A Model and Compilation Strategy for Out-of-Core Data Parallel Programs

Rajesh Bordawekar[*] **Alok Choudhary**[*]
Ken Kennedy[†] **Charles Koelbel**[†] **Michael Paleczny**[†]

Abstract

It is widely acknowledged in high-performance computing circles that parallel input/output needs substantial improvement in order to make scalable computers truly usable. We present a data storage model that allows processors independent access to their own data and a corresponding compilation strategy that integrates data-parallel computation with data distribution for out-of-core problems. Our results compare several communication methods and I/O optimizations using two out-of-core problems, Jacobi iteration and LU factorization.

1 Introduction

There can be no argument that *high-performance* I/O is essential to high-performance computing. Many users see parallelism as the best way to achieve this performance, thus motivating the calls for "parallel I/O." Informal surveys of users of high-performance computing have determined that their needs fall into three categories:

1. **Out-of-core calculations.** Although parallel computing systems typically have large aggregate memories, some programs require huge data structures that cannot fit into memory for the entire duration of a run. These large-memory applications are typically coded so that the data structure resides on disk (hence, it is "out of core") and at any one time only a portion of it resides in memory. Today, out-of-core applications are fairly rare because they are incredibly tedious to implement, requiring that almost every loop be transformed, typically by the programmer. For the purposes of this paper, we will count in the out-of-core category any application that uses disk for temporary array storage.

2. **Checkpointing.** If the user wishes to be able to stop a long run and restart it from intermediate results or restart after a machine crash, the application must write out its intermediate state from time to time. In some cases this must happen quite frequently, even within a time-critical computation loop. If this is not done efficiently, the impact on running time can be very significant.

3. **Real-time I/O.** In many applications it is important to monitor a calculation as it is carried out. If this monitoring involves heavy use of graphics, the output volume could place a significant load on the running time of the computation. Even if the monitoring is going to be done off-line, the volume of data required for playback graphics can also affect running time adversely.

Table 1 shows details of I/O requirements for some Grand Challenge applications [dRBC93]. In terms of the above discussion, temporary working storage generally comes from an out-of-core problem, archival and secondary storage usually come from checkpointing, and bandwidth requirements may be related to real-time I/O or checkpointing. Clearly, if I/O is not handled efficiently it can be an extreme bottleneck in all of these cases.

Although all of these problems are important sources of difficulty for high performance parallel computers, we shall concentrate in this paper on the problem of supporting out-of-core arrays. We choose this problem both because it is important and because it is amenable to

[*]NPAC and ECE Dept. Syracuse University, supported in part by NSF Young Investigator Award CCR-9357840, grants from Intel SSD and IBM Corp., and by USRA CESDIS Contract #5555-26. This work was performed in part using the Intel Touchstone Delta system operated by Caltech on behalf of the Concurrent Supercomputing Consortium. Access to this facility was provided by CRPC.

[†]CRPC and CS Dept., Rice University. This research was supported in part by the Center for Research on Parallel Computation (CRPC) at Rice University, under NFS Cooperative Agreement Number CCR-9120008. This work was performed in part using the Intel Paragon installed at Rice University.

Application	I/O requirements
Environmental modeling	T: 10s of GB. S: 100s of MB - 1 GB per PE. A: Order of 1 TB.
4-D data assimilation	S: 100 MB - 1 GB/run. A: 3 TB database. Expected to increase by orders of magnitude with the Earth Observing System (EOS) - 1 TB/day.
Particle algorithms in Cosmology and Astrophysics	S: 1-10 GB/file; 10-100 files/run. B: 20-200 MB/s
HP Computational Chemistry	B: 10-30 MBytes/s during execution. 10-100 MBytes/s post-processing.
Molecular Computation	B: 30 MBytes/s. S: 1-10 GBytes during execution.
Computational Fluid and Combustion Dynamics	A: 1 TBytes. B: 0.5 GBytes/s to disk, 45 MBytes/s to disk for visualization.

Table 1: I/O Requirements of Grand Challenge Applications, (A: Archival Storage, T: Temporary Working Storage, S: Secondary Storage, B: I/O Bandwidth)

attack using the compiler and runtime strategies that we have developed for Fortran D and HPF. I/O extensions to Fortran D are being implemented within the D System at Rice University, while the PASSION project at Syracuse University targets HPF.

In this paper, we will assume an abstract machine model in which a number of processors are interconnected via a high-speed network. We consider two basic models for I/O on such a machine:

- In the simpler model, each processor is connected to a local disk of its own. In this scheme, a disk access is either to a local disk or a remote disk on a remote node. In the latter case, the access must be handled by exchanging messages with the node owning the remote disk.
- In the more complex model, there are two kinds of nodes—*compute nodes* and *I/O nodes*. In this scheme, every disk access by a compute node requires a message exchange with the I/O node owning the disk being accessed.

