Compiler Support for Out-of-Core Arrays on Parallel Machines

Michael Paleczny
Ken Kennedy
Charles Koelbel

CRPC-TR94509-S
December, 1994

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

Compiler Support for Out-of-Core Arrays on Parallel Machines *

Michael Paleczny Ken Kennedy Charles Koelbel

Rice University, Department of Computer Science
Houston, TX 77005-1892

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.

We are investigating a compiler-based approach to the above problem. In general, our compiler techniques attempt to choreograph I/O for an application based on high-level programmer annotations similar to Fortran D's DECOMPOSITION, ALIGN, *and* DISTRIBUTE *statements. The central problem is to generate "deferred routines" which delay computations until all the data they require have been read into main memory. We present the results for two applications, LU factorization and red-black relaxation, on 1 to 32 nodes of an Intel Paragon after hand application of these compiler techniques.*

1 Introduction

Improvements in processor performance have outpaced developments in both memory and disk I/O speed. As a result, out-of-core applications, which require significantly more data than will fit into RAM, are likely to suffer from a bottleneck between memory and disk. Tiling the program's use of data is a common approach to improving data locality which must be applied by hand today. In addition, increasing the size of a data set typically increases the amount of computation. This makes parallel machines attractive for out-of-core problems. Unfortunately, parallel programming is also complex on current machines. Work at Sandia National Lab [13] on parallel out-of-core programs shows that low-level I/O optimization is also important, but requires significant programmer effort. Our experience suggests that much of this low-level work can be done by the compiler.

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

Our approach is to develop compiler techniques that choreograph I/O for an application, specifically, the temporary storage and retrieval of out-of-core data during execution. A programmer will declare the desired organization of input and output with statements similar to Fortran D's DECOMPOSITION, ALIGN, and DISTRIBUTE [7]. These annotations allow a high-level description of the relationship between an array and its use in the computation. The compiler will use this information and static program analysis to segment the computation, construct appropriate I/O statements, and insert them in the program. Since only part of the data set is in memory at one time, computations which require nonresident data are deferred until the data is resident. We call these groupings of computations "deferred routines." By making data accesses explicit, the compiler can also perform additional optimizations which include overlapping I/O with computation.

The similarity between the I/O annotations and those used to express data-parallel computation in Fortran D presents a consistent framework to the programmer. Providing separate annotations for out-of-core computation allows the programmer to consider the problem the problem of parallelism and I/O separately when desired. This paper describes a hand experiment using disjoint I/O and data-parallel annotations to guide the transformation to an out-of-core parallel program. This level of compiler support can reduce the programming effort spent on developing out-of-core applications and lead to the efficient execution of those programs.

The next two sections introduce our high-level I/O model and the language directives available to the programmer. Section 4 presents compiler methods to transform the program and choreograph I/O. This section also comments on some of the more interesting compiler issues that arose in our hand application of these methods. Section 5 describes aspects of the process relevant to each one of our applications separately. Section 6 presents the experimental results from run-

ning these programs on an Intel Paragon. Related work is discussed in Section 7 and our conclusions are presented in Section 8.

2 I/O system

Our model of the I/O subsystem pairs each processor with a disk as was done in [13]. A processor can access its local disk directly; remote disks require cooperation from their owning processor. We implement this on the Paragon by creating a separate file for each processor within a parallel file system. This file system is striped onto one RAID device and accessed through one I/O node. Additional details of this implementation and its effects on the results are presented in Section 6.

3 Programmer directives

The programmer provides source directives similar to the data distribution directives in Fortran Dto describe block-cyclic distributions of the data. The I/O and data-parallel directives used in our test cases are included in the untransformed source code for our sample applications at the beginning of Figures 5 and 6. The block size, specified by the programmer, refers to the size of an in-core data tile on one processor. When more processors are used, the compiler can keep additional tiles in-core until all the data is kept in memory. Input and output of these tiles is choreographed by the compiler. Initial and final data accesses can be remapped if necessary using either run-time routines [3] or compiler-generated I/O if the external distribution is known.

