Scalable I/O for Out-of-Core Structures

Ken Kennedy
Charles Koelbel
Mike Paleczny

CRPC-TR93357-S
November, 1993

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

Updated August, 1994.
This research was supported in part by the CRPC.

Scalable I/O for Out-of-Core Structures *

Ken Kennedy
ken@cs.rice.edu

Charles Koelbel
chk@cs.rice.edu

Mike Paleczny †
mpal@cs.rice.edu

Rice University
CITI/CRPC - MS 41
6100 South Main Street
Houston, Texas 77005-1892

August 4, 1994

*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

Contents

Abstract

Many computational methods are currently limited by the size of physical memory, the latency of disk storage, and the difficulty of writing an efficient out-of-core version of the application. Although hardware and software support for virtual memory allows the in-core version to be run on larger datasets, the performance is often poor. This is expected because the programming model in most high-level languages ignores the size of memory (as well as the size of the data cache and number of registers.) We propose to develop practical compiler algorithms that transform an in-core application to efficiently process out-of-core data. This will include transformations to insert the necessary I/O, improve locality, and overlap I/O with computation. In addition, we will determine the analysis necessary to support the program's transformation. The application of these transformations will require a minimum of additional information from the programmer and will assume a compiler to file–system interface which supports prefetching.

1 Introduction

Current trends in hardware performance indicate a widening performance gap between the central processor and levels of the memory hierarchy. One level of this hierarchy is the movement of data from disks to memory. Out-of-core programs, whose data set exceeds the size of main memory, are strongly effected by performance at this level. We believe that a compiler can automatically transform an appropriate in-core algorithm to operate on out-of-core data. The following four items provide motivation for this approach:

1. Out-of-core applications include important practical problems.

   According to David Scott [17], Intel's out-of-core linear algebra system generated the sale of at least five systems worth two million dollars each.

2. It is difficult for the programmer to manage the memory–disk interface explicitly.

   These difficulties arise from correctly staging data through the algorithm and efficiently using the target machine. Programmer-inserted I/O also reduces portability.

3. There are opportunities for improved I/O using program transformations.

   Abu-Sufah [1] showed that improving locality can benefit program performance when using virtual memory. Prefetching also has been shown to provide performance improvements in some cases, when done by the operating system [13] or suggested by the application level [16]. Currently, prefetching and staging transformations are often done by hand to improve performance on large data sets.

4. Useful information is available to the compiler.

   The determination of interprocessor communication for both regular and irregular problems by the Fortran D compiler supports the feasibility of identifying I/O patterns at compile time. Identifying access patterns at run time on parallel machines can be very costly and may introduce unnecessary demands on and conflicts for system resources [13].

The thesis of this research is that a compiler can transform an appropriate in-core algorithm to operate on out-of-core data through I/O insertion and *data set splitting*. Data set splitting refers to the staging of data through the steps of the computation. We illustrate the programming difficulties and our proposed solution with a sequence of examples.

Figure 1 contains a simple I/O–bound program which identifies the maximum value contained in a file. When working on a machine without virtual memory and with less physical memory than SIZE_A, this program cannot be executed without modification. When executed on a virtual memory machine with less physical memory than SIZE_A, some parts of the array will be paged out before computation begins. Depending on the paging algorithm used by the operating system, this could be the data which is needed first. The operating system cannot determine this based on past history; it requires "predicting the future" which can be done in some cases by the compiler.

In this example, these problems can be fixed easily by the programmer. However, tiling the computation and prefetching the data reduces clarity and portability as shown in Figure 2, which uses asynchronous I/O calls for the Paragon. In addition, it is no longer simple when there is reuse of data, multiple data structures, additional loops, complex algorithms, or a parallel architecture.

1.1 Approach

Our solution to the above problems is to develop compiler techniques to choreograph I/O for an application. A desired organization of I/O will be declared with statements similar to Fortran D's ALIGN, DECOMPOSITION, and DISTRIBUTION. These annotations allow the programmer to describe at a high–level the relationship between a data structure and its use in the computation. The compiler will use this information and static program analysis to segment the computation, then construct and insert appropriate I/O statements. Since only part of the data set is in memory at one time, computations which would require nonresident data are deferred until the data is resident. These groupings of computations will be called "deferred routines." The identification and insertion of I/O will allow the compiler to perform additional optimizations. The one we will focus on is overlapping I/O and computation.

Guided by the annotations in Figure 3, the compiler can introduce the I/O statements necessary to obtain the needed data. This will preserve for the programmer the model of unlimited memory close to the processor. The tile size may be suggested by the programmer, but the final selection will be done by the compiler to mimimize the amount of I/O, to balance the I/O time with computation

```
      PARAMETER (MAX_SIZE_A=10 000 000)
      INTEGER A(MAX_SIZE_A)
      INTEGER SIZE_A

      READ SIZE_A
      READ ( A(I), I=1,SIZE_A )

      MAX_A = MAX( A(1), A(2) )
      DO 100 I = 3, SIZE_A
        IF( A(I) .GT. MAX ) MAX_A = A(I)
100   CONTINUE

      WRITE MAX_A
```

Figure 1: Simple I/O bound program.

```
      PARAMETER (MAX_SIZE_A=10 000 000)
      INTEGER A(TILE_SIZE, 2)
      INTEGER SIZE_A

      INTEGER BUF1, BUF2, BUFTEMP
      INTEGER IO_ID(2)

C     Elide file open bookkeeping

      READ SIZE_A

      MAX_A = MAX( A(1), A(2) )

C     Tile data space
      BUF1  = 2
      BUF2  = 1
      DO 100 J = 3, SIZE_A, TILE_SIZE
        BUFTEMP = BUF1
        BUF1    = BUF2
        BUF2    = BUFTEMP

C       Fetch data for first block
        IF( J .EQ. 3 )
          IO_ID(BUF1) = IREAD( NUNIT, A(1,BUF1), TILE_SIZE )

        IOWAIT( IO_ID(BUF1) )

        IF( J .LT. SIZE_A - TILE_SIZE )
          IO_ID(BUF2) = IREAD( NUNIT, A(1,BUF2), TILE_SIZE )

        DO 100 I = 1, TILE_SIZE
          IF( A(I,BUF1) .GT. MAX ) MAX_A = A(I,BUF1 )
 100    CONTINUE


      WRITE MAX_A
```

Figure 2: Hand-tiled I/O bound program with prefetching.

```
      PARAMETER (MAX_SIZE_A=10 000 000)
      INTEGER A(MAX_SIZE_A)
      INTEGER SIZE_A
!*    A IS OUT-OF-CORE UNFORMATTED FILE "/TMP/DATA"
!*    I/O-DECOMPOSITION A(SIZE_A)
!*    I/O-DISTRIBUTION  A(BLOCK)


      MAX_A = MAX( A(1), A(2) )
      DO 100 I = 3, SIZE_A
        IF( A(I) .GT. MAX ) MAX_A = A(I)
 100  CONTINUE

      WRITE MAX_A
```

Figure 3: Annotated I/O bound program.

for overlap, and to fit the tiles within physical memory. The I/O inserted by the compiler would read the initial part of the file and start the associated computation before accessing the next part of the file. This removes the problems mentioned for both virtual memory and non-virtual memory systems.