As mentioned above, our approach to managing out-of-core arrays is based on compilers and runtime systems we have developed for Fortran D and High Performance Fortran. In these languages, the major program arrays are distributed across the memories of a parallel computer system according to distribution specifications that are part of the language. For the purposes of supporting out-of-core arrays, we will extend this notation by the addition of specifications for disk distributions as well as memory distributions. Thus, each array that may reside out-of-core will have a layout that combines two distributions, one for memory and the other for disk. The compiler and runtime system will be responsible for organizing and carrying out the transfer of information between disk and memory, handling all the details of strip mining and buffering.

The plan for this paper is to describe compilation strategies for compiling out-of-core data parallel programs. In Section 2, we describe the programming model that a typical user would see. Section 3 describes the execution model for the program, while Section 4 describes the design of a compiler targeting these models. Section 5 addresses the run-time libraries which implement the abstract execution model on physical hardware. Preliminary experimental results are discussed in Section 6. We review related work in Section 7 then present our conclusions and ideas for future work in Sections 8 and 9.

2 User Programming Model

Our out-of-core programming model is inspired by the data-parallel programming paradigm. In essence, data-parallel programs apply the same conceptual operations to all elements of large data structures. This form of parallelism occurs naturally in many scientific and engineering applications such as partial differential equation solvers and linear algebra routines [Fox91]. In these programs, a decomposition of the data domain exploits the inherent parallelism and adapts it to a particular machine. Compilers can use programmer-supplied decomposition patterns such as block and cyclic to partition computation, generate communication and synchronization, and guide optimization of the program. Languages based on this principle are called data-parallel languages and include High Performance Fortran (HPF) [Hig93], Vienna Fortran [ZBC+92], and Fortran D [FHK+90]. Our out-of-core approach builds on HPF.

The DISTRIBUTE directive in HPF partitions an array among processors by specifying which elements of the array are mapped to each processor. This results in each processor storing a *local array* associated with each array in the HPF program; the local array is typically a section of the entire array. The compiler divides the computation among the processors, attempting to exploit the parallelism of the program while placing each operation so that it accesses local data. One popular heuristic for doing this is the owner-computes rule, which maps each assignment statement to the processor storing the left-hand side. When a computation uses nonlocal data (for example, when adding array elements stored on two different processors), the compiler must insert communication to fetch the data. On shared-memory machines, this may simply be a read instruction; on message-passing hardware, it may require *send* and *receive* library calls. On most parallel architectures, it is vital to optimize the communication operations by using efficient libraries, advanced compilers, or both.

For out-of-core computations, we extend HPF's DISTRIBUTE directive to partition very large arrays for limited memory. Consider an array that is too large to fit in main memory; we will refer to such arrays as *Out-of-Core Arrays* or *OCAs*. In our model, the user declares OCAs to be their full size, that is, the size they are stored on disk. Only a small section can fit in memory at one time, however. We introduce a new directive, IO-DISTRIBUTE, that divides an OCA into pieces small enough to fit in memory.[1] We call the in-memory pieces *In-Core Arrays* or *ICAs*; they are analogous to the local arrays described above. The programmer may suggest a desired size for an ICA by adding a parameter to the BLOCK directive, e.g., BLOCK(10). Analogously to data-parallel compilation, the system must move ICAs between memory and disk and map the computations to ICAs to maximize locality. In this case, locality means that operations should access (read or write) ICAs that are in memory at the same time. As we will see, strategies similar to data-parallel languages can be used to implement and optimize the ICAs. These optimizations are analogous to collecting messages in data-parallel compilation. Providing both DISTRIBUTE and IO-DISTRIBUTE directives allows the user to control both the parallelism and the I/O behavior of the program. If both directives affect an array, they are applied in the order they appear in the source program.

Since we consider memory to be a limited resource, the compiler must be aware of how much memory is available to determine the size of buffers for asynchronous I/O or to automatically generate I/O distributions. This information might be obtained from a compiler configuration file containing a description of system resources and the results of performance benchmarks. An alternative is providing the programmer with a MEMORY directive or -M MEMORY_SIZE compiler flag.