4 Compiling for out-of-core execution

Our strategy for out-of-core compilation consists of three phases: program analysis, I/O insertion and optimization, and parallelization and communication optimization. This design allows us to use components of the FortranD system at Rice University in our planned implementation of an out-of-core compiler. We apply the I/O and parallel distribution directives consecutively as the available Fortran D implementation only accepts one-dimensional distributions. This also simplifies the compilation of both an I/O and a data-parallel distribution in the same dimension. As we discuss in Section 5, we have not fully implemented this strategy but have simulated it by hand.

4.1 Program analysis

The program analysis phase uses traditional and new compiler techniques to discover patterns of data use within the program. Interprocedural data-flow analysis propagates the user I/O annotations in the same way as the Fortran D data mapping directives are handled. Code sections that access out-of-core data are identified as the sections that use the annotated arrays, and their data use is summarized using Regular Section Descriptors (RSDs) [6]. Again, this analysis is similar to the communications analysis performed by the Fortran D compiler. In addition, data and control dependences are needed to determine when data reuse and overlapping I/O and computation are legal.

4.2 I/O insertion and optimization

The I/O insertion phase uses the analysis results to partition the computation among tiles and to determine for each tile which section of data is needed and which should be stored to disk. The techniques used are similar to Fortran D's owner-computes rule, which assigns computation to the processor owning a particular datum. In the I/O arena, computation (often in the form of loop iterations) is split into code sections that process individual out-of-core tiles. These code sections are called "data-deferred routines" since their computation must often be deferred until data is read from disk. Generally, the compiler must insert I/O statements to read the appropriate out-of-core tile into memory and write modified data to disk when finished. USE analysis in the deferred routine determines if data may be required from other out-of-core tiles, in which case additional I/O statements and control-flow are inserted.

Next, the compiler inserts the control flow to process out-of-core tiles. In general, inter-tile dependences determine the ordering of tile computations. If the tiles are independent, any order can be used; initially, we use the execution order of the original program. We implemented this by adding a loop around the deferred-routines to iterate through the out-of-core tiles. More complex orderings, which might be more efficient, may require a more general methodology.

For example, the red and black computation routines in red-black relaxation exemplify the different effect of global operations on in-core data-parallel vs. out-of-core compilation. On a distributed memory machine, with all data in-core, it is reasonable to compute all the red points then compute the interspersed

black points. For an out-of-core problem, this approach requires two complete scans of the entire data set. When transforming red-black relaxation we align the loop which accesses out-of-core tiles for the black computation, with respect to that for the red, to allow the black computation for tile N to execute after the red computation for tile $N+1$. This requires that sufficient memory is available in-core for two tiles. If insufficient memory is available for a profitable transformation the compiler can provide feedback to the programmer. This transformation is similar to alignment of vector operations to allow reuse [2].

Next, the compiler can optimize the inserted I/O statements. Dependence analysis results can determine when it is safe to overlap computation and I/O. A cost model must also be applied to determine that this is profitable, as the overlapping may require more main memory. Another optimization involves detecting that data from one tile might be needed for the next tile. We believe that intersecting the summary RSDs is sufficient to produce this information.

When the initial or final data is not compatible with the temporary storage distribution, or the programmer requests a redistribution during execution, the data must be remapped. This can be done with run-time routines extended for out-of-core data [3] or by explicit I/O interleaved with computation when the source and destination distribution are known at compile time.

4.3 Parallelization and communication optimization

Finally, the parallelization phase compiles the transformed program for parallel machines. This uses the regular Fortran D compilation process, with extensions to allow access to the disk. In particular, we assume each processor has a private disk. When parallelizing the out-of-core I/O, each processor restricts the I/O statements to the values it "owns." Reflecting this in the implementation, each processor opens a separate file and does its reads and writes to that file. When a processor needs data it does not own, the owner performs the I/O and sends the data in a message.