Unlike compilation, which produces code that executes directly on hardware, I/O generally involves the operating system and disk subsystem. At this time we consider some of the interactions between these hardware and software systems and the programmer and compiler.

1.2 Processor and Disk Parallelism

Although many techniques for managing out-of-core structures will be useful for both single processor and multi-processor machines, we expect that they will be more beneficial for MIMD systems. Run-time analysis of access patterns is more expensive in a multi-processor system [13], so compiler analysis that coordinates multi-processor I/O or selects an appropriate library routine may significantly reduce run-time overhead. These techniques should be particularly effective on programs which are readily expressed in data–parallel form. The separability of data will allow both parallelism across processors and splitting the data on each individual processor.

When using parallel disk subsystems the most important consideration is not which disk the data is on relative to the processor which requests it, but what other data shares that disk. The cost of a network transfer is less than the latency of disk access, while the distribution of data to disks affects both the ability of a program to access data in parallel and to amortize access latency by reading larger blocks of data from the same disk. Although we will assume an independent disk for each processor to simplify the development of cost models, our implementation will be able to generate code which does not require a separate disk at each processor.

1.3 Utilizing the Operating System

Although the operating system and file system for uniprocessors can provide access to some out-of-core structures (virtual memory), there are users who require better performance. These users write their own I/O into the program and optimize for each architecture. Improvements to the operating system are possible, but the potential for optimization of I/O before run time needs to be explored. This may result in the automatic generation of "hints" as discussed in [16] or the complete management of I/O by the compiler and file system as is done for databases.

To avoid requiring programmers to manage the I/O for their application and support architecture independence, the compiler, file system, and run-time system must work together to provide better performance.

1.4 Hardware Support

Improving performance through compiler overlap of I/O with computation depends upon hardware support for non-blocking I/O. Uniprocessor machines provide this with DMA hardware, while parallel architectures may provide dedicated message passing or I/O processors. Each of these allows access to the file system to proceed in parallel with execution in the CPU.

2 Example: LU Factorization

To test the feasibility of using a data–parallel specification to guide compiler construction of I/O statements, we used LU factorization as a case study. The original program [21] was provided by David Womble at Sandia National Laboratory. A summary is provided here; details are provided in Appendix A. The works done at Sandia shows that low–level I/O optimization is important, but requires significant programmer effort. Our experience suggests that much of this low–level work can be done by the compiler.

The original program as obtained from Sandia was hand–coded to run in parallel on the nCUBE using out–of–core data. We simplified this version to a sequential in–core program to which the proposed compiler techniques could be applied.

Given an I/O–distribution and I/O–decomposition specifying a block distribution of the data, we used Fortran D compilation techniques to identify data accesses and then constructed deferred routines. Deferred routines are composed of operations which require or produce significant amounts of data which are not resident in memory. Execution of these routines is "deferred" until the required data is available.

Constructing the correct I/O statements for a subroutine containing deferred operations requires a summary of the reference information for the routine. In general, data elements which are not available in memory but used in the routine need to be read from disk. Results which are live on exit from the deferred routine usually need to be written to disk. However, in some instances these results can be kept in memory as an optimization.

The sequential out–of–core version we derived was run on SPARC workstations and single nodes of an Intel Paragon. A later data–parallel version was developed with the aid of the Fortran D system at Rice University and has been tested on the Paragon. The preliminary results from sequential execution are shown in Tables 1 and 2. Each table indicates the total time taken by the unmodified computation using virtual memory and compares this to the best time for a blocked computation using virtual memory, explicit file I/O, or overlapped I/O and computation. The "best" time is selected from the results for different block sizes. The order of computation and locality characteristics of all versions are the same, varying only by the blocking parameter

Matrix Size	Total Time (sec) Original Order Virtual Memory	Segment Size	Total Time (sec) Reordered		
			Virtual Memory	Explicit I/O	Overlapped I/O
100x100	0.53	best	0.53	0.55	0.57
800x800	432	best	337	311	290
1200x1200	71049	best	1049	1112	990

Table 1: Results from SPARC 1+ (8 MB Memory)

(segment size) that determines how much data is kept in–core to be used by a deferred routine and the granularity of access to data which is needed for deferred operations, but are not within the deferred segment.

Matrix Size	Total Time (sec) Original Order Virtual Memory	Segment Size	Total Time (sec) Reordered		
			Virtual Memory	Explicit I/O	Overlapped I/O
100x100	0.08	best	0.09	0.11	0.14
800x800	33.3	best	33.3	21.4	20.5
1200x1200	2270	best	178	120	98.4
1600x1600	82587	best	399	268	234

Table 2: Results from Intel Paragon (1 node, 16 MB Memory)

The results from a SPARC1+ indicate that unoptimized explicit I/O introduces approximately 6% overhead for either a small or large problem size. On the Intel Paragon however, there is a 33% speedup using explicit I/O for the out–of–core problem size when compared to using virtual memory. In both cases, the reordering of computation based on the user's declarations provided very significant improvements compared to the original ordering of computation. This is a locality enhancement at the memory–disk level of the memory hierarchy. Improvements at the cache–memory level have not yet been performed, but could be applied to the output code.

We conclude that the performance of unoptimized explicit I/O is, at worst, comparable to that of virtual memory for our test case. On systems where virtual memory is not very efficient, explicit I/O can be faster even before optimization. The ability to automatically generate an explicit out–of–core version of a program can be beneficial even without additional optimization.

Figures 4 and 5 graph the total execution time of three versions of our test code using an out-of-core data set while running on a SPARC 1+ and a Paragon. Each version does worse at block sizes larger than those displayed. On the SPARC, virtual memory performs better than file I/O, but overlapping file I/O with computation is the best. Explicit I/O on the Paragon, for its best segment size, achieves 33% better performance than any segment size used with virtual memory. The best result for overlapped I/O is an additional 12% faster.

Both non–overlapped and overlapped I/O have a pair of lines plotted in Figure 5 which compares the effects of using a different block size to access operands than used for accessing the deferred data. This allows the program to fit a larger amount of deferred data into memory, which results in a reduced amount of I/O. The effects of this approach when doing synchronous I/O are illustrated