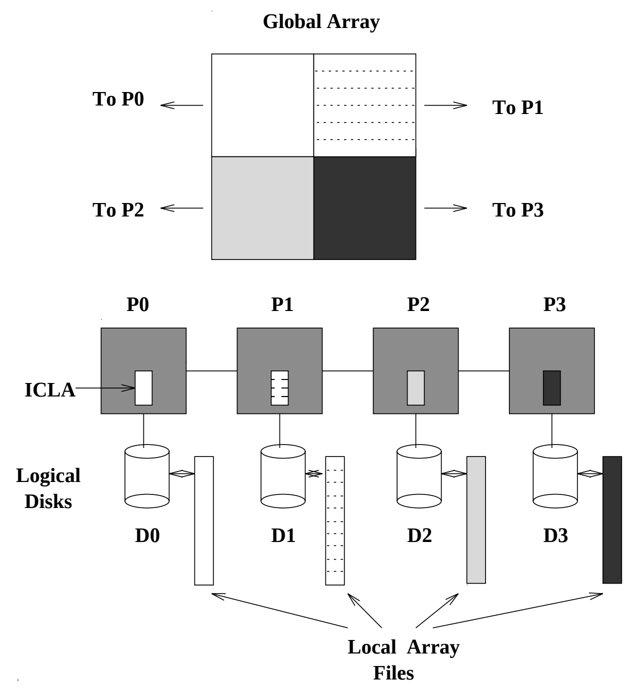

Figure 1: Local Placement Model for out-of-core compilation

There is an important complication to out-of-core computations on parallel computers. On a parallel machine, each ICA may itself be partitioned among many processors. Thus, a second level of mapping is needed. When an ICA is distributed, we refer to the section on each processor as the *In-Core Local Array* or *ICLA*. It is sometimes convenient to refer to the portion of the OCA that is mapped to a single processor. We call this section of the array the *Out-of-Core Local Array* or *OCLA*. (This is equivalent to the union of the ICLAs of that OCA mapped to that processor.)

3 Execution Model

The user program will be translated into a lower-level model for execution. Figure 1 illustrates the model used by our systems, called the *Local Placement Model*. The simplest way to view this model is to think of each processor as having a virtual disk that acts as another (albeit slower) level of memory. In other words, it is a straight-forward extension of the usual distributed memory model. The one-to-one mapping of disks to processors is conceptually convenient and, as we will see in Section 5, can be efficiently implemented by runtime libraries on many machines. Later, we will see examples

[1] There is also an extension of the HPF ALIGN directive called IO-ALIGN, but we will not need it for the examples in this paper.

```
1      REAL A(1024,1024), B(1024,1024)
         ..........
2      !HPF$ PROCESSORS P(4,4)
3      !HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO P ::A,B
         ...........
4      FORALL (I=2:N-1, J=2:N-1)
5              A(I,J) = (B(I,J-1) + B(I,J+1)
    &                 + B(I+1,J) + B(I-1,J))/4
6      END FORALL
```

Figure 2: HPF Program Fragment

of efficient library operations that involve collections of processors; these functions resemble collective communications in parallel processors.

To implement out-of-core computations in the Local Placement Model, each processor stores its out-of-core local array in a separate logical file called the *Local Array File (LAF)* on its disk. The LAFs can be stored as separate files or they can be part of a common file; we will assume they are separate files. The node program explicitly reads from and writes into the LAF when required. In this model, a processor cannot explicitly operate on a file owned by a different processor. If a processor needs to read data from a file owned by a different processor, the required data will be read by the owner and then communicated to the requesting processor. Since each local array file contains the OCLA of the corresponding processor, the distributed (or the user-specified) view of the out-of-core global array is preserved.

In order to store the data on the disks based on the distribution pattern specified in the program, redistribution of the data may be needed in the beginning when the data is staged. This is because the way data arrives (e.g. from archival storage or over the network) may not conform to the distribution specified in the program. Redistribution requires reading the data from the external storage, shuffling the data over the processors and writing the data to the local virtual disks. This increases the overall cost of data access. This cost can be amortized if the out-of-core array is used repeatedly.

4 Compilation Strategies

This section describes the basic compilation strategy for HPF arrays and then discusses the additional support needed for handling out-of-core arrays. We explain the compilation strategy for both cases with the help of the HPF program fragment given in Figure 2. In this example, arrays `A` and `B` are distributed by two-dimensional blocks on a 4×4 grid.

Figure 3 shows the steps used in compiling an array or `FORALL` statement. The strategy for compiling an entire program is conceptually identical, but may involve more

1. Analyze the distribution pattern of each array used in the `FORALL`.
2. Depending on the distribution, detect the type of communication required.
3. Perform data partitioning and calculate local lower and upper bounds for each participating processor.
4. Use temporary arrays if the same array is used in both LHS and RHS of the `FORALL` body.
5. Generate the corresponding loosely synchronous SPMD node program.
6. Add calls to runtime libraries to perform collective communication.