One important interaction of this phase with the I/O insertion concerns the in-core overlap regions generated by the compiler. These may be interleaved in memory with the addresses available to hold an out-of-core tile. Although we use the simplest solution—each processor reads and writes the boundary data along with the original—a better solution would be to reorder the computation to isolate accesses to the border area and store the border in a separate buffer. This preserves the connectedness of the out-of-core tile and allows some overlap of the computation with I/O for the border.

5 Compilation examples

Each application is hand-compiled using the methods descibed in Section 4. The principal stages are summarized here: First, translate the I/O distribution information to Fortran D annotations and use the Rice Fortran D compiler to generate a tiled computation. These are converted by hand into deferred routines. Second, transform (by hand) the communication statements into I/O requests and hand-optimize the inter-tile I/O. Third, use the Fortran D compiler to generate a distributed memory node program. Finally, we manually modify the I/O operations to access only local data and resolve the interactions between data-parallel and out-of-core tiling. Before each use of the compiler, the code is simplified to satisfy restrictions in the implementation such as constant loop bounds, etc.; these changes are reversed after output. We use simple block distributions for both I/O and parallelism with the I/O distribution in the last dimension of the arrays.

5.1 Transforming red-black relaxation

Our first test case is red-black relaxation on a $320 \times 320 \times 320$ Cartesian mesh. This is representative

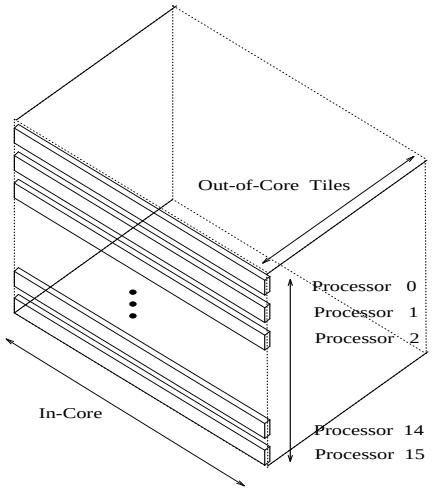

Figure 1: One out-of-core tile distributed to processors for red-black relaxation.

of many scientific algorithms on large 3-dimensional domains. The program is shown in Figure 5.

The out-of-core annotations used in this program are the I/O-DECOMPOSITION, I/O-ALIGN, and I/O-DISTRIBUTE directives. The intuitive meaning is that one tile (on one processor) is a $320 \times 320 \times 10$ block. When two processors are used, the tile in core at any time is 20 elements wide, and so on. This keeps memory filled on all processors. Orthogonal to the I/O distribution, the Fortran D directives specify blocking among parallel processors in the second dimension. The combined I/O and parallel distribution of data for one in-core plane of the array is illustrated in Figure 1.

In this application, each tile required extra communication only at the boundaries. (This is handled by small overlap regions, well-known from data-parallel compilers.) Inter-tile dependence edges show that the deferred black computation for tile N depends upon the red computation for tiles $N-1$, N, and $N+1$. Skewing the tiling loop for the black computation by one iteration, plus peeling the first iteration from the front of the red loop and the last iteration from the black loop, allows the two loops to be fused.

The boundary data also affects a read of the next tile, as the first plane of the next tile is already in-core. We identify this by intersecting the summary RSD, representing data in-core for computation on the current tile, with the RSD for data used by the next tile. This shows that the two planes of data adjacent to the next tile will be reused. Both are kept in memory and the I/O request for the next tile is reduced in size. After these optimizations are performed, no duplicate I/O occurs within an iteration of red-black relaxation.

We discovered several bugs in the Fortran D compiler apparently related to the loop stride of 2 (rather than 1), and modified the output by hand when it affected correctness. One significant bug was the compiler's inability to partition the loop bounds; instead, it inserted guards on all the assignment statements. As this produced correct but slow code, we did not rewrite it. This decision reduces the speedup of computation which affects the execution times we report later.

