Scalable I/O for Out-of-Core Structures

Ken Kennedy
Charles Koelbel
Mike Paleczny

CRPC-TR93357-S
November 1993

Center for Research on Parallel Computing
Rice University
P.O. Box 1892
Houston, TX 77251-1892

Scalable I/O for Out-of-Core Structures * (Extended Abstract) †

Ken Kennedy ken@cs.rice.edu
Charles Koelbel chk@cs.rice.edu
Mike Paleczny ‡ mpal@cs.rice.edu

Center for Research on Parallel Computation
Box 1892, Rice University
Houston, TX 77251-1892

Abstract

Programmer managed I/O for out-of-core data requires significant programming time and specializes the application to one target machine. This work demonstrates that it is possible for automatic techniques to extend an algorithm for work on very large data sets. Starting with the basic algorithm for LU factorization, we hand compiled the program to produce an out-of-core LU factorization with explicit I/O operations. This process required interprocedural symbolic analysis of scalars and array sections, interprocedural `MOD` and `REF` information, and dependence analysis. We believe that these techniques may be generalized to allow the extension of in-core algorithms for out-of-core data. This will simplify the expression of out-of-core algorithms and improve portability by moving machine specific optimizations into the compiler.

1 Introduction

One of the greatest attractions of modern parallel machines is the availability of large main memories. However, for the largest problems even these memories may not be enough. In these situations, the only option is to process the data in stages, moving data from secondary to primary storage as necessary. Such algorithms are commonly called *out-of-core* algorithms, as opposed to *in-core* methods which keep all their data in main memory. Out-of-core methods can be programmed either explicitly (using disk operations) or implicitly (using virtual memory). We consider new methods for handling implicit out-of-core methods in the compiler, thus making out-of-core methods more usable by programmers working on large problems.

---

*Submitted to SHPCC94

†This research was supported by The Center for Research and Parallel Computation (CRPC) at Rice University, under NFS Cooperative Agreement Number CCR-9120008.

‡Corresponding author: 713-527-8101, ext. 2738; Fax 713-285-5136

Due to the lack of virtual memory on some parallel machines and the desire for peak performance, most out-of-core applications have been written explicitly. One example is the LU factorization program developed at Sandia National Laboratories [6], which we have used in this experiment. The main advantage of this approach is that it allows fine control of the implementation, including such optimizations as overlapping I/O with computation. However, converting an in-core algorithm for out-of-core use requires a significant amount of work, obscures the central algorithm, and often specializes the program to one architecture.

Use of virtual memory can provide the user with the illusion of additional memory [3], thus avoiding some of the pitfalls described above. Unfortunately, it is unavailable on many older parallel machines. Even when it is available, optimal use of virtual memory is not easy to achieve. For sequential machines, Abu-Sufah [1] recognized the importance of code reorganization to improve data locality in virtual memory and suggested doing this in the compiler. Our work extends this suggestion by having the compiler convert virtual memory operations into explicit disk operations, which may then be further optimized. We also consider parallel machines, which Abu-Sufah did not.

Our thesis is that the compiler can effectively insert the I/O operations to convert an in-core algorithm to an out-of-core algorithm. By "effectively" we mean that the compiler-generated code will perform competitively with hand-written explicit out-of-core programs.[1] Such automatic generation would allow computations on very large data sets to be programmed at a higher level than is possible today.

As a preliminary test of our thesis, we have hand-compiled a parallel LU factorization program, converting it from in-core form to explicit out-of-core form. The remainder of this paper describes that experiment. Section 2 describes the factorization program. Section 3 describes the process of inserting I/O into this program, and Section 4 describes the reintroduction of parallelism into the program. Section 5 describes the result of the experiment, while Section 7 describes our conclusions and future plans.