Figure 3: An HPF Compilation Strategy on Distributed Memory Machines

```
Call communication routine to perform overlap shift.
do j = lower_bound, upper_bound
    do i = lower_bound, upper_bound
        A(i,j)=((B(i,j-1)+B(i,j+1)
            +B(i-1,j)+B(i+1,j))/4)
    end do
end do
```

Figure 4: Translation of a `FORALL` statement.

complex analysis to link the statements together. The compiler uses distribution directives (Figure 2, lines 2-3) in the source program to find the distribution pattern of the arrays. Using the distribution information, arrays are partitioned into local arrays. After data distribution, the compiler analyzes the `FORALL` body (Figure 2, Line 5). Since array `B` is distributed in a block-block fashion over 16 processors, the array expression requires fetching data from up to four neighboring processors. The HPF compiler generates a call to a specific collective communication routine (*overlap shift*), and the array expression is sequentialized into `DO` loops, as shown (in pseudocode form) in Figure 4.

4.1 Out-of-core Compilation

For out-of-core programs, in addition to the steps in Figure 3, the compiler must also perform many more functions, e.g., schedule explicit I/O accesses to fetch/store appropriate data from/to disks. The compiler has to take into account the data distribution and memory at each node, the communication patterns between ICLAs, the data distribution on disks, the number of disks used for storing data, and the prefetching/caching strategies used. The last three items are encapsulated in the compiler's choice of a placement model for data storage. In this section, we use the Local Placement Model to extend data parallel compilation methods to out-of-core

1. Data-parallel Phase
   (a) Analyze parallelism and I/O distribution patterns of each array.
   (b) Partition computation according to parallelism distribution patterns.
   (c) Determine communication required for array accesses.
   (d) Determine local space bounds.
2. Out-of-core Phase
   (a) Partition computation (in local space) according to I/O distribution patterns.
   (b) Determine I/O required for array accesses.
   (c) Determine in-core bounds.
3. Code Generation Phase
   (a) Sequentialize local code.
   (b) Insert communication and I/O.
   (c) Optimize communication and I/O.

Figure 5: Compilation Phases in the Local Placement Model

problems. We do this by introducing partitioning along ICA boundaries and I/O insertion. These steps are analogous to the computation partitioning and communication insertion steps in data-parallel compilation.

Figure 5 presents the proposed compilation strategy for applying the data-parallel and I/O distributions suggested by the corresponding `DISTRIBUTE` statements in the source program. The compiler is conceptually divided into three phases. The data-parallel phase distributes data to multiple processors. First the HPF arrays are partitioned according to the parallelism directives and local lower and upper bounds for each local array are calculated. Statements (such as `FORALL`) are then analyzed to determine the required communication. In other words, the compilation in this phase proceeds in the same manner as for in-core computations, except that it stops short of generating the SPMD code. This is a direct consequence of the model. The out-of-core phase partitions out-of-core data into in-core arrays, according to the `IO-DISTRIBUTE` statement. (For convenience, the I/O partitioning directives are propagated in the same phase as the parallelism directives, although they are independent.) The resulting in-core arrays are analyzed to determine inter-array communication and the computation is partitioned to operate on one in-core array at a time. In-core local arrays are determined by the intersection of local arrays and in-core arrays. The code generation phase inserts and optimizes communication and I/O, generating an SPMD node program with explicit message passing and I/O operations. This process sequentializes the in-core local operations and inserts the necessary I/O and communication calls. Section 4.2 describes two models for generating this communication; which is chosen will affect the placement, organization, and efficiency of both communication and I/O. The final optimizations applied include overlapping I/O with computation and maintaining data in memory between processing of in-core local arrays. Results from applying the first optimization are presented in Section 6.

Some additional issues need to be considered when distributing out-of-core data. Since each processor performs computation on the data in an ICLA, the portion of the local array currently required for computation is fetched from disk into memory. The size of the ICLA is specified at compile time and usually depends on the amount of memory available. Although a larger ICLA reduces the number of disk accesses, this benefit must be balanced against other memory requirements, e.g., buffer space to overlap I/O and computation, and storage for operands needed to perform the local in-core computation. When operand storage exceeds the space available at the node, the computation must be partitioned further and I/O requests must be inserted into the in-core local computation.

4.2 Compiling Communication

For the `FORALL` in Figure 2, the communication and I/O requirements for processor 5 are shown in Figure 6(B). The out-of-core local array (OCLA) has four slabs, each of which is equal to the size of the in-core local array (ICLA). The slabs are shown using different patterns. The figure also shows the overlap area for array `B` in this example, which consists of the data to be communicated per processor. This communication can be executed using either of the following two methods, the *Out-of-core Communication Method* and the *In-core Communication Method.*

4.2.1 Out-of-Core Communication Method

In this method, the compiler determines what off-processor data will be required for the entire OCLA. That is, the entire light gray area in Figure 6(C) is communicated in one step. Figure 7(A)) shows the corresponding pseudocode. Note that communication in this case requires that the sending processor read its local array file, select data, and send it to the destination processor. This processor stores the data in its own local file, effectively expanding its OCLA and the ICLAs to include the overlap area. Thus, even the communication step requires accessing the secondary storage for both send and receive operations. The computation can then be performed using only the ICLA (i.e. without interprocessor communication), since all the data has already been communicated.