5.2 Transforming LU factorization

The second test case is LU factorization with pivoting, a method for solving dense linear systems. Our program is based on one provided by David Womble at Sandia National Laboratory [13, 14]. Since his program is out-of-core and parallel, we started by serializing the computation then used the compiler to rediscover the original implementation. The sequential code is shown in Figure 6.

In this case, the out-of-core annotations specify blocks of columns, while the parallel annotations specify blocks of rows, giving the combined I/O and parallel distribution of data shown in Figure 2. Again, the compilation follows the outline of Section 4. Since an in-core tile consists of a group of columns, a pivot operation spans all tiles to the right of the one containing the current diagonal element. The pivot operation is split into computation on the data currently in memory, and deferred pivots which apply to tiles to the right. The outer-product operation is likewise divided into current and deferred computations. The deferred pivot and outer-product operations execute in their original order when a tile is brought into memory. The deferred code is essentially a node program from which operations performing computations owned by other tiles are elided or deferred.

The inter-tile communication, for the outer-product, is more complex than in red-black relaxation requiring data from all previously computed tiles. Since not all these tiles can fit in memory at once, a loop must stage data into memory. Overlapping this I/O with the execution of deferred operations produces significant savings for LU factorization. Overlapping the I/O necessary for the outer loop does not, as it requires too much buffer space. Decreasing the size of deferred tiles by one half approximately

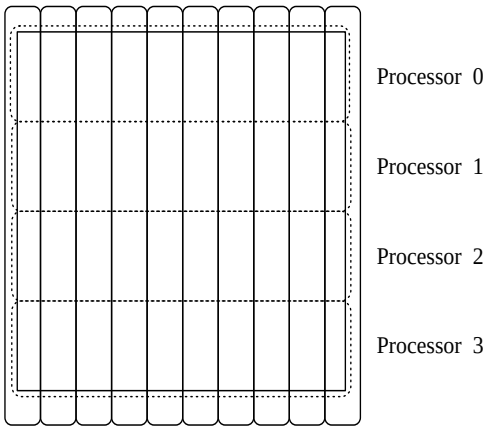

Figure 2: Interaction of data-parallel and I/O distribution for LU factorization

doubles the amount of I/O.

6 Results

Each of the applications was executed on the Intel Paragon located at Rice University under OSF/1 release 1.0.4 version 1.2.3. This release supports asynchronous file I/O. All of the file-I/O results were obtained performing I/O to one parallel file system (PFS) using two I/O nodes and two RAID disk systems, each connected using a SCSI-1 interface. The RAIDs are configured with 64K disk blocks and memory is divided into 8K pages. Results using virtual memory on more than 1 node paged to two RAID systems. The performance graphs relate total execution time to the number of processors and compare synchronous file I/O with asynchronous I/O. These results are also compared to the performance of the application using virtual memory (when that version completed execution). The distribution of initial data into files matches the distribution of the array. The timings include reading initial data from files and the final writing of results.

6.1 Red-black relaxation

The red-black relaxation code was run on nodes with 32 MB of main memory. Our results for virtual memory, synchronous I/O, and overlapped I/O with computation are shown in Figure 3. In addition, two different tiling strategies are compared. The solid line shows the virtual memory performance for 1 to 32 nodes. The performance improvement at 16 nodes occurs when all the data is retained in memory. The *large tile* version (shown as dashed lines) uses a tile size of 10 as explained in Section 5.1. Although this version kept all data in memory when using 16 processors, the large interprocessor messages crashed the application. This should be fixed within the parallelization pass of the compiler. The *small tile* version (shown as dotted lines) keeps exactly two tiles of the matrix in memory at any given time, each small tile holds two planes of data. This does not make full use of available memory, but is useful as a comparison to the large-tile version. The small size of the tiles avoided the large message problem described above.

