Proc. IEEE Frontiers'96 of Massively Parallel Computing Symp.

Disk Resident Arrays: An Array-Oriented I/O Library for Out-of-Core Computations

Jarek Nieplocha
Pacific Northwest National Laboratory
Richland, WA 99352
j_nieplocha@pnl.gov

Ian Foster
Argonne National Laboratory
Argonne, IL 60439
foster@mcs.anl.gov

Abstract

In out-of-core computations, disk storage is treated as another level in the memory hierarchy, below cache, local memory, and (in a parallel computer) remote memories. However, the tools used to manage this storage are typically quite different from those used to manage access to local and remote memory. This disparity complicates implementation of out-of-core algorithms and hinders portability. We describe a programming model that addresses this problem. This model allows parallel programs to use essentially the same mechanisms to manage the movement of data between any two adjacent levels in a hierarchical memory system. We take as our starting point the Global Arrays shared-memory model and library, which support a variety of operations on distributed arrays, including transfer between local and remote memories. We show how this model can be extended to support explicit transfer between global memory and secondary storage, and we define a Disk Resident Arrays library that supports such transfers. We illustrate the utility of the resulting model with two applications, an out-of-core matrix multiplication and a large computational chemistry program. We also describe implementation techniques on several parallel computers and present experimental results that demonstrate that the Disk Resident Arrays model can be implemented very efficiently on parallel computers.

1 Introduction

We address the problem of managing the movement of large arrays between different levels in the storage hierarchy of a parallel computer. As is well known, parallel programming can be simplified by libraries or languages that support distributed array abstractions and that manage automatically the often onerous conversions between local and global indices [1,2]. In this paper, we examine how such abstractions can be extended to allow programmer management of data transfer between memory and secondary storage. In particular, we describe the design, implementation, and evaluation of a system called Disk Resident Arrays that extends the distributed array library called Global Arrays (GA) to support I/O.

The GA library [2, 3] implements a shared-memory programming model in which data locality is managed explicitly by the programmer. This management is achieved by explicit calls to functions transferring data between a global address space (a distributed array) and local storage. In this respect, the GA model has similarities to distributed shared-memory models that provide an explicit acquire/release protocol [4,5]. However, the GA model acknowledges that remote data is slower to access than local data and allows data locality to be explicitly specified and used. It is also distinguished by its focus on array data types and blocked access patterns and by its support for collective operations. The GA library allows each process in a MIMD parallel program to access, asynchronously, logical blocks of physically distributed matrices, without the need for explicit cooperation by other processes. This functionality has proved useful in numerous computational chemistry applications, and today many programs, totally nearly one million lines of code, make use of GA (R.J. Harrison, personal communication), with NWChem [6] alone exceeding 250,000 lines.

The GA model is useful because it exposes to the programmer the Non-Uniform Memory Access (NUMA) characteristics of modern high-performance computer systems, and by recognizing the communication overhead for remote data transfer, it promotes data reuse and locality of reference. As illustrated in Figure 1, the Disk Resident Arrays (DRA) model extends the GA model to another level in the storage hierarchy, namely, secondary storage. It introduces the concept of a disk resident array—a disk-based representation of an array–and provides functions for transferring blocks of data between global arrays and disk resident arrays. Hence, it allows programmers to access data located on disk via a simple interface expressed in terms of arrays rather than files. The benefits of global arrays (in particular, the absence of complex index calculations and the use of optimized, blocked communication) can be extended to programs that operate on arrays that are too large to fit into memory.

Disk Resident Arrays have a number of uses. They can be used to checkpoint global arrays. Implementations of out-of-core computations can use disk resident arrays to implement user-controlled virtual memory, locating arrays that are too big to fit in aggregate main memory in disk resident arrays, and then transferring sections of these disk resident arrays into main memory for use in the computation. DRA functions are used to stage the disk resident array into a global array, and individual processors then use GA functions to transfer global array components into local storage for

