I/O Optimizations for Compiling Out-of-Core Programs on Distributed Memory Machines

M. Kandemir\textsuperscript{1} R. Bordawekar\textsuperscript{2} A. Choudhary\textsuperscript{3}

Extended Abstract

Since many of the large scale computational problems usually deal with large quantities of data, optimizing the performance of I/O subsystems of massively parallel machines is an important challenge for system designers [1, 2]. We describe data access reorganization strategies for efficient compilation of out-of-core data parallel programs on distributed memory machines. In a data parallel programming environment distribution of data to each processor results in each processor having a local array associated with it. For large data sets local arrays cannot fit in local memory. We refer such local arrays as out-of-core local arrays. The portions of local arrays, called tiles, currently required for computation are fetched from disk into memory. It should be emphasized that the node memory should be divided among the tiles of different out-of-core local arrays. The compilation for an out-of-core data parallel program consists of two phases. In the first phase, called in-core phase, the arrays in the source program are partitioned according to the distribution information and bounds for local arrays are computed. Array expressions are then analyzed for detecting communication. The second phase, called out-of-core phase, involves adding appropriate statements to perform I/O and communication. The local arrays are tiled according to the node memory available and then the resulting tiles are analyzed for communication. Finally the loops are modified to insert necessary I/O calls. In a straightforward compilation where no I/O optimization is performed, the compiler divides the memory equally among competing arrays and then fetches tiles with sizes of equal length in each dimension, works on them and stores them back in files if necessary. We believe that this straightforward translation can be improved substantially by being more caggy about file layouts, occurring of tiling loops and memory allocations for different arrays. Our approach consists of three steps: 1) Determination of the most appropriate file layouts for all arrays referenced in the loop nest, 2) Permutation of the tiling loops in the nest to maximize spatial and temporal locality, and 3) Partitioning the available memory among references based on I/O cost estimation. In order to achieve these, we define an index cost function $ICost(IT, R, r, layout)$ of tiling loop index $IT$ with respect to an array reference $R$, an index position $r$ of $R$ and a file layout which may be row-major or column-major. Simply $ICost$ is the cost of reading a data tile of a specific shape from file into memory. Using $ICost$ values, basic I/O cost of a tiling loop index $IT$ is defined as follows.

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

First we iterate over loop indices in the nest to get array cost values as

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

Notice that the $ACost$ gives the cost of an out-of-core array under the corresponding layout. Next our algorithm considers all possible layout combinations and chooses the one with the minimum I/O cost. After selecting the layouts for all arrays, our algorithm reorders the tiling loops by computing the

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

for all $IT$ where $R$ is the array reference and $layout_R$ is the layout of the associated file as determined in the previous step. The tiling loops are permuted from outermost to innermost according to non-increasing values of $TCost$, if the desired permutation observes existing dependences; otherwise the order is not changed.

\textsuperscript{1}CIS Dept., Syracuse University, Syracuse, NY 13244, e-mail: mtk@cse.syr.edu
\textsuperscript{2}CACR, Caltech, Pasadena, CA 91125, e-mail: ramesh@cacr.caltech.edu
\textsuperscript{3}ECE Dept., Northwestern University, Evanston, IL 60208-3118, e-mail: choudhary@ece.nwu.edu
Table 1: I/O times for unoptimized and optimized versions of a four-deep nest on IBM SP-2.

The last part of the algorithm is an intelligent memory allocation scheme. It starts by allocating tiles with sizes of equal length in each dimension for all arrays (as in the straightforward case), and then employing an iterative approach it finds the optimum allocation under specific memory and processor constraints. To see the performance improvement obtained by our approach consider a four-deep DO nest, with loop indices i, j, k and l, that contains the statement $A(i, j) = A(i, j) + B(k, i) + C(i, k) + 1$. Assume arrays A, B and C are 4096 x 4096 real out-of-core arrays and each loop iterates from 1 to 4096 with unit stride. Table 1 shows the I/O times (in seconds) for such a nest with different number of processors on IBM SP-2. It can be seen that the I/O optimizations can reduce the time spent in I/O as much as an order of magnitude. in-core ratio is the ratio of available memory to the total size of out-of-core arrays.

We conducted experiments on both IBM SP-2 and Intel Paragon using common out-of-core kernels such as matrix-multiplication, four point relaxation, iterative solver etc., and concluded the following: 1) The programs optimized by our approach improves the I/O cost significantly due to optimized file readings and writings. 2) The optimized programs also scale better than their unoptimized counterparts since more processors can work I/O optimally at the same time. 3) When the number of processors is increased, the effectiveness of our approach increases. 4) Optimized programs perform better as the amount of node memory is reduced since the straightforward translations issue many more I/O requests. 5) Demonstrations on two different platforms with varying compile-time and run-time parameters, such as number of processors, available memory, array sizes etc., prove that our algorithm seems to be quite robust. We also optimized the same nests assuming fixed file layouts for all arrays and concluded that the three steps of our algorithm can be applied separately or in any combination considering the I/O and computational requirements of the application in hand. To further evaluate the effectiveness of our approach at reducing the number of I/O calls, we simulated the number of I/O calls for different programs with different communication requirements and derived some formulae which prove the effectiveness of our approach analytically.

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

References