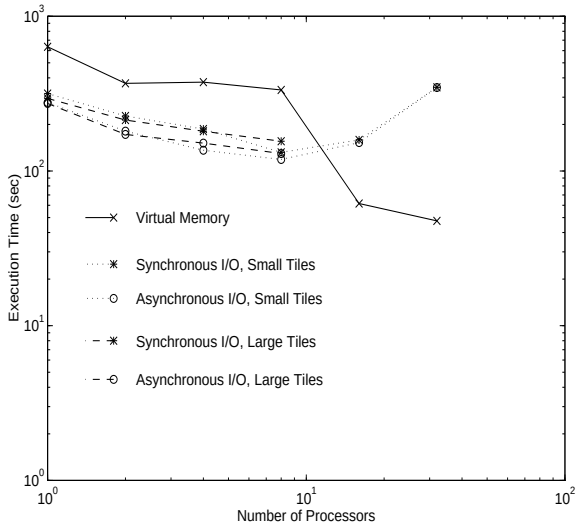

Figure 3: Execution time for red-black relaxation

Comparing the virtual memory performance to synchronous I/O shows that even at 8 processors, when more than half of the data fits in memory, our approach for compiler management of I/O is more than two times faster than virtual memory. We believe this result will still hold for larger out-of-core problems run on larger numbers of processors. Testing using different I/O request sizes indicates that much of this improvement is due to the larger requests made by the explicit I/O calls. Increasing the system's page size should significantly improve the performance of the virtual memory system on the large sequential accesses in our applications. However, this could adversely affect the performance of applications with different paging requirements.

As shown in Table 6.1, overlapping I/O and computation in red-black relaxation further improves performance an average of 17.4% for small tiles, and 14.9% for large tiles, on 1 to 8 processors. This behavior does not scale to larger numbers of processors for the small-tile version due to a reduction in computation at each node and I/O contention. A modified program that performs only I/O shows similar increases

Tile Size	Number of Processors			
	1	2	4	8
Small Tiles	12.3%	20.3%	27.0%	10.1%
Large Tiles	7.4%	19.4%	16.2%	16.7%

Table 1: Percent improvement in execution time between synchronous and asynchronous I/O for red-black relaxation.

in execution time. This is consistent with Intel's recommendation [8]: "In general, the recommendation is to have one I/O node for every ten compute nodes."

6.2 LU factorization

Our results for LU factorization are shown in Figure 4. One set of results is from nodes containing 16 MB of memory, the other from nodes with 32 MB of memory. The memory available for program text and data on each node is approximately 5 MB and 21 MB respectively. The graph compares the performance using synchronous file I/O operations ($\circ$ data points) with the performance using asynchronous file I/O ($*$ data points) for a 6400×6400 matrix using either 16 MB nodes or 32 MB nodes. Table 2 shows a consistent improvement obtained by overlapping I/O with computation while varying the number of processors.

The virtual memory version of this program did not complete execution at this problem size. The results of previous tests with a smaller data set are summarized in Table 3. The "Initial Program" "Virtual Memory" time used a version of the program with very poor locality. We transformed that program so that operations were performed in the same order as for our hand-compiled tests. This tiling gives a factor of 200 speedup from improved locality at the memory to disk interface. Further improvements occur from using file I/O instead of virtual memory, and from overlapping I/O and computation.