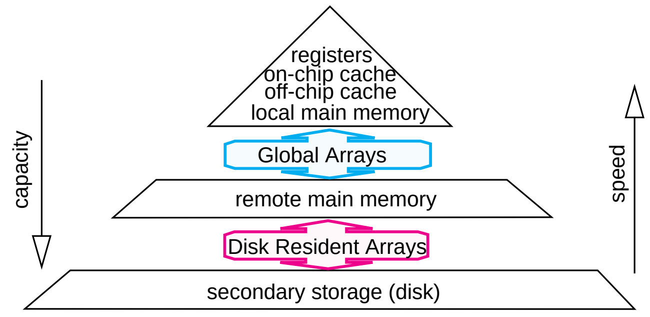

Figure 1: Access to NUMA memory with Global Array and Disk Resident Array libraries

computation. If the global array is updated, a DRA write operation may be used to write the global array back to the appropriate component of the disk resident array. The matrix multiplication algorithm described below has a similar structure.

While the DRA library forms a stand-alone I/O package, it also forms part of a larger system called ChemIO. The ChemIO project is developing high-performance I/O techniques specialized for scalable computational chemistry algorithms designed to exploit future Teraflop systems. These techniques are encapsulated in a standard I/O API designed to meet the requirements of chemistry applications. This API supports three distinct I/O patterns: disk resident arrays, exclusive access files (independent I/O on scratch files allocated on a per-processor basis), and shared files (independent I/O on scratch files shared by multiple processors). Application experiments, performance optimization, and the development of an integrated ChemIO system are ongoing activities.

The principal contributions of this paper are as follows:

1. The definition of a high-level I/O model, Disk Resident Arrays, that provides a particularly simple interface to parallel I/O functionality for scientific applications requiring out-of-core or checkpointing mechanisms.

2. The integration of disk resident arrays and global arrays, to provide a programming model in which programmers can control locality at multiple NUMA levels, by managing the movement of data between local memory, remote memory, and secondary storage.

3. Performance studies using synthetic and matrix multiplication benchmarks on the IBM SP and Intel Paragon, which demonstrate that the DRA library achieves very good I/O performance and efficiency on both machines.

In addition, we provide an initial report on the suitability of the DRA library for a large-scale computational chemistry program. This experiment suggests that its functionality is useful for a variety of applications.

In the rest of this paper, we describe the design, implementation, application, and performance of Disk Resident Arrays. In Section 2, we review Global Arrays. In Section 3, we introduce Disk Resident Arrays, in Section 4 we discuss implementation issues, and in Section 5 we describe DRA applications and performance on the Intel Paragon and IBM SP.

2 Global Arrays

The GA library provides collective functions for creating and destroying distributed arrays and for performing linear algebra operations such as matrix-matrix product or eigensolving on global arrays or sections of these arrays. However, its distinguishing and most important features are its emphasis on locality of reference, and its noncollective functions that allow any processor to transfer data between a local array and a rectangular patch of a global array in a shared memory style. Noncollective fetch, store, accumulate, gather, scatter, and atomic read-and-increment operations are supported. Global Arrays were designed to complement rather than replace the message-passing programming model. The programmer is free to use both the shared-memory and message-passing paradigms in the same program and to take advantage of existing message-passing software libraries.

We use a simple example to illustrate the GA model. The following code fragment (written using the Fortran 77 language binding) uses the Fortran interface to create a distributed double-precision array of size $n \times m$, blocked in chunks of size at least 10×5. This array is then zeroed and patch filled from a local array. The arguments to the `ga_create` call are the datatype, dimensions, a name, distribution directives, and the new array handle (output). The arguments to the `ga_put` call are a global array handle, the bounds of the patch within the global array into which data is to be put, the local array containing the data to be put, and the lower dimension of the local array.

```
integer g_a, n, m, ilo, ihi, jlo, jhi, ldim
double precision local(1:ldim,*)
call ga_create(MT_DBL, n, m,'A', 10, 5, g_a)
call ga_zero(g_a)
call ga_put(g_a,ilo,ihi,jlo,jhi,local,ldim)
```

We note that this code is similar in functionality to the following High Performance Fortran (HPF) [1] statements:

```
   integer n, m, ilo, ihi, jlo, jhi, ldim
   double precision a(n,m),local(1:ldim,*)
!HPF$ distribute a(block(10),block(5))
   a = 0.0
   a(ilo:ihi,jlo:jhi) = local(1:ihi-ilo+1,
                              1:jhi-jlo+1)
```

The difference is that this single HPF assignment would be executed in data-parallel fashion, while the GA put operation is executed in a MIMD mode that permits each processor to reference different array patches.