2 Application

To be certain that the core algorithm was suitable for out-of-core extension, we started with an implementation of a parallel out-of-core LU factorization for the nCUBE [6]. This program was simplified until it could be run sequentially on a `SUN` workstation. The simplification included replacing explicit task-parallel constructs by parallel loops and removing explicit I/O statements (including buffering operations). This removed both the explicit parallelism and the algorithmic structure for segmenting data to fit into memory.

Although the automatically generated program is similar to the original version there are several differences. The most important difference is in the handling of the `permute()` routine implementing partial pivoting. In the original program it is executed immediately after `factor()`, changing the format of stored data necessary for updating the right submatrices. The generated code executes `permute()` after the last time a segment is used to update data to its right. This preserves stability of the data, and does not introduce additional I/O statements (see figure 1).

[1] It is worth pointing out that our "compiler" is actually a source-to-source translation system, not a traditional compiler to object code.

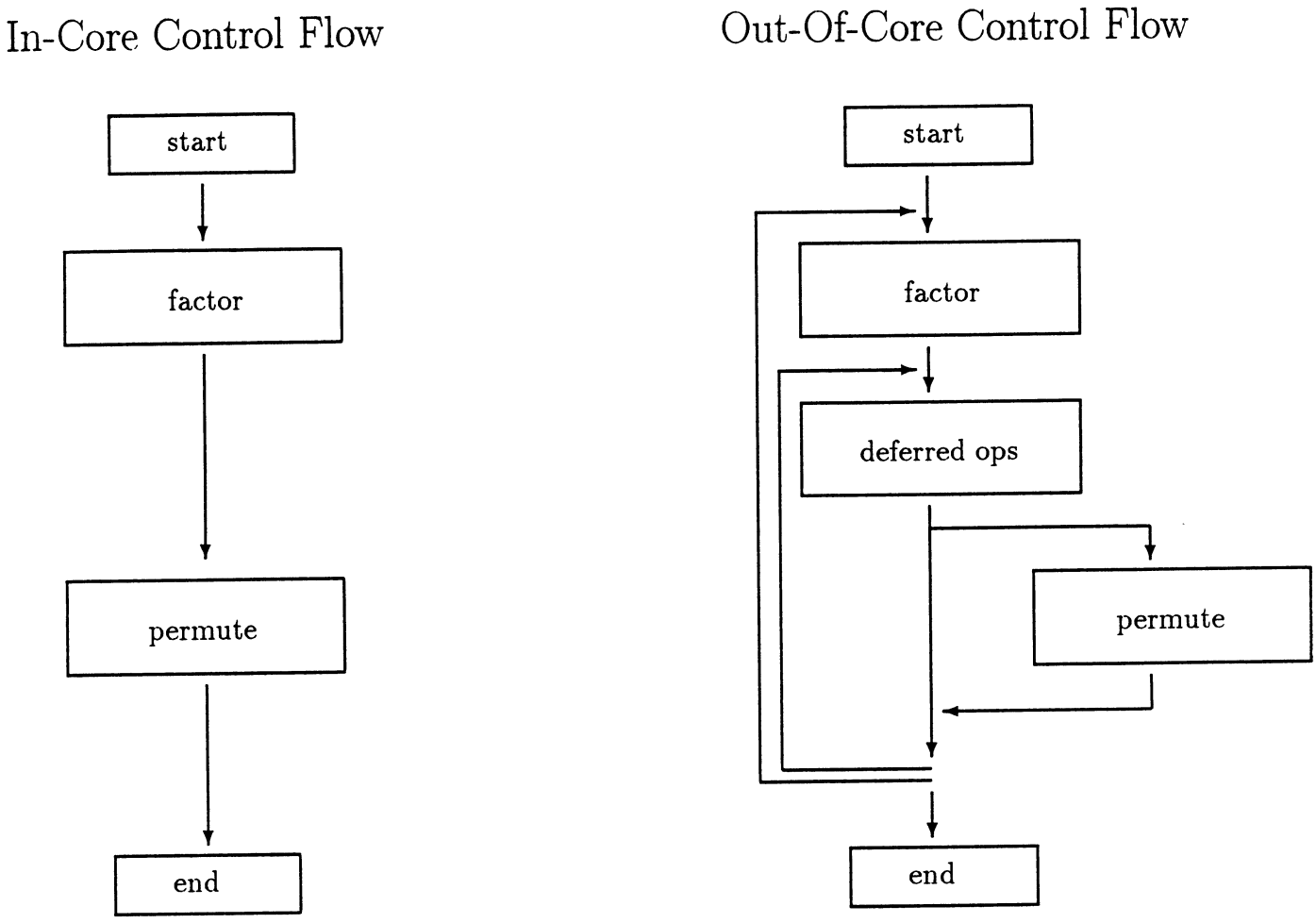

Figure 1 Reorganization of the program

3 Converting In-Core to Out-Of-Core

Starting with the sequential in-core algorithm we proceeded with a standard compilation pattern of analysis followed by transformation.

3.1 Analysis

There are two principal goals of the analysis. First, identify the relationship between the control structures and the amount of data accessed. Second, provide sufficient information to allow a "safe" segmentation of the computation.

The first problem is clarified by the following observation. In order to access amounts of data larger than main memory it is necessary to use looping constructs. For this reason we parameterize the `REF` and `MOD` summaries by the induction variable(s) used in the array reference. This enables us to "scale" the amount of data accessed by restricting the number of loop iterations. To summarize this information we use a format similar to bounded regular sections. Although more precise representations of data access patterns are known, it is not efficient to access arbitrarily complex patterns on disk. In this case bounded regular sections

were sufficient.

The second problem is more difficult since there is no apriori guidance as to how much information will be required. We have assumed an interprocedural symbolic analysis system as powerful as that described by Paul Havlak [4]. In addition to interprocedural constant propagation, this includes interprocedural value numbering of both scalars and expressions in a gated single assignment representation. The ability to do relative comparisons of symbolic quantities is also provided, and was necessary to determine the correctness of one transformation. Results of analysis were provided by hand for library routines.