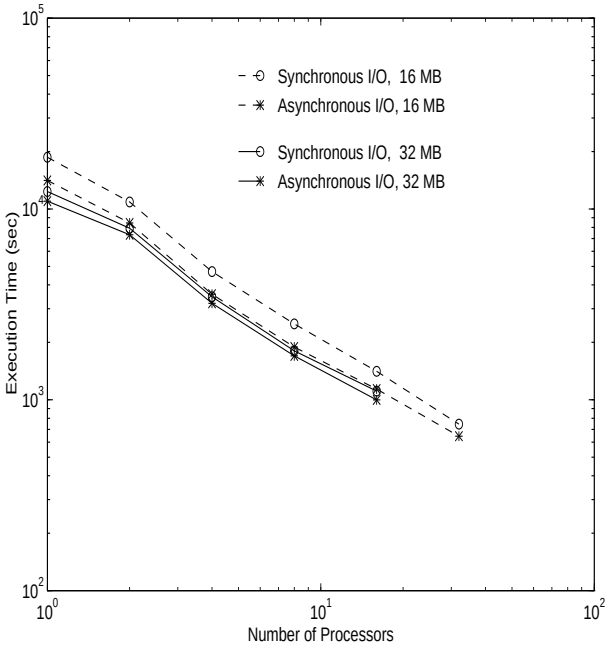

Figure 4: Execution times for LU factorization

6.3 Discussion

Synchronous file I/O performs better than virtual memory on our system, and much of this improvement derives from the compiler's requests for large blocks of data and the large block-size of the RAID I/O devices. Overlapping I/O with computation provides further improvements which are possible because the I/O is visible to the compiler. In this sense, the original insertion is an enabling transformation, allowing the compiler to generate code which utilizes the parallelism both within the I/O system and between it and computation.

One traditional advantage of virtual memory systems is the automatic use of additional memory. We took advantage of the additional memory available on multiple processors by scaling the size of in-core tiles. This works very well for LU factorization and should allow our large-tile version of red-black relaxation to avoid I/O when all tiles fit into memory.

One additional benefit is the separation of data into distinct files by the compiler. This makes it simple to distribute data used by different processors onto the available I/O devices.

An important question is whether these applications will exhibit similar performance characteristics when scaled to larger data sets, more processors, and additional I/O nodes. In red-black relaxation, the amount of both computation and I/O scale linearly with the number of data points. Executing a program ten times larger on a system which has been scaled by a factor of ten, processors and I/O nodes, will produce similar results. This is not the case for LU factorization. Computation and I/O increase faster than the size of the data set. Although scaling problem size, processors, and I/O nodes by a factor of ten will result in significantly longer execution times than for our example, the improvements from overlapping I/O and computation should remain similar as they depend upon the ratio of problem size to available memory. This ratio determines the amount of I/O which will be performed, but does not affect the amount of computation.

7 Related work

Hiranandani, Kennedy and Tseng [7] described a compilation method for Fortran D programs based

Memory Size	Number of Processors					
	1	2	4	8	16	32
16 MB	24.4%	22.4%	23.8%	24.4%	19.3%	13.4%
32 MB	10.9%	7.5%	7.4%	5.8%	10.0%	*

Table 2: Percent improvement in execution time between synchronous and asynchronous I/O for LU factorization. (* result unavailable)

Matrix Size	Total Execution Time (sec)			
	Initial Program	Tiled Program, after I/O distribution		
	Virtual Memory	Virtual Memory	Synchronous I/O	Overlapped I/O
100x100	0.08	0.09	0.11	0.14
800x800	33.3	33.3	21.4	20.5
1200x1200	2270	178	120	98.4
1600x1600	82587	399	268	234

Table 3: Comparison of explicit I/O to virtual memory for small problem sizes on 1 processor with 16 MB memory

upon programmer-supplied directives for data alignment and distribution. We are extending this method to support the automatic construction of out-of-core parallel programs with compiler overlapped I/O and computation.

Bordawekar, del Rosario, and Choudhary [3] designed a library of user-accessible primitives which are configured at runtime to the desired memory and disk distributions of the data. We believe such an I/O library can be used effectively by our approach.

Thakur, Bordawekar and Choudhary [10] are working on compiler methods for out-of-core HPF programs. This has many similarities to our work, however, we believe our use of programmer I/O directives provides useful information to the compiler, the discovery of which is still a significant research problem. This allows our work to focus on efficient deferred routines and I/O optimization.

Previous compiler-related work has focused on transformations to improve virtual memory performance. Abu-Sufah [1] examined the application of loop distribution followed by loop fusion to improve locality. Trivedi [11] examined the potential benefits of programmer-inserted prefetching directives for matrix multiplication on the STAR computer and compiler support for demand prepaging on sequential machines [12]. A growing body of work [4, 9] examines similar concerns for cache memories.