3 Disk Resident Arrays

The Disk Resident Arrays (DRA) library extends the GA NUMA programming model to disk (Figure 1). The DRA library encapsulates the details of data layout, addressing and I/O transfer on a new class of objects called disk resident arrays. Disk resident arrays resemble global arrays except that they reside on disk instead of in main memory. Operations similar to the GA data transfer functions support the movement of data between global arrays and disk resident arrays. DRA read and write operations can be applied both to entire arrays and to sections of arrays (disk and/or global arrays); in either case, they are collective and asynchronous.

The focus on collective operations within the DRA library is justified as follows. Disk resident arrays and global arrays are both large, collectively created objects. Transfers between two such objects seem to call for collective, cooperative decisions. (In effect, we are paging global memory.) We note that the same model has proved successful in the GA library: operations that move data between two global memory locations are collective. To date, application programmers have not expressed a strong desire for noncollective operations. If demand develops, we will certainly consider providing some form of data transfer between local and DRA memory. However, we note that because we would be jumping between nonadjacent levels in the NUMA hierarchy, performance may be disappointing, and portable implementations problematic.

The DRA library distinguishes between temporary and persistent disk resident arrays. Persistent disk resident arrays are retained after program termination and are then available to other programs; temporary arrays are not. Persistent disk resident arrays must be organized so as to permit access by programs executing on an arbitrary number of processors; temporary disk resident arrays do not need to be organized in this fashion, which can sometimes improve performance. Currently, we make this distinction by specifying that disk resident arrays remain persistent unless the user deletes them with a `dra_delete` function. We create all arrays in volatile temporary space and move nondeleted arrays to a more accessible location only when `dra_terminate` is called to shut down the DRA library. This approach optimizes performance for arrays that are accessed many times.

We use a simple example to illustrate the use of the DRA library. The following code fragment creates a disk resident array and then writes a section of a previously created global array (`g_a`) to the larger disk resident array (see Figure 2).The `dra_init` function initializes the DRA library and allows the programmer to pass information about how the library may be used and the system on which the program is executing. This information may be used to optimize performance.

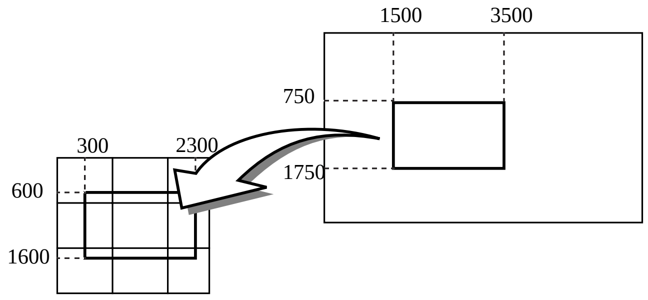

Figure 2: Writing a section of global array to disk resident array

```
#include 'dra.fh'
#include 'global.fh'
rc = dra_init(max_arrays, max_array_size,
                total_disk_space, max_memory)
rc = dra_create(MT_DBL,1000,2000,'Big Array',
                'bigfile',DRA_W,-1,1,d_a)
rc = dra_write_section(.false.,g_a,1,100,1,
                200,d_a, 201, 300, 201, 400,
                request)
rc = dra_wait(request)
rc = dra_close(d_a)
rc = dra_terminate()
```

The `dra_create` function creates a two-dimensional disk resident array. It takes as arguments its type (integer or double precision); its size, expressed in rows and columns (1000×2000); its name; the name of the “meta-file” used to represent it in the file system; mode information (`DRA_W` indicating that the new disk resident array is to be opened for writing); hints indicating the dimensions for a “typical” write request (-1 indicates unspecified); and finally the DRA handle (`d_a`). The function, like all DRA functions, returns a status value indicating whether the call succeeded.

The `dra_write_section` function writes a section of a two-dimensional global array to a section of a two-dimensional disk resident array. In brief, it performs the assignment:

```
d_big[201:300, 201:400] = g_a[1:100, 1:200]
```

where `d_big` and `g_a` are a disk resident array and global array handle, respectively. The first argument is a transpose flag, indicating whether the global array section should be transposed before writing (here, `.false.`). The operation is asynchronous, returning a request handle (`request`); the subsequent `dra_wait` call blocks until termination.