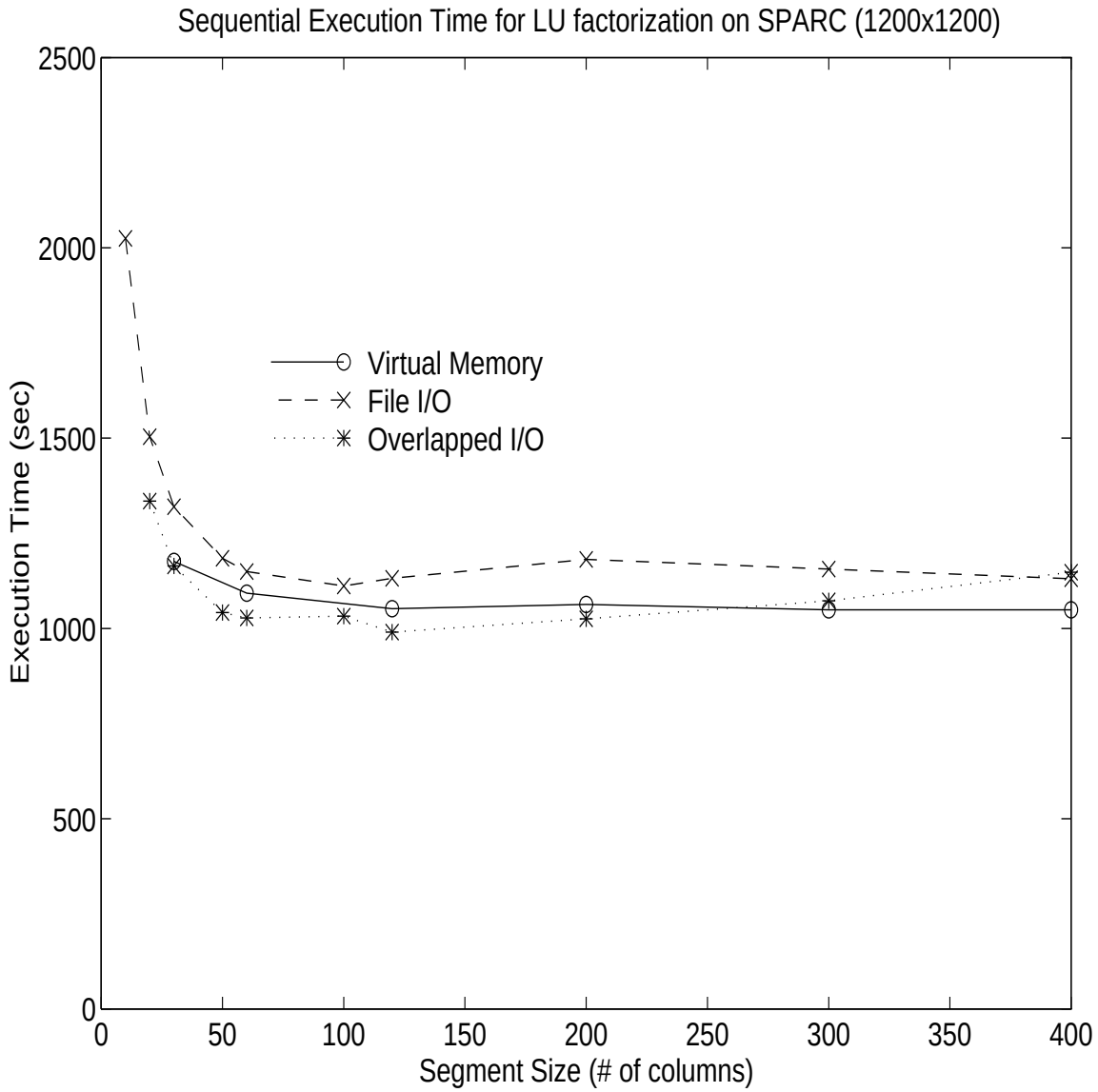

Figure 4: Results from SPARC 1+

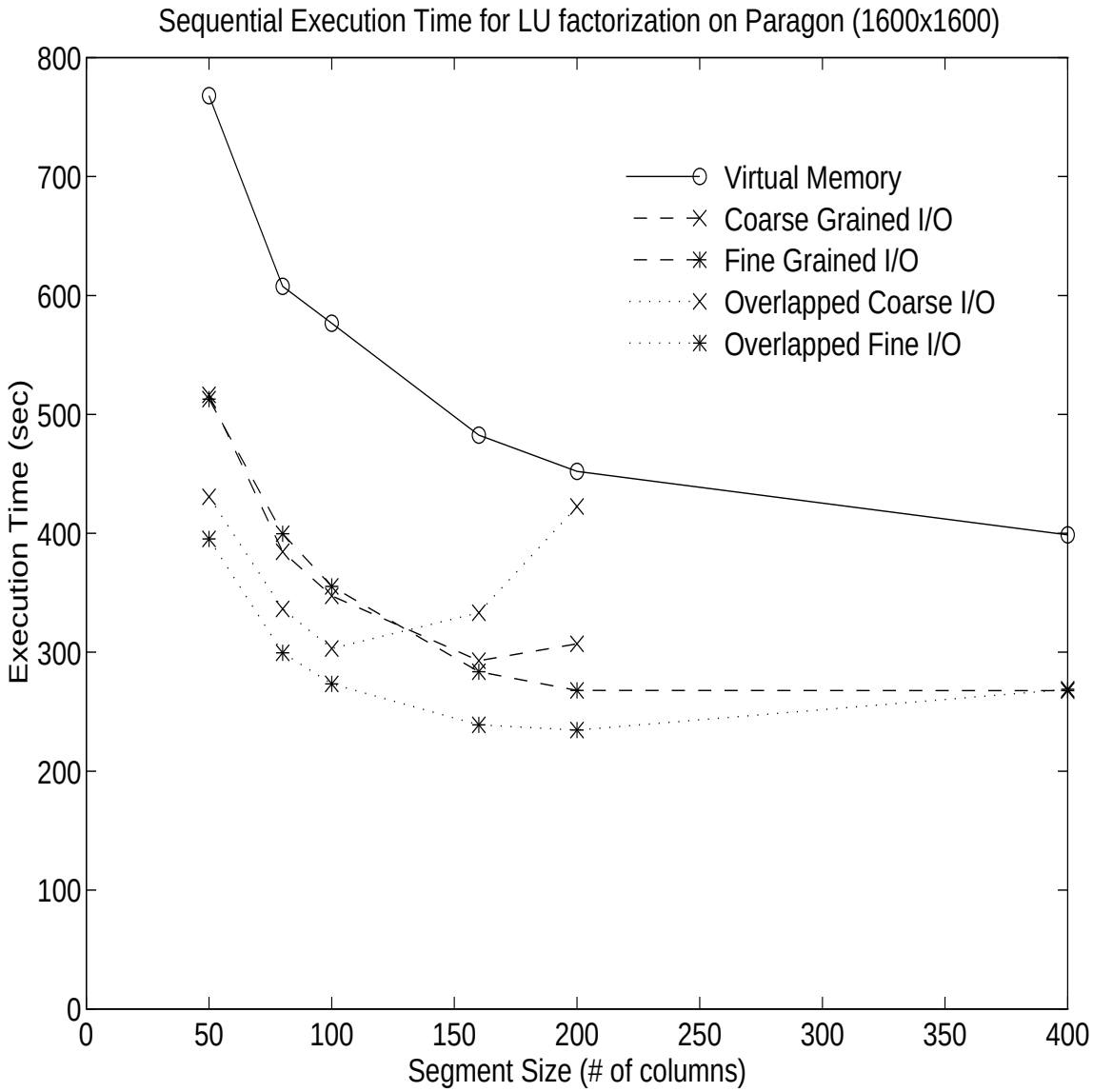

Figure 5: Results from Intel Paragon

by the two dashed lines. The dashed line using "×" for each data point shows that using a smaller segment for the operands external to the deferred segment allows a larger deferred segment in memory. This does not change the amount of computation required, but does reduce the amount of I/O.

Using memory for buffering incoming data when using asynchronous I/O reduces the memory available for the deferred segment. As described above, this increases the total amount of I/O. Using a smaller block size to access operands outside the deferred segment, as was done for synchronous I/O, reduces the memory required for the prefetch buffer. This is essential since overlapping I/O and computation provided, in our tests, a 20% reduction in I/O cost. For the overlap optimization to be beneficial, its savings must be larger than the increase in I/O volume required by using additional memory for the buffer. The two dotted lines in Figure 5 compare using a prefetch buffer the same size as the deferred segment "*"s with using a small prefetch buffer "×"s. The results indicate that a small asynchronous buffer provides a large enough reduction in I/O time to offset the increase in I/O volume.