The key feature of this method is that communica-

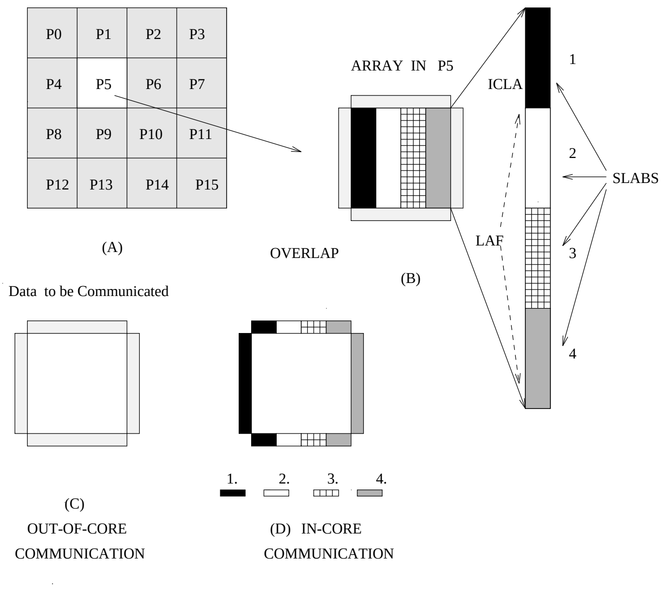

Figure 6: Compiling the `FORALL` Statement in the Local Placement Model

tions operations may require access to disk. However, this method allows the compiler to identify and optimize collective communication patterns because the communication pattern depends on the logical shape of arrays and the access patterns for the entire array (and not on individual slab size and/or shape). For example, there are four shift communications in the example shown in Figure 7. This communication pattern is preserved, although the implementation is somewhat different and the message size rather large. Libraries like the one described in Section 5 can efficiently execute this type of pattern.

There are two situations where this method should not be applied. The first occurs when there is a very large amount of inter-processor communication during processing of an in-core tile. The out-of-core communication method stores all the received data into the local array file which can lead to significant disk access costs and increases in file size. The second is when there are true dependences from one in-core local array to another. A true dependence between in-core local arrays means that the value in the destination slab may be modified after starting computation on the source slab. Thus, it would not be correct to perform I/O for the entire out-of-core array before starting computation.

4.2.2 In-Core Communication Method

Figure 6(D) shows an example of the In-core Communication method. In this method, the compiler analyzes each slab or ICLA instead of the entire out-of-core array. Thus, the individual shaded regions in Figure 6(D) are each sent in separate steps (each shade in Figure 6(D) represents data to be communicated to

(A) Out-of-Core Communication

```
C     Schedule communication for entire out-of-core data.
    Call communication routine to perform overlap shift.
C     Partition code based on the local memory Size.
C     Repeat operation k times (once for each ICLA).
        do 10 l=1, k
C     Perform local I/O access.
    Call I/O routine to read the ICLA and overlap region.
C           Perform local computation on the ICLA.
              do j = lower_bound, upper_bound
                   do i = lower_bound, upper_bound
                        A(i,j)=((B(i,j-1)+B(i,j+1)
                              +B(i-1,j)+B(i+1,j))/4)
                        end do
                   end do
C           Perform local I/O access.
              Call I/O routine to store the result.
    10 enddo
```

(B) In-Core Communication

```
C           Partition code based on memory size.
C           Repeat k times (once for each ICLA).
              do 10 l=1, k
C           Perform local I/O access.
C           Schedule communication for in-core data.
              Call communication routine for overlap region.
C           Perform local computation on the ICLA.
              do j = lower_bound, upper_bound
                   do i = lower_bound, upper_bound
                   A(i,j)=((B(i,j-1)+B(i,j+1)
                   +B(i-1,j)+B(i+1,j))/4)
                   end do
              end do
C           Schedule communication for in-core data.
                   Call communication routine.
C           Store local data.
                   Call I/O routine to store the results.
    10       enddo
```

Figure 7: Compilation Alternatives for Out-of-core Programs in the Local Placement Model

the correspondingly shaded slab shown in Figure 6(B)). Once the necessary *comm_data* is fetched, the computation on each *slab* begins. After the computation is over, the *comm_data* is again communicated or *scattered*.

The corresponding pseudocode is shown in Figure 7(B). In this case, each ICLA must be analyzed to identify needed out-of-core and nonlocal data. Local out-of-core data is obtained from the OCLA, and nonlocal in-core data is received from its "owner." (Obviously, previously communicated values may be reused as well.) The possible access methods to obtain nonlocal out-of-core data can be broken down as follows. If the owning processor previously had the data in memory, the compiler can schedule it to communicate the data at that time and schedule the requesting processor to store the data in its local array file. Otherwise, a two-phase method [CBH+94] where the owner reads the data and sends it to the requesting processor when needed can be used. A final possibility applies if there is some processing capability at the I/O node itself, in which case disk-directed I/O can be used to send the data [Kot94].