4 Implementation

The GA and DRA libraries have been implemented on a number of systems, including distributed-memory computers (Intel Paragon, IBM SP, Cray T3D), shared-memory computers (KSR-2, SGI PowerChallenge), and networks of

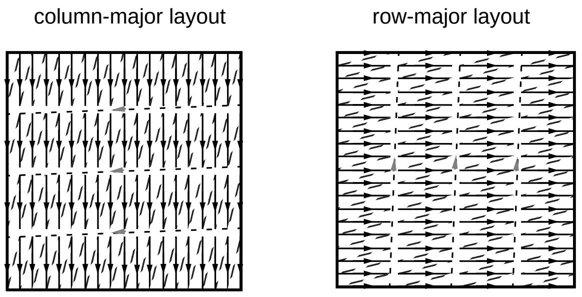

Figure 3: Two types of data layout on the disk in DRA library

workstations. Disk Resident Arrays are implemented in different ways on different systems. In some cases, a single disk resident array is stored as a single file in an explicitly parallel I/O system (PIOFS on the IBM SP, PFS on the Intel Paragon) or in a sequential filesystem striped implicitly across multiple disks (for example, Cray T3D, SGI); in other cases, a single disk resident array is stored as multiple disk files (for example, local disks available on every node of the IBM SP), each containing part of the disk resident array. To simplify file access in the latter case, we are developing a set of UNIX commands to copy, rename, move, delete, and change access attributes of DRA files stored as multiple files.

Implementation efforts have focused on two related questions. First, how should disk resident arrays be laid out on disk to optimize expected data transfers? Second, what hints can a programmer provide to guide the optimization process? In general, we assume that DRA functions will be used for large data transfers. This is certainly the case for all applications that we have considered to date. Hence, programmer-supplied hints can focus on characteristics of the memory system and expected I/O patterns. To support applications that explicitly manage consumption of local main memory, we allow user control over the amount of internal buffer space in the library. The library determines data layout for each array, depending on the amount of memory the application is willing to sacrifice for DRA buffering, the array dimensions, the dimensions of a "typical" request section, and characteristics of the underlying file system and I/O system. Either row-major or column-major storage may be selected, see Figure 3, and stripe width is chosen to maximize the size of the corresponding disk I/O transfers. The number of concurrent I/O requests that the given filesystem can support without a significant loss of efficiency is also taken into account. The logical data layout in Figure 3 is used directly in implementations that store a disk resident array in a single file. In implementations that target collections of independent disks and files, an additional level of striping is applied, with the result that data is effectively stored in two-dimensional chunks. A similar approach is adopted in Panda [7]. Each chunk contains data stored in consecutive disk blocks. Chunks are sized so as to fit in the DRA buffer; hence, they can be read or written in a single I/O request. This size is typically much larger than the default stripe factor used in parallel file systems such as PFS or PIOFS. We have not adopted fixed two-dimensional chunking for disk resident arrays that are stored in a single file, because the scheduling of I/O operations that can be obtained with this configuration is less flexible than in our approach, and might lead to more I/O operations that transfer smaller amounts of data when the requests are not perfectly aligned with the chunk layout.

In selecting the data layout for a disk resident array, we attempt to distribute data so that typical read and write operations are "aligned." An operation is said to be aligned if the corresponding section of the disk resident array can be decomposed into subsections that can be read (or written) as a single consecutive sequence of bytes. Otherwise, it is nonaligned. A disk resident array section specified by the user in a DRA read or write operation is decomposed into blocks according to the data layout and available buffer space in the DRA library. If the requested section is nonaligned with respect to the layout, the library augments the disk resident array section so that the low-level I/O operations always involve aligned blocks of data on the disk. For read operations, aligned blocks of data are staged to buffer memory, and only the requested (sub)sections are transferred to global memory. Write operations to nonaligned array sections are handled by first performing an aligned read to transfer an augmented, aligned block to the buffer, and then overwriting the relevant portion before writing the augmented block back to disk. These techniques for handling nonaligned requests are evocative of cache memory management mechanisms; similar techniques have been used in PASSION [8]. For example, in Figure 4, we show an array distributed using a column-major layout with a stripe width of 10 KB and consider two read operations (the shaded blocks). The upper block has size 2500 8-byte words (20 KB) in the column dimension, and so is aligned: it can be read without transferring unnecessary data. In contrast, the lower block has size 1500 words (12 KB) in the column dimension, and so is nonaligned. The block is internally augmented to 20 KB for reading, and the low-level I/O operations must transfer 20/12 ≈ 1.67 times more data.