A graph of execution time for the combined out-of-core and parallel version on two processors is shown in Figure 6. The "V" shape on the left of the graph occurs when the I/O requests increased beyond the 64K-byte block size. Aside from this artifact, the results show that the combination of out-of-core I/O and data-parallel computation can be achieved by an extension of data-parallel compilation techniques. In addition, compiler-managed prefetching from disk remains profitable. Unfortunately, when executing the same code on four processors we did not observe benefits from overlapping I/O and computation although file I/O was still faster than using virtual memory. We believe this is the result of having only one I/O node on our system and are currently attempting to gain access to a system with multiple I/O nodes for verification.

The complexity involved in predicting the correct segment size and choosing the correct buffering methods for an individual application further support our belief that compiler management of I/O will be a productive research area.

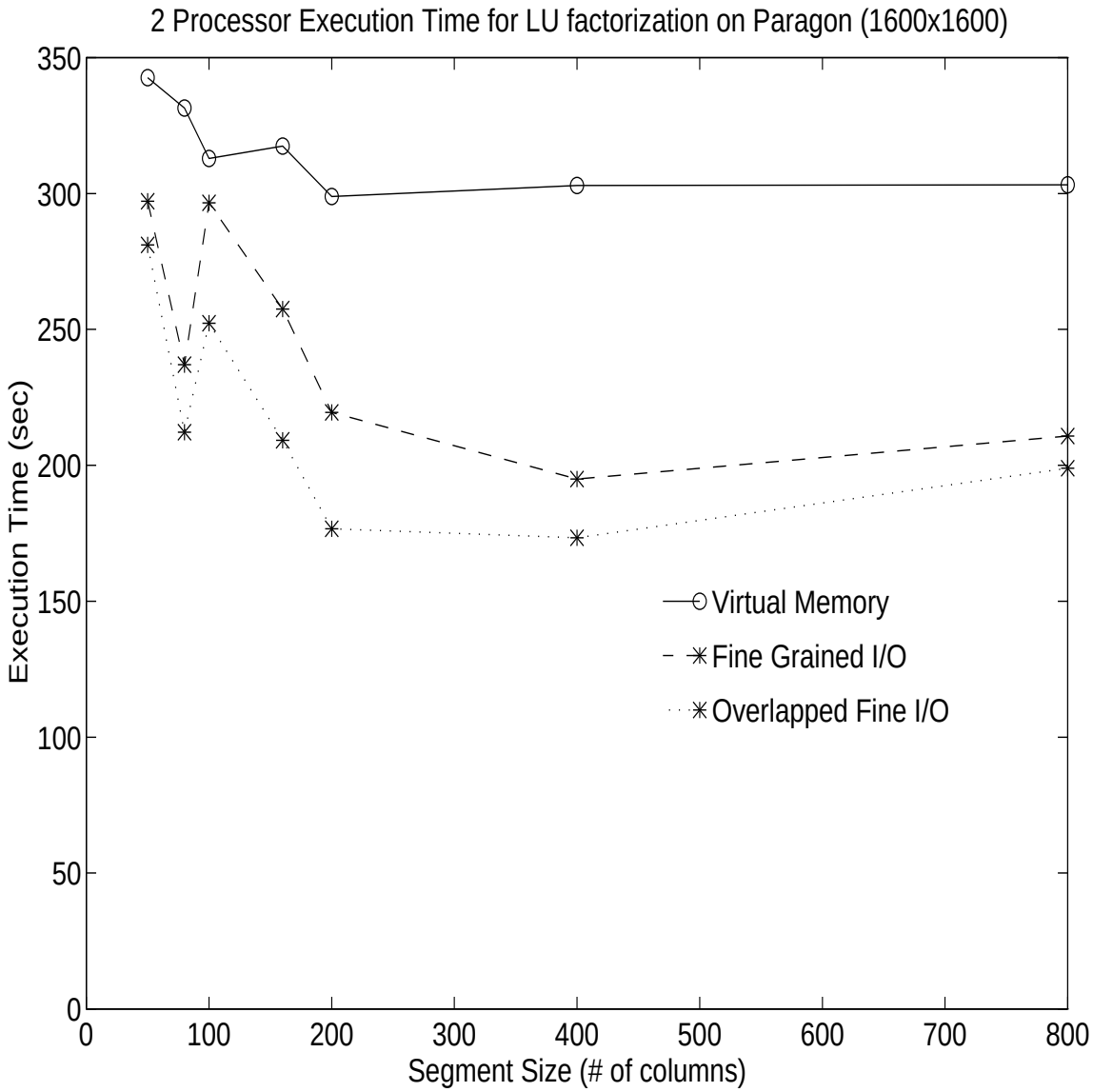

Figure 6: Results from Intel Paragon using 2 processors

3 Related Work

We have not found compiler techniques to automatically insert and optimize the I/O for out-of-core structures in the literature. There is, however, related work on out-of-core problems in algorithms, applications, and operating systems. Algorithm development for out-of-core problems on parallel disk systems has demonstrated the need for balanced use of resources; operating systems are continuing to reduce overhead and providing increased control over resources to the application.

Related research in compilation deals with several aspects of the memory hierarchy, including vector registers, cache prefetching, virtual memory, and balancing data access with computation. Last is an outline of data-parallel compilation for multiple instruction multiple data (MIMD) parallel processors using data-parallel annotations. This provides a starting point for our language features which allow a high-level description of the interaction between parallelism and the memory hierarchy.

3.1 Out-of-Core Applications

There has been a significant amount of research work on out-of-core versions of several classic problems. One of these is sorting, which has progressed from the use of magnetic tape [12] to parallel machines with parallel disk systems. The most important characteristic of these new parallel algorithms is "balance," distributing data evenly across both processors and disks [19]. Using the data decomposition and distribution directives and some knowledge of the target machine characteristics, we believe the compiler can maintain this balance for regular scientific computations.

In his dissertation [7] Tom Cormen lists many scientific applications which can benefit from efficient out-of-core memory management. These include computational biology, geophysics, ocean modeling, and visualization graphics. Our work with an implementation of LU factorization with pivoting done at Sandia National Laboratories [21] was discussed briefly in Section 2.

Another application area which requires support for large data sets is the work by Bill Symes on *Differential Semblance Optimization* [18]. This project provides a framework for user applications to make use of the DSO technique and supports user access to data in external structures via requests to named data fields. While the current compiler does not provide any optimization of I/O, some optimizations have been implemented by hand. Compiler optimization of this software package would require extensive interprocedural analysis to support I/O optimization.

3.2 Run-Time Systems

Cormen [7] developed a run-time system that simulates virtual memory and parallel disks as a back-end to the NESL system. [1] The simulator was used to measure the amount of I/O required by several different out-of-core algorithms for matrix permutation. Although run-time selection of the best out-of-core matrix permutation algorithm can significantly reduce the amount of I/O, Cormen concluded that source-level specification would be more desirable.