3.2 Segmenting the computation

In general, the outermost loop of an application will access the entire data structure. This is the case in our example, where the main computational portion of the program consists of an outer loop indexing columns of the matrix, and several nested loops which work either on the column, or rows and submatrices to the right. Since it is not feasible to keep all the data in memory at one time, someone must choose the data to be kept in memory, and implicitly, which operations will be deferred. This choice is made by the user based upon the extent of data being modified, the extent and stability of data necessary for the computation, and the predictability of the preceeding. In our example, the pivot row is not predictable making a decomposition into blocks of rows undesirable for I/O. The other choice, using an I/O decomposition which blocks the columns, is desirable for a different reason. The data necessary for operations on the right submatrices is "stable". It is not modified between the time when it would be updated if all the data where in memory, and the time when it will be needed for the deferred computation.

Given the I/O decomposition by the user, into blocks of columns, the structure of the algorithm requires that we start by factoring the initial segment. After this our choice is between applying the resultant Gauss transforms to each segment to the right, or updating the adjacent segment and then factoring it thus finishing two segments. The first follows the original coding of the algorithm, and is the execution sequence which would be followed if using a virtual memory system. The second defers updates to most segments which are not in memory so it can finish processing the next segment while it is resident. Although the number of segments which must be read for either approach is similar, defering the updates is preferable since it reduces the number of segments which are written. Updating aggressively results in $O(N^2)$ segment writes, while defering updates until all segments to the left are finished results in $O(N)$ segment writes. This difference is visible to the compiler since all looping constructs have fixed bounds.

Having decided the execution order to support out-of-core data and which operations to defer, the compiler builds the necessary control structure. The original core algorithm requires only slight modification to work on a segment instead of the entire data set. In addition, regions of this algorithm are transformed to provide the defered operations.