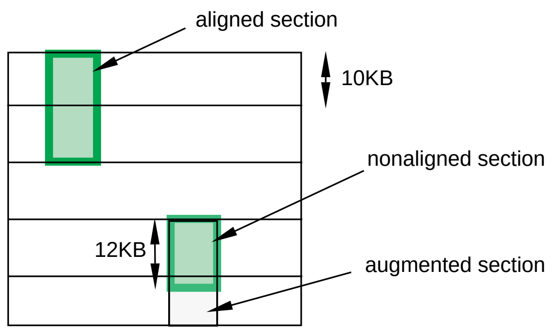

Figure 4: Example of aligned and nonaligned sections in column layout

The DRA library is implemented in terms of low-level I/O primitives that hide the specifics of different I/O systems. The low-level interface adopts some of the principles outlined by Gibson et al. [9]. In particular, file pointers are not supported (we support just a byte-addressable disk memory, rather than seek functions), and asynchronous forms are provided for all I/O requests.

The low-level interface provides functions to create, open, and delete files corresponding to a disk resident array, and to read and write to a specified location in files. These primitives are used to transfer data between disk resident arrays and the internal buffers maintained in the memory of the processes that can perform I/O. Data is then transferred between the buffer and global memory by using the `put` and `get` operations provided by the GA library. Message-passing mechanisms are not used in the implementation at all, because GA provides the necessary communication operations portably across all target platforms. This use of GA array-oriented communication functions significantly simplifies the DRA implementation. Furthermore, as the bandwidth of the GA operations is close to that of native message-passing libraries [2], and the bandwidth of the in-core (local and global memory) operations is much higher than the available I/O bandwidth, the use of GA functions does not significantly impact performance.

To date, we have implemented asynchronous I/O using either native asynchronous I/O (on the Paragon under PFS) or Posix asynchronous I/O (version 1003.4a or earlier; available on KSR, SGI, RS/6000 and DEC Alpha). On systems that provide neither of these mechanisms, only synchronous I/O is supported. We are currently investigating the use of lightweight threads in order to support multiple outstanding asynchronous requests; the Nexus multithreaded library [10] may be used for this purpose. Unfortunately, while threads provide a convenient notation for specifying asynchronous requests, available scheduling mechanisms do not always provide the degree of control required for efficient I/O. One solution to this problem is to introduce a polling mechanism designed to yield control to schedulable DRA threads, hence providing the DRA library with an opportunity to verify status of outstanding requests or issue new I/O requests as buffer space becomes available. The polling mechanism can be invoked explicitly by the user (via a new function `dra_flick`) or implicitly in the GA library synchronization routine.

5 Performance and Applications

We use three examples to illustrate the capabilities and evaluate the performance of the DRA library: a synthetic I/O benchmark, an out-of-core matrix multiplication program, and a computational chemistry program.

We report results obtained on the Intel Paragon at CalTech and the IBM SP at Argonne. The Paragon has 533 mesh-connected Intel i860/XP processors, each with 32 MB of memory; 21 of these processors are dedicated to I/O and are connected to 16 4.8 GB RAID-3 disk resident arrays. The Paragon runs the OSF/1 operating system and uses Intel's Parallel File System (PFS) to provide parallel access to files. PFS stripes files across the I/O nodes, the default stripe factor being 64 KB. We note that while PFS provides reasonable performance, it would appear that alternative file system organizations could provide better performance in some situations (e.g., see [11]). The Argonne IBM SP has 128 Power 1 processors connected by a multistage crossbar; 8 of these have an attached disk drive and are used for both computation and I/O. The SP runs the AIX operating system and uses IBM's Parallel I/O File System (PIOFS) to provide parallel access to files striped across the 8 I/O nodes.