Cormen [5] developed efficient out-of-core permutation algorithms and examined their I/O requirements within a data-parallel virtual memory system (VM–DP). He also recommended the development of language and compiler techniques for parallel out-of-core I/O.

8 Conclusions

Currently, applications that require more data than can be stored in main memory are difficult to write. In large part, this is because of the lack of language and system support. We have presented a preliminary design for attacking these problems in a compiler-based system. Using a few high-level directives like those in data-parallel languages, the compiler will insert and optimize input and output statements to convert the program into out-of-core form. The techniques for doing this are based on previous work on the Fortran D compiler. To evaluate the effectiveness and feasibility of this approach, we have hand-compiled a few test cases. The results, while certainly not as good as in-core performance, are encouraging.

We are now beginning an implementation of these ideas as an extension of the Fortran D compiler. The new compiler will allow us to more thoroughly evaluate the effectiveness and applicability of our methods. A successful compiler will allow the easy conversion of

standard algorithms to out-of-core form. The benefits of this for solving problems requiring large memory should be obvious.

Much work, however, remains to be done. Some of our analysis requires more formalization, and our transformations are not fully general yet. It is an open question whether regular section descriptors are sufficiently precise for our purposes. Perhaps most interesting is the question of optimizing tile size. An automatic system must balance considerations of available memory, disk contention, and parallel execution to choose optimal parameters for the I/O process. We see the area of out-of-core computations as an excellent new direction for research with practical applications.

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] Rajesh Bordawekar, Juan Miguel del Rosario, and Alok Choudhary. Design and evaluation of primitives for parallel I/O. In *Proceedings of Supercomputing '93*, pages 452–461, Portland, November 1993.

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

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

[6] 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.

[7] 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.

[8] Ed Kushner. Optimizing I/O performance for the Paragon (tm) supercomputer. *Intel On-Line, http://abacus.training.ssd.intel.com/InfoSelect/bulletinV01N02.html*, 1(2), August 1994.

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

[10] Rajeev Thakur, Rajesh Bordawekar, and Alok Choudhary. Compiler and runtime support for out-of-core HPF programs. In *Proceedings of the 1994 ACM International Conference on Supercomputing*, pages 382–391, Manchester, July 1994.