We agree with Cormen's assumptions that memory management should (1) focus on data instead of code, (2) exploit the predictability of data accesses, and (3) start at the source level. A compiler can accomplish these goals, identifying and optimizing I/O for static patterns of data use guided by the annotations mentioned in Section 1.1. We will test the feasibility of this approach by applying it to regular scientific applications. We will not attempt to do algorithm detection and specialization as was done by Cormen.

[1] The simulator runs under UNIX on Sun Sparcstations.

3.3 Operating Systems

Operating systems have traditionally provided support for efficient access to disk via direct disk block transfers. The addition of asynchronous I/O calls allowed programmers to overlap computation and I/O within the same application. This ability to improve the efficiency of an individual application is distinct from efforts to reduce the idle time of machine resources by multiprogramming. Current operating system designs are including more capabilities for applications to improve their own efficiency. Examples of this in sequential operating systems include the *madvise* system call in UNIX and the research done at Stanford on "external page–cache management" for the V++ system [8]. A good example in parallel file systems is the "logical partitioning" facility in IBM's VESTA [6].

Although file prefetching and caching is done profitably by sequential systems, new problems occur on multiprocessors. The most paradoxical is that improving the hit rate at the cache may not improve the application's runtime. Experiments conducted by Kotz and Ellis [13] indicate that even "successful" prefetching can have a negative impact on other processes and may increase overall execution time.

We believe that the compiler can make use of the facilities provided by the operating system to improve the efficiency of an individual application. On parallel machines, compiler placement of prefetching requests may reduce the risks of resource contention. Managing inter–program resource contention and allocation remains an operating system task, but even here the operating system might be aided by trustworty information concerning the resources needed by compiler generated code for efficient execution.

3.4 Compilers

Compilers strive to improve performance given particular hardware features and support language designs which simplify the programming task. The first item below is a seminal article concerned with the memory–disk level of the memory hierarchy. Abu-Sufah's results are encouraging, and they provide motivation for further compiler optimizations. The second, on vector register allocation, describes a method based on dependences which divides a large computation into several smaller operations capable of being performed efficiently by vector units. We will use data-parallel compilation techniques to perform a similar operation, dividing out-of-core computations into smaller sections. The third illustrates the importance of making "balanced" use of the machines resources. We will apply these ideas to both overlapping I/O with computation and performing I/O with parallel disks. The fourth is a compiler method for inserting prefetch instructions for the data cache. We intend to implement similar capabilities for the memory–disk level of the memory hierarchy using a high-level approach that we believe is more suitable for whole program transformation. The last is Fortran D, which supports high-level language annotations to ease the programming task in the presence of hardware parallelism. We will use similar annotations to suggest to the compiler an efficient structure for I/O in out-of-core scientific computations.

Virtual Memory. Abu-Sufah [1] concentrates on reference locality in scientific FORTRAN programs. His impression is that human generated programs do not in general exhibit good locality, and program transformations such as loop distribution and loop fusion can provide significant improvement.

His work does not address the central question of our thesis, the placement of I/O statements within the program. Additional goals of our work which are not addressed in his dissertation

include: improving performance by overlapping I/O with computation and allowing programs using out-of-core data sets to be run on machines without virtual memory.

Vector Register Allocation. Allen and Kennedy [2] focus on two main ideas for vector registers: *sectioning* vector operations which are too large for a vector register and program transformations to improve locality (allowing data to remain in a vector register). Sectioning of vector operations uses dependence analysis to detect when tiling of a vector operation violates copy-in/copy-out semantics. The transformations to improve locality are checked against the dependence relations between accesses in the vector computation to determine safety. During execution the vector registers are loaded from main memory.

The sectioning and allocation of registers differs from I/O management in the scale of data which is being sectioned and the number of "registers" which are available. However, the semantic problems introduced by sectioning are similar to those that will occur when "sectioning" an out-of-core computation.

Computation and Communication Balance. Callahan, Cocke, and Kennedy [4] pioneered balancing the amount of computation and communication done inside a loop nest with the capabilities of the target machine. Later, a specific transformation, scalar replacement, was successfully applied to scientific codes [3] to improve balance. This optimization allows the compiler to maintain array data in registers instead of cache, reducing latency for accesses to this data. We hope to use static performance estimation along with our out–of–core transformations to identify regions of code where the balance between I/O and computation can be improved.

Cache Management. Mowry [15] describes a method for software-controlled data prefetching which focuses on issues pertinent to the data cache. The algorithm contains three stages: identify reuse; isolate predicted misses; and schedule prefetches. Identification of reuse is done using a matrix representation of dependence equations, zeroing the dimension in which a cache line is fetched to identify potential accesses to the same cache line. Predicted cache misses are isolated using loop-splitting techniques to avoid using guards on prefetches inside an inner loop. Prefetch scheduling is done using software pipelining [2] to support overlapping prefetches with computation from a previous iteration.

The general ideas developed here apply in part to the memory–disk interface as well, but our specific solutions are different. We are using high-level data-parallel annotations to describe a cooperative division of data between disks and processors. In addition, this provides a description of data locality and aids in identification of reuse. We also use estimates of computation time when identifying overlap opportunities, but in addition to pipelining I/O and computation loops, we look for opportunities to overlap I/O with one or more loop bodies. Finally, while Mowry's approach has been successful for array accesses within inner loops, our high-level approach works with programs containing multiple loop nests.

Distributed Memory. Hiranandani, Kennedy and Tseng [11] described a compilation method for data–parallel scientific programs written in Fortran D. Programmer supplied directives for data alignment, decomposition, and distribution provide a guide for the compiler concerning the structure of communication and computation. The "alignment" statement describes the relationship between different data structures which will reduce communication. The DECOMPOSITION and DISTRIBUTION statements describe the computation's structure and guide the compiler in mapping the computation onto a physical machine. After introducing message passing communication and applying transformations to both the computation and control flow, the result is a SPMD

[2] Software pipelining separates initial and final iterations from the *steady state* body of a loop.

distributed memory program.

We will use the Fortran D annotations for describing parallelism and a similar set of I/O annotations for describing the structure of data which should be in memory during the out-of-core computation. The compilation techniques for multiprocessors will be extended to handle the identification of deferred operations as described in Section 2. Finally, we hope to use the performance estimator to balance resource usage during overlapped I/O and computation.

4 Research Plan

The following sections outline which problems will be solved and the initial approaches for their solution. If an approach turns out to be unsuitable, a method will be developed to achieve the necessary functionality.

4.1 Requirements

To achieve the goals described in Section 1, we intend to satisfy the following requirements in this project.

1. Compiler-managed I/O

   We will provide automatic insertion and optimization of I/O statements based upon brief programmer annotations. For a suitable in-core algorithm, programmer modifications will be limited to the insertion of annotations to guide the compiler's tiling of data for I/O.

2. Correct generated code

   We require that the original data dependence relationships be satisfied in the transformed program.

3. Efficient generated code

   We believe that compiler-managed I/O can be as efficient as virtual memory management and explicit hand-coded I/O statements for regular problems. Thus, our goal is to achieve comparable performance to these two approaches.

4. Apply to the entire program

   To apply I/O management to the entire program we must recognize data access points and relate them to control flow within the program both within and between procedures.

5. Balance I/O and computation on target machine

   Our goal is to match the I/O requirements for groups of deferred operations with the I/O capabilities of the target architecture. To accomplish this we can vary the number of deferred operations, their grouping, and the tiling of data.

