), I/O optimized version using row-major file layouts for all arrays (Row), and the version that is optimized by our approach (Opt).

From these results we conclude the following:

- For the examples presented, the Opt version performs much better than the Ori version. The reason for this result is that in the Opt version array accesses are optimized as much as possible.
- When the slab ratio is decreased, the effectiveness of our approach increases (see Figure 3). As the amount of memory is

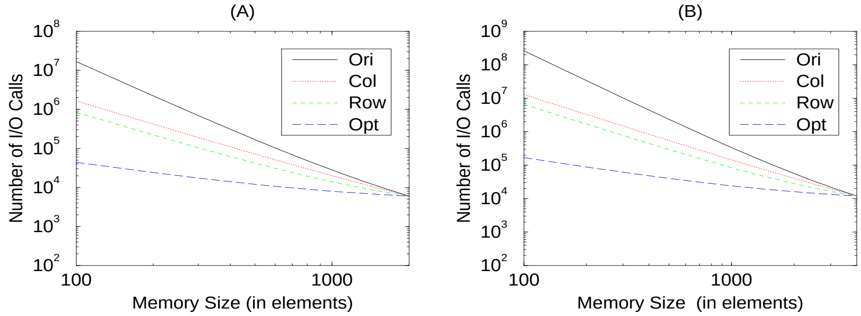

Figure 2: **Number of I/O calls for (A) $2K \times 2K$ and (B) $4K \times 4K$ double arrays.**

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

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

2.5 Number of I/O Calls

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

3 Conclusions

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

We observed that for the shared-disk systems I/O optimizations are particularly critical. For the optimized programs, the network latency and the contention on I/O nodes are the main bottlenecks. As a result, for shared-disk systems we advocate the use of fast I/O nodes. Since some I/O nodes may be over- or under-utilized, we also suggest the techniques that allow co-operation between I/O nodes in servicing disk requests. In fact, it has been shown that those techniques led to much better performance than the traditional parallel file systems [3]. Once the hardware/software techniques to eliminate the contention on the I/O nodes are employed, we believe, the approaches based on I/O optimizations are promising.

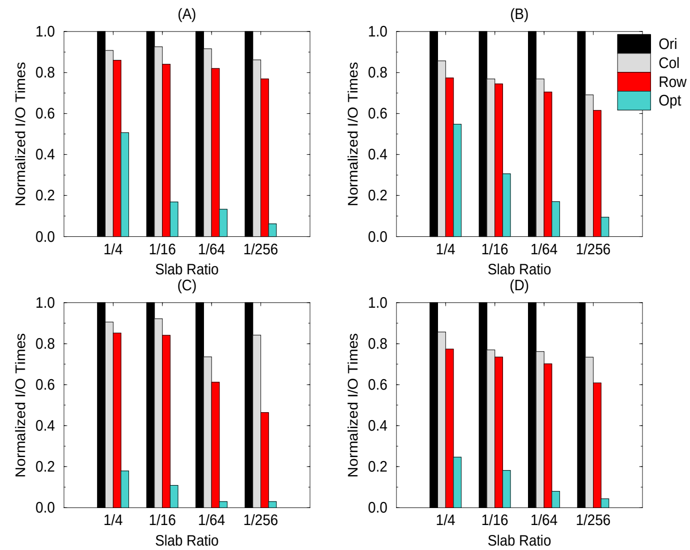

Figure 3: **Normalized I/O times for the second example. (A) $2K \times 2K$ arrays on IBM SP-2 (B) $4K \times 4K$ arrays on IBM SP-2 (C) $2K \times 2K$ arrays on Intel Paragon (D) $4K \times 4K$ arrays on Intel Paragon.**

References

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

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

[3] 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.

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