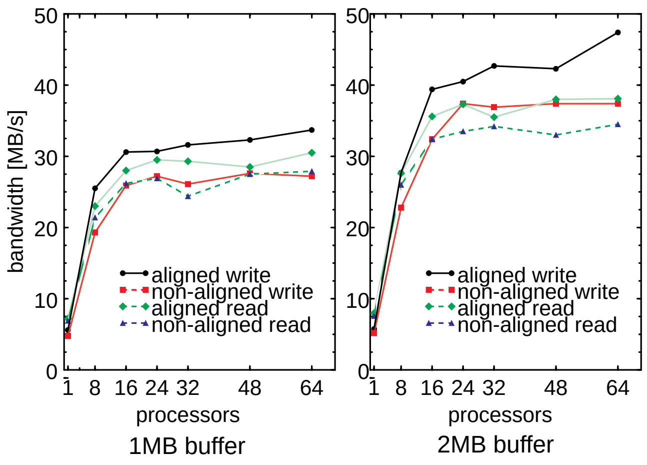

Figure 5: Performance of DRA I/O operations on Paragon as a function of the number application processors that perform I/O

5.1 Low-level Performance Studies

The first program that we consider is a synthetic benchmark designed to characterize the read and write performance of the DRA library. This program performs reads and write operations involving a 5000×5000 double-precision global array and a 10000×10000 disk resident array. Transfers to both aligned and nonaligned sections of the disk resident array are tested. The disk resident array is closed and opened between consecutive DRA write and read operations to flush system buffers. We timed a series of calls to obtain average times.

Figure 5 shows measured transfer rates on the Paragon. The DRA library stores the disk resident array data in one PFS file opened in `M_ASYNC` mode. We varied both the size of the internal DRA buffer and the number of application processes that perform I/O. Aligned write operations appear to execute faster than the corresponding read operations when more than one I/O processor is used. In the nonaligned cases, the disk resident array section involved in the DRA I/O operation was decomposed into aligned and nonaligned subsections. When using DRA buffers of 1 MB and 2 MB, there were 20 and 40 nonaligned subsections and 90 and 180 aligned subsections, respectively. The difference in DRA I/O performance for both aligned and nonaligned I/O transfers is consistent with the implementation scheme described in Section 4. However, when using the 2 MB buffer with 24 or

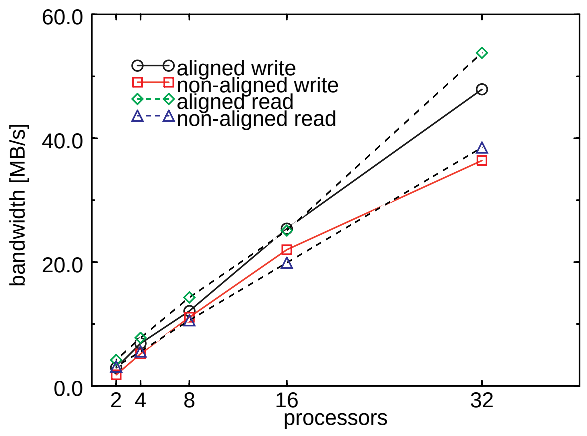

Figure 6: Performance of DRA I/O operations on the IBM SP -- local disks implementation

more I/O processors, nonaligned writes execute faster than corresponding nonaligned reads. We attribute this result to the fact that with a 2 MB buffer, we are able to generate a sufficient number of concurrent read and write requests; in addition, write operations are faster than reads on the Paragon. In general, increasing the size of the DRA internal buffer results in larger PFS I/O requests and in higher achieved bandwidth.

The performance achieved by the DRA library is 42.7 MB/s, for an aligned write from 32 processors with 2 MB internal buffer. (A nonaligned write achieves 36.9 MB/s, and aligned and nonaligned reads achieve 35.5 and 34.2 MB/s, respectively.) Our results are consistent with I/O rates reported by other researchers using tuned hand-coded applications on the same computer. For example, Miller et al. [12] report peak read rates of 35.2 MB/s for a 94 MB transfers and 37.8 MB/s for 377 MB read transfers [12]; we get 35.5 MB/s for a 200 MB transfer. Thakur et al. [13] report a write bandwidth of 10.85 MB/s from 32 Paragon processors when using the Chameleon I/O library in an astrophysics application. Chameleon uses a fixed internal buffer size of 1MB.