3.3 Introducing I/O

While the compiler builds the control structures, it also inserts placeholders for the I/O statements which will bring the required data into memory, and write out the results of a modified segment. Each placeholder includes information describing which out-of-core data is needed for the computation.

3.4 Overlapping I/O and computation

At this point the compiler has information readily available concerning when data will be brought into memory, when it will be accessed or modified, and if it will be written back to disk. This is also the information necessary to overlap the segment I/O with computation. Overlapping requires the use of one or more buffer segments and further modifications to the program's control structure.

4 Introducing Parallelism

In addition to choosing a distribution of the data for I/O purposes, the user must choose one for parallelism. This decision reflects a program-wide estimate of which communication and access patterns should be done via I/O or interprocessor communication. This will be implemented within the segment of data that is currently in memory. Providing both the I/O and parallelism distribution statements allows the user better management of the memory hierarchy. Message passing between processors is much faster than storage and retrieval from disk.

Parallelism within a segment will be exploited using the techniques developed for Fortran D [5]. In general, this should consist of parallelizing the central algorithm before it was expanded to handle the multiple segments of out-of-core data. In this case study, the principle operations which need to be parallelized include: a BLAS DMAX() and DSCAL() operation, a row exchange operation, and the update of memory resident columns to the right.

Although the parallelization proceeds almost independently from the segmentation of the application, the I/O statements for handling each segment need to be specialized for each processor. This parallelization of I/O is done based on the "owner" of the data, and the type of file-system available.

5 Results

A performance comparison of the original and compiler reconstructed program , using an Intel Paragon, will be presented in the full paper. The comparison between the two programs will focus on the ability to automatically generate comparable quality I/O. Additional work on I/O optimization is in progress.

6 Related Work

Previous approaches to handling very large data sets include virtual memory and hand-coded I/O. The use of virtual memory is restricted to those machines on which it is implemented; hand-coded I/O requires significant programmer effort, and encourages architecture specific optimizations in the algorithm.

We have not found compiler support for out-of-core computation in the literature. Some pertinent work has been done for out-of-core permutation and sorting [2], but these techniques were not incorporated into a compiler.

7 Conclusion

The introduction of I/O statements by the compiler moves concerns about memory size away from the programmer. This improves clarity and portability of the algorithm while encouraging compiler optimizations of I/O. The example application illustrates the applicability of these techniques to regular computations. However, successful application to irregular problems will require software solutions for interprocessor coherence of data.

For machines with virtual memory, it is still desirable for the compiler to identify overlap opportunities. Working in cooperation with the operating system, the compiler can identify "safe" regions for prefetching. In addition, it is desirable for the compiler to help identify when interprocessor data coherence will or will not be a problem.

It remains to be seen how efficient compiler-generated I/O will be in practice. We argue that the information collected in the process of generating the I/O is sufficient to perform most optimizations now done by hand. Further testing is ncecessary to verify this, and to quantify the role of machine parameters in I/O generation. In short, while we are encouraged by the results of this experiment, it will still be some time before users can leave out-of-core computations in the compiler's hands.

References

[1] W. Abu-Sufah. *Improving the Performance of Virtual Memory Computers.* PhD thesis, 1979.

[2] T. H. Cormen. *Virtual memory for data-parallel computing.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1992.

[3] Peter J. Denning. Virtual memory. *ACM Computing Surveys*, 2(3):153–189, September 1970.

[4] Paul Havlak. *Interprocedural Symbolic Analysis.* PhD thesis, Department of Computer Science, Rice University, 1993.

[5] S. Hiranandani, K. Kennedy, and C. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[6] David Womble, David Greenberg, Stephen Wheat, and Rolf Riessen. Beyond core: Making parallel computer i/o practical. In *DAGS: Issues and Obstacles in the Practical Implementation of Parallel Algorithms and the Use of Parallel Machines*, Hanover, NH, June 1993.