[11] Kishor S. Trivedi. Prepaging and applications to the STAR-100 computer. In *Proceedings of the Symposium on High Speed Computer and Algorithm Organization*, pages 435–446, April 1977.


```
C   ************************************************
C   Three-dimensional red-black relaxation
C   ************************************************
    double precision a(0:319,0:319,0:319)
    parameter (n$proc = 16)
C   I/O-decomposition iod( 320, 320, 320)
C   I/O-align         a with iod
C   I/O-distribute    iodd( : ,  : , block(10) )
C
C   decomposition     d( 320, 320, 320 )
C   align             a with d
C   distribute        d( : , block, : )

C   Open the file and read data from disk
C   ********** iterate red-black computation. **********
    do  n  = 1, 5
C   Compute red points
      do  k  = 2, 316, 2
       do  j  = 2, 316, 2
        do  i  = 2, 316, 2
         a(i,j,k)    = (a(i+1,j,k) + a(i-1,j,k)
     *          + a(i,j+1,k) + a(i,j-1,k)
     *          + a(i,j,k+1) + a(i,j,k-1) )/6
         a(i+1,j+1,k)= (a(i+2,j+1,k) + a(i,j+1,k)
     *          + a(i+1,j+2,k) + a(i+1,j,k)
     *          + a(i+1,j+1,k+1) + a(i+1,j+1,k-1) )/6
         a(i,j+1,k+1)= (a(i+1,j+1,k+1) + a(i-1,j+1,k+1)
     *          + a(i,j+2,k+1) + a(i,j,k+1)
     *          + a(i,j+1,k+2) + a(i,j+1,k))/6
         a(i+1,j,k+1)= (a(i+2,j,k+1) + a(i,j,k+1)
     *          + a(i+1,j+1,k+1) + a(i+1,j-1,k+1)
     *          + a(i+1,j,k+2) + a(i+1,j,k) )/6
        enddo
       enddo
      enddo
C   Compute black points
      do  k  = 2, 316, 2
       do  j  = 2, 316, 2
        do  i  = 2, 316, 2
         a(i,j,k+1)    = (a(i+1,j,k+1) + a(i-1,j,k+1)
     *          + a(i,j+1,k+1) + a(i,j-1,k+1)
     *          + a(i,j,k+2) + a(i,j,k) )/6
         a(i+1,j+1,k+1)= (a(i+2,j+1,k+1) + a(i,j+1,k+1)
     *          + a(i+1,j+2,k+1) + a(i+1,j,k+1)
     *          + a(i+1,j+1,k+2) + a(i+1,j+1,k) )/6
         a(i,j+1,k)    = (a(i+1,j+1,k) + a(i-1,j+1,k)
     *          + a(i,j+2,k) + a(i,j,k)
     *          + a(i,j+1,k+1) + a(i,j+1,k-1) )/6
         a(i+1,j,k)    = (a(i+2,j,k) + a(i,j,k)
     *          + a(i+1,j+1,k) + a(i+1,j-1,k)
     *          + a(i+1,j,k+1) + a(i+1,j,k-1) )/6
        enddo
       enddo
      enddo
    enddo
C   Write the results out to disk and close file
    end
```

Figure 5: Sequential red-black relaxation program

```
C   ************************************************
C   LU factorization with pivoting.
C   Original pivot detection used library routine,
C   this code assumes data is positive.
C   ************************************************
    double precision a(6401,6400),row1(6400),row2(6400)
    parameter (n$proc = 4)
C   I/O-decomposition iod( 6401,  6400   )
C   I/O-align         a with iod
C   I/O-distribute    iod( : , block(40) )
C
C   decomposition     d( 6401, 6400 )
C   align             a with d
C   distribute        d( block, :   )

    do j = 1, 6400
       pivotEntry = 0.0
C      Find pivot row
       do i = j, 6400
          if( a(i,j) .GT. pivotEntry ) then
             pivotEntry = a(i,j)
             pivotRow   = i
          endif
       enddo
       a( 6401, j) = pivotRow

C      Scale pivot row
       scale =  1.0/pivotEntry
       do i = j, 6400
          a( i, j ) = scale * a( i, j )
       enddo
       a(pivotRow, j) = pivotEntry

C      Copy the row containing diagonal
       if( j .ne. pivotRow ) then
          do i = j, 6400
             row1(i) = a( j, i )
          enddo
       endif

C      Copy the pivot row
       do i = j, 6400
          row2(i) = a( pivotRow, i )
       enddo

C      If pivot row is not diagonal, swap rows.
       if( pivotRow .NE. j ) then
C         DCOPY((6400-j)+1,row1(1),1,a(pivotRow,j),6400)
          do i = j, 6400
             a( pivotRow, i ) = row1(i)
          enddo
C         DCOPY((6400-j)+1,row2(1),1,a(j,j),6400)
          do i = j, 6400
             a( j, i ) = row2(i)
          enddo
       endif

C      Perform outer-product computation
       do i = j+1, 6400
C         DAXPY(6400-j,-row2(i),a(j+1,j),1,a(j+1,i),1)
          do k = j+1, 6400
             a( k, i ) = a( k, i ) - row2(i) * a( k, j )
          enddo
       enddo

     enddo
    end
```

Figure 6: Sequential LU factorization program

[12] Kishor S. Trivedi. On the paging performance of array algorithms. *IEEE Transactions on Computers*, C-26(10):938–947, October 1977.

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

[14] David E. Womble. Sandia National Laboratories, July 1993. Private communication.