On the IBM SP, we obtained peak performance results of 33.4 MB/s for aligned writes, 31.4 MB/s for nonaligned writes, 44.0 MB/s for aligned reads, and 42.1 MB/s for nonaligned reads from 16 processors. These also appear to be competitive with hand-coded I/O programs. Performance results for the implementation based on a collection of local disks (such disks are available on every node of the IBM SP), are depicted in Figure 6, and demonstrate almost linear scaling with the number of processors/disks used (bandwidth in AIX read/write operations to local disk on the Argonne SP system is less than 2MB/s). When taking these measurements, we made a special effort to avoid perturbations due to caching of file data in memory, something that the IBM AIX operating system is very good at. In particular, we used separate programs to test read and write DRA operations, and in between runs of these two programs ran another unrelated program that touches all physical memory on each node. This strategy ensures that all previously cached data is paged out. While we cannot guarantee that performance of write operations was unaffected by caching, we are convinced that the DRA reads fetched data from the disks.

We draw three conclusions from these results. First, the DRA library is able to achieve high I/O rates on the Intel Paragon and IBM SP. Second, internal buffer size makes a significant difference to performance, with a size of 2 MB allowing us to achieve close to peak. Third, aligned I/O operations perform better than nonaligned operations.

5.2 Out-of-Core Matrix Multiplication

Our second example is an out-of-core matrix multiplication program written by using DRA mechanisms. This program multiplies two arrays that reside on disk and writes the result to the third array, also stored on disk. The three matrices are assumed not to fit in the main memory, and so the program pages smaller matrix sections in and out of memory by using DRA mechanisms, operating on pairs of blocks using an in-core parallel matrix multiplication algorithm.

The program uses a block-oriented algorithm to multiply the two matrices, fetching one $n \times n_b$ section of matrix *B* at a time to multiply with an $n_b \times n$ section of matrix *A* (Figure 7). The next section of matrix *B* is prefetched, to overlap the expensive I/O operations with computation. Since an $n_b \times n_b$ section of matrix *C* is assumed to be much smaller than a $n \times n_b$ section of matrix *B*, and the current implementation of DRA library can effectively perform asynchronous I/O for only one request at a time, we prefetch only matrix *B*. Prefetching increases memory requirements by around 50 percent, and hence reduces the memory available for the matrix sections that are being multiplied in core while I/O is taking place. Since performance of the in-core matrix multiplication increases with matrix size [14], prefetching can potentially reduce the performance of the in-core computations.

The in-core matrix multiplication is performed by using the Scalable Universal Matrix Multiplication Algorithm (SUMMA) [14]. We chose this particular algorithm for several reasons. Not only does it appear to be the most efficient parallel algorithm available, but it requires little workspace. It was also simpler to implement an out-of-core algorithm based on SUMMA rather than on some other matrix-multiplication algorithms, in part because the distributed array

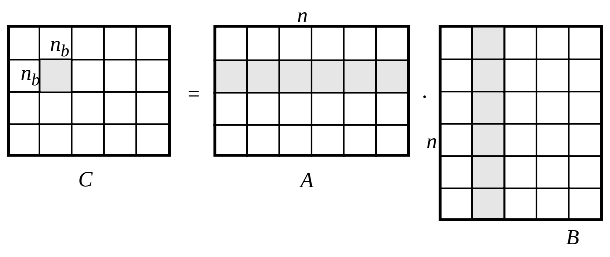

Figure 7: Blocking in the out-of core matrix multiplication

layout that it requires can be represented by the GA library without additional data rearrangements.