The most important point to note here is that data needed by other processors is communicated while the slab is in memory when possible. Thus, this method reduces the number of disk accesses, produces smaller individual messages (although the total communication volume is the same), and is more easily applied to serial loops. However, a structured communication pattern is transformed into distinct patterns (for each slab) as shown by the different shapes of the shaded regions in Figure 6(D). Optimizing such communication patterns may be difficult and require extensive compile-time analysis.

5 Run-time Libraries

In this section, we show how the Local Placement Model of Section 3 can be mapped onto more realistic hardware. Most current parallel machines do not have one disk on every node. Instead, a set of nodes are connected to a set of disks via an interconnection network. The I/O subsystem may have a separate interconnection network or the disks may be embedded in the processor grid (and share the same interconnection network). This section describes the runtime library to implement the Local Placement Model. The runtime routines can be broadly classified as 1) Mapping Routines, 2) Access Routines, and 3) Routines for collective communication.

1. Mapping Routines: Mapping routines map the execution model (in this case, the Local Placement Model) on the underlying machine. Mapping routines include functions to map virtual disks to physical disks. That is, for a system that has N processing nodes and D disks, this routine associates each processing node, $p_i, 1 \leq i \leq N$, to one or more disks, $d_j, 1 \leq j \leq D$. Mapping of individual local files also depends on this mapping. One way of achieving this mapping is to associate each processor with a distinct set of disks. Another way could be to associate each processing node with all disks, and thus stripe each local file over all the disks. Some of these options may not be possible if the underlying file system does not provide the required flexibility.

Another set of routines needed to implement the Local Placement Model is the redistribution routines which will take the input data files, and based on the distribution, create the local files. Each file will have to be redistributed into local files based on the out-of-core distribution. These routines use the two-phase access strategy [dRBC93] which will read the data from the input data file using the most optimal access pattern (which depends on how the data is stored on disks) and redistribute the data over the processors using the high speed processor interconnection network. The processors will then store the data into appropriate local files.

2. Access Routines: These routines are used to stage data in the memory for out-of-core computations once the local files are created. These routines essentially include read and write functions. Each of these functions is provided the offsets, strides and the size of the data to be accessed in terms of the out-of-core array and the in-core array. It is the responsibility of these routines to convert these accesses into appropriate file accesses.

3. Routines for Collective Communication: In general, a collective communication routine implements a pattern of communication (e.g. shift) which is implemented using a set of send and receive calls inside the routine. However, when arrays are out-of-core, a collective communication routine also requires accessing files because the data to be communicated may not be present in memory at that time. Thus, an implementation of a collective communication routine requires reading data from local files, communicating data to the appropriate destination processors, and finally, writing the data into the files of receiving processors.

A detailed description of these routines is given in [CBH+94].

6 Experimental Results

This section presents experimental results for three out-of-core applications: Laplace equation solver by the Jacobi iteration method, LU factorization with pivoting, and three dimensional red-black relaxation. These applications were compiled by hand using the local placement model, storing the OCLAs for each processor into a separate file.

The Jacobi iteration application is essentially a repeated execution of the code in Figure 2. New values in each iteration are computed using the values from the previous iteration. This requires the newly computed array to be copied into the old array for the next iteration. In the out-of-core case, this would require copying the local array file. We do an optimization in which, instead of explicitly copying the file, the file unit numbers are exchanged after each iteration. This is equivalent to dynamically changing the virtual addresses associated with arrays. Hence the program uses the correct file in the next iteration. In the out-of-core program, the array is distributed across processors by blocks of columns rather than the 2-dimensional distribution shown earlier. The I/O distribution is then applied by blocks of

Table 2: Performance of Laplace Equation Solver (time in sec. for 10 iterations)

	Array Size: $2K \times 2K$		Array Size: $4K \times 4K$	
	32 Procs	64 Procs	32 Procs	64 Procs
Direct File Access	73.45	79.12	265.2	280.8
Explicit Communication	68.84	75.12	259.2	274.7
Explicit Communication with data reuse	62.11	71.71	253.1	269.1

columns, within the processor blocks.

The performance of the Jacobi program on the Intel Touchstone Delta is given in Table 2. We use the out-of-core communication method and compare the performance of the three implementations—*direct file access*, *explicit communication* and *explicit communication with data reuse*. In the direct file access version, each processor reads data from the local array file of some other processor as required by the communication pattern. This requires explicit synchronization at the end of each iteration. In the explicit communication version, each processor accesses only its own local array file. Data is read into memory and sent to other processors. Similarly, data is received from other processors into main memory and then saved on disk. In the third method, data which was communicated in an earlier iteration is reused again in the next iteration. In our example, this occurs when the overlap area needed by an adjacent processor on the next iteration is sent after performing computation but before the data is written to disk.