6. Hide the latency of I/O

   To hide latency, the architecture and operating system must allow processing to continue while there are outstanding requests to the file system. This overlap of computation and I/O is done using asynchronous I/O on the Paragon.

7. Reasonable compile-time overhead

   This restricts the analysis and representation used by the compiler as well as the complexity of the program transformations.

8. Parameters

   The developed algorithms will be parameterized to make use of realistic hardware features such as independent I/O processors, parallel file-systems, and hardware virtual memory. This is necessary to provide architectural independence for the application code.

4.2 System Design

This section outlines parts of the compilation system which will identify, introduce, optimize, and parallelize I/O. Each component will be implemented within the Fortran D System at Rice University, enabling us to utilize the analysis tools and user interface for data–parallel compilation currently being developed.

1. Analysis: Discover patterns of data use within the program.
   (a) Propagate user-supplied I/O annotations.
   (b) Identify both the computation requirements and data-access requirements for each control structure, including initial input data and final output results.
   (c) Use Bounded Regular Section Descriptors (RSDs) [9] linked to the code similar to a Static Single Assignment (SSA) graph [10] to describe both data access and its relationship to control flow. Parameterize by induction variables where appropriate.
   (d) Examine value-flow through array sections to support optimization.
   (e) Identify regions of computation whose data-access requirements fit into available memory and regions whose data-access requirements do not.
2. I/O Insertion: Use analysis results to determine which section of data is needed and which should be stored to disk to reduce memory pressure.
   (a) Construct data–deferred routines using techniques similar to those of Fortran D.
   (b) Insert I/O for data–deferred routines considering the capabilities of I/O libraries and operating system characteristics.
   (c) Generate an explicit I/O program.
3. Optimization: Use the compiler's knowledge of the program to improve performance.
   (a) Overlap computation and I/O where profitable and safe.
   (b) Consider interaction with locality optimizations.
4. Parallelization: Use annotations to compile for parallel machines.
   (a) Parallelize the application.
   (b) Manage interaction of out-of-core I/O and multiprocessors.
   (c) Use multiple disks and parallel file systems.

4.2.1 Analysis and Representation

To support I/O insertion and optimization as mentioned in Section 4.2, we will require a representation for data accesses which relates them to the sequence of execution of loop iterations. To provide this information at a reasonable cost during compilation we will use a combined gated SSA and RSD representation of data access. The gated SSA style will provide us with the relationship to control flow, and the RSD representation will reduce the cost of analysis. Although the RSD representation cannot precisely represent arbitrary data access patterns, it is efficient at representing rectangular data access patterns. This analysis and representation approach will be based on the previous work done by Paul Havlak [10] and Vadim Maslov [14].

Information about data use will be collected in a manner similar to that done by the Fortran D compiler [11]. Access information will be collected first at individual references, and then summarized at control–flow nodes as a data–flow problem. It may be useful at this stage to allow a limited number of outlying data accesses to remain separate from the summary RSD (e.g., keep the first and last references in the list A(1), A(1000:1100), A(1100:1200), A(2000) separate). This can prevent excessively large I/O requests for unnecessary data. These RSDs will be parameterized by induction variables and indicate the direction of data reference. The latter will help indicate the potential for interleaving and overlapping I/O with computations.

4.2.2 I/O Insertion

The framework for inserting I/O statements is developed from data decomposition and distribution work in data-parallel languages. We assume that I/O–decomposition and I/O–distribution information is provided by the user. Using this information, deferred routines are constructed in a manner similar to that of processor-local code in the SPMD data-parallel programming paradigm. Most of the data necessary to execute these deferred routines is known from the I/O-distribution statement. Any additional data required for execution is identified by the data-access analysis mentioned previously in Section 4.2.1. This also indicates which results must be saved to disk for later use.

After completing the analysis of data access and the construction of the deferred routines, I/O statements are inserted to preserve data values. Initially we will insert synchronous I/O. Later, where there is potential for overlap with computation, we will insert asynchronous I/O initiate and complete statements.

4.2.3 Optimization

The principal optimization we will consider is overlapping I/O with computation. This starts by considering the summary information for computation surrounding the I/O statements and seeing how early the initiation of I/O can be pushed. Next, the compiler determines how late the completion of the I/O can be delayed. Up to this point the optimization can use the Give–N–Take framework described by von Hanxleden [20]. This framework needs to be extended to handle (1) constraints on the number of asynchronous I/O operations which may be active at once, (2) the potential conflicts in disk access, and (3) the size of memory which must store all the data until it is used. This is similar to an instruction-scheduling problem which must avoid exceeding the number of available registers. Additional optimizations to be investigated include locality improvement and pipelining I/O with computation.

Although we are primarily concerned with I/O for out–of–core computations, we also believe these optimizations will benefit UNIX utility codes. Managing I/O prefetching in the compiler can

avoid some of the difficulties encountered when prefetching is directed by the operating system [16].

4.2.4 Parallelization

Data-parallel style annotations [3] provide the basis for our description of both parallelism and out-of-core I/O structure. Providing separate annotations allows expression of the desired relationship between I/O and parallelism where the sequential order of the constructs can be used to resolve conflicts.

Data-parallel single-program multiple-data (SPMD) code will be generated by the Fortran D compiler. Parts of this compiler will also be used to construct the routines which operate on deferred tiles of data. When both of these operations are performed on the same program, the distribution of data will be done in the order of occurrence of the `DISTRIBUTION` and `I/O-DISTRIBUTION` statements. I/O on multiprocessors will make use of parallel I/O libraries when available, and will generally assume a separate disk is available for each node.

4.3 Validation

We will compare the results of our out-of-core transformation to the performance of an *equivalent* program using virtual memory support. An equivalent program will use the same computational ordering and locality optimizations but will read in all the data initially and rely on the system's virtual memory support for data access. To compensate for differences in the hardware and software on our test systems, we will use a calibration test. This will compare the performance of the system using virtual memory to the performance using explicit I/O on a simple program where the expected performance is similar. This comparison will be used to correct for system differences; e.g., the hardware configuration gives virtual memory an advantage over explicit I/O.

Optimization. The initial comparison of our out-of-core transformed program and the equivalent virtual memory program will be done before I/O optimizations. This will allow us to separate the effects of explicit I/O management from the improvements made possible by having explicit I/O available for optimization. It will also allow us to confirm that the initial I/O insertion is reasonably efficient. A final comparison of the optimized out-of-core program and the previous two programs will also be done to determine the benefits achieved by enabling I/O optimizations.

Criteria for Evaluation. We will consider the initial I/O insertion a success if the adjusted results for the out-of-core transformed program are within 10% of the results for the equivalent virtual memory program. This means that we expect explicit synchronous I/O and virtual memory to be similar given equivalent hardware, software, and locality. The introduction and optimization of I/O will be considered a success if it is at least 20% faster than the calibrated results using virtual memory. We hope to achieve this level of improvement on more than 50% of the test cases.

Version	Execution Time
Calibrated Virtual Memory	T
Explicit I/O	$T \pm 10\%$
Optimized explicit I/O	$\leq T - 20\%$

Test Codes. Our original desire to work with and compare our results to industrial out-of-core codes will not be possible as these programs are proprietary and so far we have not been successful

[3] We use data-parallel in this case even though the I/O-annotations do not require parallelism. A better term in this case may be data-organization.

in obtaining them. The ideal test code would be (1) written in FORTRAN, (2) data–parallel, and (3) one for which very large data sets are appropriate.