We observe that the direct file access method performs the worst because of contention for disks. The best performance is obtained for the explicit communication method with data reuse as it reduces the amount of I/O by reusing data already fetched into memory. If the array is distributed in both dimensions, the performance of the direct file access method is expected to be worse because in this case each processor, except at the boundary, has four neighbors. So, there will be four processors contending for a disk when they try to read the boundary values, thus increasing the overall access time. For other data patterns, however, the direct access method might get some advantage from disk cacheing at the physical I/O nodes. This points to an advantage of runtime libraries for these operations—they can be tuned for particular access patterns rather than relying on the underlying operating system for performance.

For LU factorization, we use a block distribution by rows across processors and a block distribution by columns for the I/O. The in-core algorithm would normally modify all columns to the right after scaling a column with its pivot element. For the out-of-core version, when the columns to the right are not in the current in-core array, their modification is "deferred" until the slab containing them is brought into memory. This is consistent with the ownership approach for an in-core local array. When the slab is in memory, each processor must read the columns on the left to obtain the data necessary to perform the deferred operations. Essentially, this converts the "right-looking" variant of LU factorization into a "left-looking" one.

The performance results for LU factorization on an Intel Paragon with two I/O nodes are shown in Tables 3 and 4. We compare three implementations: (1) virtual memory, (2) synchronous I/O using the local placement model, and (3) asynchronous I/O overlapped with computation. All three implementations use the same memory and computation tiling to allow fair comparisons.[2] Inter-processor communication is performed using the in-core communication method with explicit communication. The asynchronous I/O version is derived from the synchronous I/O code by moving I/O across computation when the regular section descriptors (RSDs) do not intersect. Table 4 shows the performance of a 1600×1600 factorization problem. These results show a consistent 35 to 40 percent improvement in performance using file I/O operations over virtual memory. Table 3 shows the results for a 6400×6400 case. The Paragon system we used did not provide enough virtual memory paging space for the larger problem, so we were unable to provide comparisons to virtual memory. We can, however, compare synchronous with asynchronous I/O in this case. Although this is a compute-bound application, there is a 5 to 10 percent savings from overlapping computation on the in-core array with reading operands needed for further computation on the ICLA.

Red-black relaxation is an iterative finite-differences method similar to Jacobi iteration in its I/O access requirements. For this application we used one dimensional block data-parallel and I/O distributions in separate dimensions of the array; the I/O distribution tiles the last array dimension. The I/O accesses for red-black relaxation consist of a sequential read of the matrix and write of the new values each iteration. The sequential read is divided into successive reads of in-core local arrays which are overlapped with computation of new red

[2]In the case of virtual memory, this was accomplished by changing the execution order of loops, an optimization that improved performance by 200 times.

Table 3: Performance of LU factorization with pivoting (time in sec.)

Number of Processors	Array Size: 6400×6400			
	Synchronous I/O	Overlapped I/O	% Overlap	% of Optimal Overlap
1	12339	10991	11	67
2	7920	7324	7.5	71
4	3464	3206	7.4	71
8	1804	1700	5.7	62
16	1110	999	10	61

Table 4: Performance of LU factorization using virtual memory (time in sec.)

Number of Processors	Array Size: 1600×1600		
	Virtual Memory	Synchronous I/O	% reduction
1	483	284	41.2
2	303	195	35.6
4	121	71.6	40.8

and black data points.

The execution times for red-black relaxation are presented in Table 5. When the problem size is out-of-core, 1 through 8 processors, overlapping I/O and computation is the fastest followed by the synchronous I/O version; virtual memory is slower, in part, because the request sizes for paging are smaller. At sixteen processors, the problem was almost entirely in-core and performance "using" virtual memory improved dramatically. The last column compares synchronous I/O with overlapped asynchronous I/O. There is a 10% to 27% speedup using 1 through 8 processors. The reduced improvement that occurs at eight and sixteen processors is caused by two effects. The first is a decrease in computation time at each node, the second is contention in the I/O system. Although we are scaling the number of processors, we are not scaling the number of I/O nodes or disks.

7 Related Work

Tiling of out-of-core programs has been done by many applications programmers and we wish to acknowledge their extensive previous work. As is often the case, programmers suffer in advance of compiler writers. Our approach integrates compiler management of out-of-core data sets with the data-parallel approach of languages such as Fortran (HPF) [Hig93], Vienna Fortran [ZBC$^+$92], and Fortran D [FHK$^+$90].

Previous work on compiler improvements at the memory to disk interface starts with Abu-Sufah and Trivedi at the end of the 1970's. Abu-Sufah [AS79] demonstrates that applying loop distribution and loop fusion can reduce the space-time costs for numerical algorithms. Trivedi [Tri77a, Tri77b] shows that profitable opportunities for demand prefetching can be identified from a program's syntax. Highlights of managing other aspects of the memory hierarchy include: Allen and Kennedy on vector register allocation [AK87], Carr and Kennedy on compiler blocking of scientific codes [CK92], and Mowry on software prefetching for cache [Mow94].

Related projects include disk-directed I/O, by David Kotz at Dartmouth College, in which I/O processors direct the transfer of data from disk to processors. The Jovian framework for optimizing parallel I/O, being developed at the University of Maryland, optimizes independent and collective I/O requests at run-time. MPI-IO, designed by IBM T.J. Watson and NASA Ames Research Centers, provides parallel file I/O with a message-passing interface. PANDA, from the University of Illionois at Urbana-Champaign, explores the performance and user interface benefits of array "chunking". PIOUS, developed at Emory University, uses a transaction-based model to provide consistency. Further details about these projects and many others are available at "`html://www.cs.dartmouth.edu/pario.html`".

8 Conclusions

High Performance Fortran has already generated a great deal of interest in the user and vendor community as a language for portable parallel programming on high-performance computers. In order to permit implementation of large-scale problems, e.g., many grand challenge problems, support for out-of-core compilation is important.

For out-of-core compilation, the compilation strategy depends on how data is stored as well as how it is accessed. We presented a data storage model, the Local Placement Model, to organize, view, and access out-of-core data. We also described a general compilation methodology that uses this model. Furthermore, we demonstrated how communication strategies are af-

Table 5: Performance of Red-Black relaxation (time in sec. per iteration)

	Array Size: $320 \times 320 \times 320$			
Number of Processors	Virtual Memory	Synchronous I/O	Overlapped I/O	% Overlap
1	634	316	277	12.3
2	368	227	181	20.3
4	375	186	136	27.0
8	334	132	119	10.1
16	62[1]	159	153	3.9

[1]With 16 processors, the problem is almost entirely in-core.

fected and presented some alternatives for implementing them for out-of-core data.

Although the techniques described in this paper are discussed with respect to HPF, they are applicable to data parallel languages in general.

9 Future Work

Once we have established the notion of an out-of-core array, it is an easy step to a *persistent* out-of-core array, that is, one that has a permanent home on disk and which can be passed between different programs.

In addition to the Local Placement Model described in Section 3, alternate storage methods are possible. One that we plan to investigate is the Global Placement Model, which would maintain all the data in one file. This file could be distributed to disk devices by the file system.

As a final matter, we will consider the impact of out-of-core arrays in the distributed computing environment, which involves collections of high-performance workstations interconnected by high-speed networks.

References

[AK87] J. R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[AS79] W. Abu-Sufah. *Improving the Performance of Virtual Memory Computers*. PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1979.

[CBH+94] A. Choudhary, R. Bordawekar, M. Harry, R. Krishnaiyer, R. Ponnusamy, T. Singh, and R. Thakur. PASSION: Parallel and Scalable Software for Input-Output. Technical Report SCCS–636, NPAC, Syracuse University, Sep 1994.

[CK92] S. Carr and K. Kennedy. Compiler blockabilty of numerical algorithms. *Proc. of Supercomputing'92*, November 1992.

[dRBC93] J. del Rosario, R. Bordawekar, and A. Choudhary. Improved parallel I/O via a two-phase runtime access strategy. In *Proceedings of the Workshop on I/O in Parallel Computer Systems at IPPS '93*, April 1993.

[FHK+90] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C. Tseng. Fortran D language specifications. Technical Report COMP TR90-141, Rice University, 1990.

[Fox91] G. Fox. The architecture of problems and portable parallel software systems. Technical Report SCCS-78b, Northeast Parallel Architectures Center, Syracuse University, Syracuse, NY 13244, 1991.

[Hig93] High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, 2(1-2):1–170, 1993.

[Kot94] D. Kotz. Disk-Directed I/O for MIMD multiprocessors. Technical Report PCS-TR94-226, Dept. of Computer Science, Dartmouth College, July 1994.

[Mow94] T. Mowry. *Tolerating Latency Through Software Controlled Data Prefetching.* PhD thesis, Department of Computer Science, Stanford University, March 1994.

[Tri77a] K. 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.

[Tri77b] K. S. Trivedi. On the paging performance of array algorithms. *IEEE Transactions on Computers*, C-26(10):938–947, October 1977.

[ZBC+92] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. Vienna Fortran - a Language Specification. Technical Report ICASE Interim Report 21, MS 132c, ICASE, NASA, Hampton VA 23681, 1992.