Our current proposed set of test codes does not match the ideal. The most suitable is LU Factorization with pivoting for which we have completed the I/O insertion and parallelization. Additional test codes which can be used for sequential testing include UNIX utility codes like *grep*. Although these are written in C, their size is such that hand analysis and transformation may be attempted. We hope to be able to work with the Differential Semblance Optimization package, but it is written in C and C++, which are not accepted by our current compiler environment. Other programs which are being considered include additional numerical and simulation codes. In some cases this will be the first out–of–core version constructed from these algorithms.

5 Contributions

This work will provide a framework for compiler management of I/O for out-of-core applications. The annotations, which will be similar to that used in Fortran D, will require minimal additional input from the programmer. The introduction of explicit I/O statements during the compilation process will allow the compiler both to optimize I/O and to consider the interaction of other optimizations on I/O.

References

[1] Walid Abu-Sufah. *Improving the Performance of Virtual Memory Computers.* PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1979.

[2] J. R. Allen and K. Kennedy. Vector register allocation. *IEEE Transactions on Computers*, 41(10):1290–1317, October 1992.

[3] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the SIGPLAN '90 Conference on Program Language Design and Implementation*, White Plains, NY, June 1990.

[4] D. Callahan, J. Cocke, and K. Kennedy. Estimating interlock and improving balance for pipelined machines. *Journal of Parallel and Distributed Computing*, 5(4):334–358, August 1988.

[5] Steve Carr and Ken Kennedy. Compiler blockability of numerical algorithms. In *Proceedings of Supercomputing '92*, Minneapolis, MN, November 1992.

[6] Peter F. Corbett, Dror G. Feitelson, Jean–Pierre Prost, and Sandra Johnson Baylor. Parallel access to files in the Vesta file system. In *Proceedings of Supercomputing '93*, pages 472–481, Portland, OR, November 1993.

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

[8] Kieran Harty and David R. Cheriton. Application–controlled physical memory using external page-cache management. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 187–197, Boston, MA, October 1992.

[9] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[10] Paul Havlak. *unpublished.* PhD thesis, Rice University, unpublished 1994.

[11] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[12] D. E. Knuth. *The Art of Computer Programming. Volume 3: Sorting and Searching.* Addison–Wesley, Reading, MA, second printing edition, 1975.

[13] David Kotz and Carla Schlatter Ellis. Prefetching in file systems for MIMD multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):218–230, April 1990.

[14] Vadim Maslov. Lazy array data–flow dependence analysis. In *Proceedings of the Twenty-First Annual ACM Symposium on the Principles of Programming Languages*, pages 311–325, Portland, OR, January 1994.

[15] Todd C. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching.* PhD thesis, Department of Electrical Engineering, Stanford University, March 1994.

[16] R. Hugo Patterson, Garth A. Gibson, and M. Satyanarayanan. A status report on research in transparent informed prefetching. *Operating Systems Review*, 27(2):21–34, April 1993.

[17] David S. Scott. Parallel I/O for dense matrix factorizations. Invited speaker. DAGS Symposium, June 1993.

[18] William W. Symes. DSO user's manual. Technical report, Rice University, in progress 1993.

[19] Jeffrey Scott Vitter and Mark H. Nodine. Load balancing paradigms for optimal use of parallel disks and parallel memory hierarchies. In *DAGS93*, pages 26–39, June 1993.

[20] Reinhard von Hanxleden and Ken Kennedy. Give–N–Take — a balanced code placement framework. Technical Report CRPC–TR94388-S, Rice University, March 1994.

[21] David Womble, David Greenberg, Stephen Wheat, and Rolf Riessen. Beyond core: Making parallel computer I/O practical. In *DAGS93*, Hanover, NH, June 1993.

A Data–Deferred Operations

We applied the methods described in Section 1.1 to the LU Factorization program developed at Sandia National Labs [21]. In the first section of the outline below we converted the out-of-core parallel nCUBE program to a sequential version with the same algorithm. This was done to ensure that the starting program was suitable for both parallelization and extension to out-of-core data. We do not believe very large data sets are appropriate for every program and wanted to start our investigation with a test case for which an out-of-core version is desirable.

Below is an outline of the steps taken during the case study. The first group describes the simplification of the original algorithm. The second part of the outline describes the steps followed to construct the out-of-core version. This corresponds to the proposed system design in Section 4.2. The following sections describe the different stages of this process.

1. Simplify Parallel Out–of–Core Algorithm:

done	Sequentialize
done	Remove out–of–core control flow and code
done	"Clean up" sequential version

2. Extend Sequential Algorithm:

done	Specify I/O template
done	Analyze program
done	Construct control flow
done	Construct out–of–core routines
in progress	Test
in progress	Parallelize and test
in progress	Optimize and test

A.1 Preparation

A.1.1 Sequentialize

The program developed at Sandia was written for the nCUBE running the Vertex operating system. The process of sequentializing the program proceeded by first isolating the calls necessary for interprocessor communication and synchronization. These calls were then either removed or replaced by sequential equivalents to keep temporary data structures consistent with their former use. When there was not a performance reason for these structures in the sequential version, they were removed during cleanup.

A.1.2 Remove Out–of–Core Control Flow and Code

The original out–of–core program contains several subroutines which are designed specifically to handle the extension of the "core" algorithm to out-of-core data. In addition there is control–flow to manage the execution of these routines as different segments of data are brought into memory. The first addition to be removed was the buffering to allow asynchronous I/O.

Converting this explicit out-of-core program into an in-core program required identifying the core set of operations which could be used as a complete algorithm. In this case this set of operations

can all be found in one routine. This is the first routine executed to start the algorithm and can be viewed as the full algorithm applied to the first available segment of data.

A.1.3 "Clean Up" Sequential Version

After obtaining a workable sequential algorithm, the program was simplified to remove unnecessary variables, data structures, etc.

A.2 Hand–Compiled Transformation

A.2.1 Specify I/O Template

The I/O template we chose to implement by hand is similar to that used in the hand-coded version. We are using a block distribution in one dimension instead of a cyclic distribution[4] because we felt it would be simpler to implement by hand. A rough guide for this was obtained using the current Rice Fortran D compiler to help identify data distribution (segmenting) and communication (I/O) requirements. This did not, however, provide a guide to the control–flow necessary to choreograph the I/O and computation. The I/O–decomposition and I/O–distribution used are very simple for the programmer to insert as stated in Requirement 4.1.

```
C      I/O-decomposition   matIO( 1600, 1600  )
C      I/O-distribution    matIO(  : , block(200) )
```

The block parameter indicates a suggested size for the block. Unlike a block distribution for parallelism, the number of I/O blocks cannot be determined directly from the number of processors or the number of available disks.

A.2.2 Analyze

The above template divides the data into segments as illustrated in Figure 7 and Figure 8. Using this segmentation of the data, we used the analysis from the prototype Fortran D compiler at Rice to determine which operations use or modify the data in each segment. This gives us the following structure for the pivot operation.

The structure for the scaling operation is similar; two buffers hold the previous results and the data currently being scaled.

A.2.3 Construct Control Flow

The analysis performed above has identified the relationship between loop iterations and data access. Using this information we can construct routines consisting of "deferred operations" in a manner similar to the "program partitioning" in Fortran D [11]. One difference from the Fortran D approach is that we may produce different routines for successive levels of deferred operations (i.e., we do not require adherence to the SPMD model for deferred routines). Later parallelization of the out–of–core program will use the SPMD model.

For LU Factorization, the control–flow we contructed to surround the deferred operations reflects the control–flow of the in–core version. There was a choice which needed to be made by the compiler to determine the ordering of segment execution. Here the question was whether to update each segment to the right of the current segment after completing changes to the current

[4] The cyclic distribution was chosen for load-balancing reasons, not for I/O purposes.

segment, or update a deferred segment with all data to its left when all data is available to complete computations in the deferred segment. We chose the second because this reduces the number of times that the deferred segment is updated while in memory.

A.2.4 Construct Out–of–Core Routines

For simplicity of expression, the deferred operations are encapsulated into subroutines. These are preceeded and followed by the necessary I/O to provide data and store results. Using analysis like that in the Fortran D compiler, we identified which loop iterations where associated with the data which would be in memory and constructed routines containing these operations.

A.2.5 Construct Out–of–Core I/O

The data needed to apply deferred operations to a deferred tile is a combination of the data contained in this tile and data provided from other tiles. In LU factorization, the data needed from outside a deferred tile comes from columns to the left of the tile. I/O statements to access this data are inserted before the deferred routine. This I/O statement is inside a loop which stages data into memory to allow completion of all deferred operations in the deferred tile.

A.2.6 Test

The resulting sequential program has been tested on the Intel Paragon and the SPARC workstations available at Rice.

A.2.7 Parallelize and Test

The sequential out–of–core program was parallelized using a Fortran D block distribution of rows.

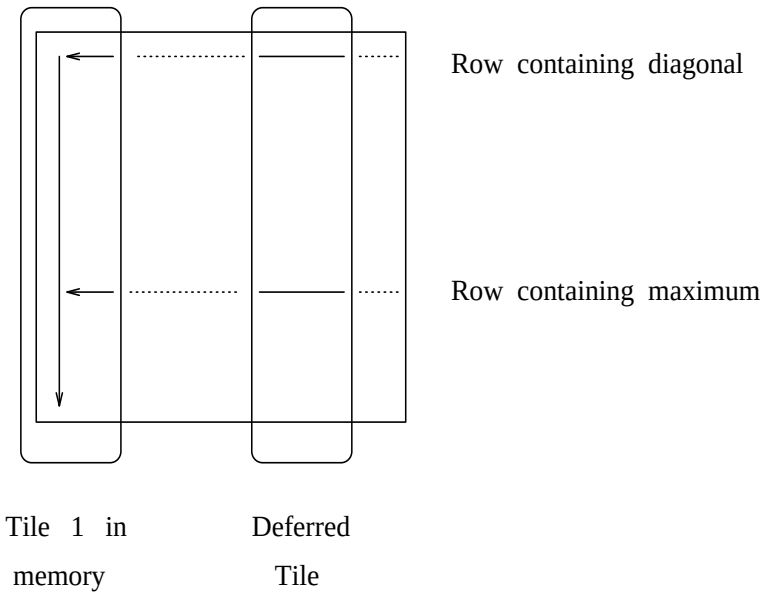

Figure 7: Deferred pivot operation.

```
      parameter       ( n$procs = 2 )
C     decomposition   matD( 1600, 1600  )
C     distribution    matD( block, : )
```

The combined I/O and parallel distribution of data is presented in Figure 9. The dotted rectangles surrounding blocks of rows illustrate the distribution of data to different processors. The large solid rectangle labeled *Deferred Tile* encloses a block of columns as suggested in the I/O-DISTRIBUTION statement. The two smaller groups of columns on the left represent the access pattern for data which is required to perform the deferred operations when the deferred tile is resident in main memory. This organization of out-of-core I/O and parallel computation has been tested on 2–node and 4–node configurations of an Intel Paragon.

A.2.8 Optimize and Test

The sequential and parallel version for the Paragon has been optimized by overlapping I/O and computation using the Paragon asynchronous I/O calls. A summary of the results was presented in Section 2. An additional 12% improvement was obtained using asynchronous I/O and fine-grained buffering.

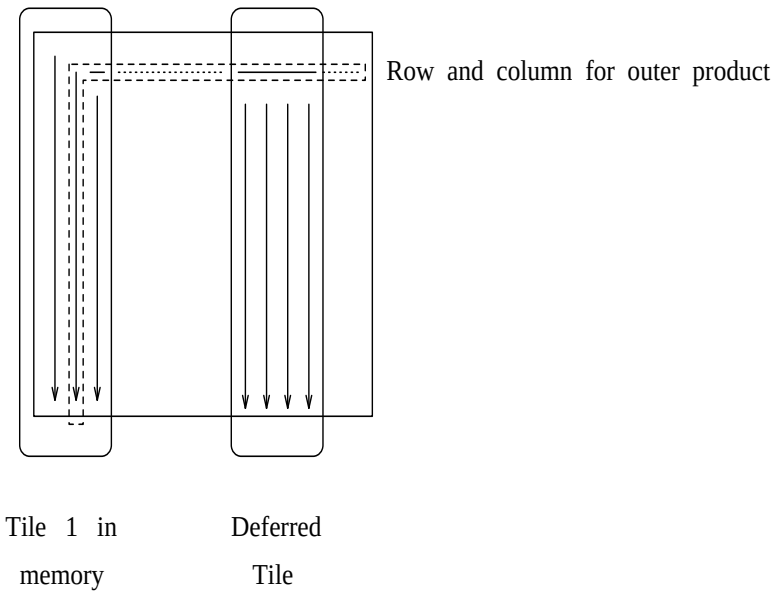

Figure 8: Deferred outer product operation.

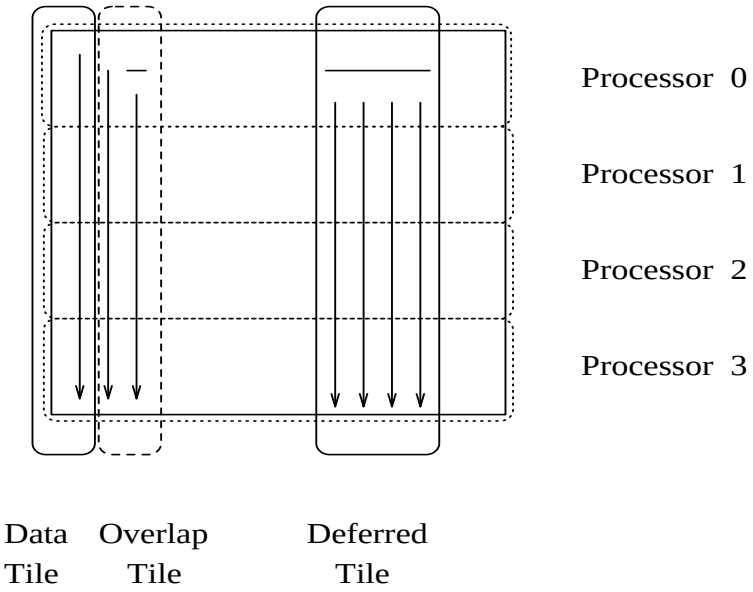

Figure 9: Interaction of Data and I/O DISTRIBUTION.