The implementation of the out-of-core program passes the handles of the three disk resident arrays to the matrix multiplication routine as arguments. Four global arrays are created to store sections of these disk resident arrays: one chunk of *A*, two chunks of *B* (because of the prefetching), and one chunk of *C*. The following Fortran pseudo-code presents the algorithm.

```
prefetch asynchronously $B_{chunk}$(next)
do i = 1, $chunks_i$
  read $A_{chunk}$
  do j = 1, $chunks_j$
    swap next and current
    wait until $B_{chunk}$(current) is available
    prefetch asynchronously $B_{chunk}$(next)
    call multiply($A_{chunk}$,$B_{chunk}$(current),$C_{chunk}$)
    write $C_{chunk}$
  enddo
enddo
```

The I/O associated with the prefetching of blocks of matrix *B* can be fully overlapped with computation if sufficient I/O bandwidth, $B_{I/O}$ measured in MB/s, is available

$$B_{\mathrm{I/O}} = \frac{4PM}{n_b}, \qquad (1)$$

where *M* is the processor Mflop/sec rate in the in-core matrix multiplication, *P* is the number of computational processors, and we assume 8-byte representation for double-precision numbers. This model can be used to select the block size for the out-of-core algorithm.

We found that the array-oriented high-level functionality provided by the DRA and GA libraries greatly minimized programming effort. The entire out-of-core algorithm was expressed in about 140 lines of Fortran 77 code. This result provides an interesting data point for discussions regarding the advantages of compiler support for out-of-core arrays [15].

We measured performance of the out-of-core matrix multiplication program on the Intel Paragon. We first created, initialized, and stored in disk resident arrays two double-precision input matrices of size 10000×10000. A third disk resident array was created to store the result. We then measured the time required to multiple the two input matrices when using 64, 128 and 256 processors. For the 64- and 128-processor runs, we used n_b = 2000, while for the 256-processor run n_b = 2500. We evaluated both the in-core matrix-multiply component of the program, and the full out-of-core algorithm, including the time to read two arrays from disk and write the result back to the disk. Figure 8 shows both the achieved performance in Mflop/sec, and the relative efficiency of the out-of-core algorithm compared with its in-core component.

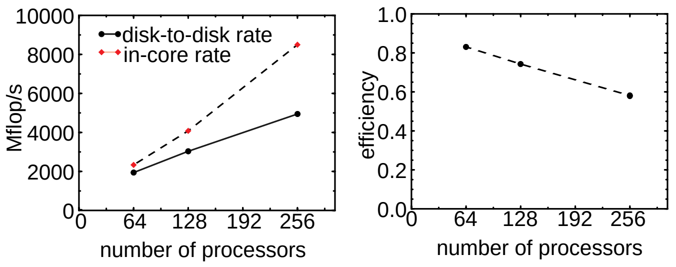

Figure 8: Mflop/s rates and efficiency of the out-of-core algorithm relative to performance of its in-core component

We see that efficiency degrades as the number of processors increases. Because blocks of *A* and *C* matrices are not prefetched, the I/O transfer rate remains approximately constant while the floating-point performance increases with processor count. We observe good overall performance on 64 and 128 processors. For 256 processors, the matrix becomes too small, and I/O costs become significant. In fact, on 256 processors the computation could be performed in core. We include this case only to demonstrate the scaling characteristics of the algorithm.

5.3 RI-SCF: A Chemistry Application

Our third example application is a large computational chemistry application. Electronic structure computation is a field of computational chemistry that is concerned with determining electronic structure of molecules and other small or crystalline chemical systems [16]. These calculations can predict many chemical properties that are not directly accessible by experiments; they consume a large proportion of the total number of supercomputer cycles used for computational chemistry.

All electronic structure methods compute approximate solutions to the non-relativistic electronic Schroedinger equation. The iterative Self-Consistent Field (SCF) algorithm is the simplest method used for this purpose. The kernel of the SCF calculation is the contraction of a large, sparse four-index matrix (electron-repulsion integrals) with a two-index matrix (the electronic density) to yield another two-index matrix (the Fock matrix). The irregular sparsity and available symmetries of the integrals drive the calculation. Both matrices are of size *N*, where *N* is the size of an underlying basis set. (Typically, *N*=1000–5000). The number of integrals scales between $O(N^2)$ and $O(N^4)$ depending on the nature of the system and level of accuracy required. In conventional approaches, integrals were calculated once and stored on the disk to be reused in every SCF calculation. On massively parallel systems, so-called direct methods avoid the bottleneck associated with this I/O [17] by computing integrals in-core as needed.

In order to reduce the high-cost of the SCF calculations for larger molecules and/or large basis sets, the Resolution of the Identity (RI) method computes an approximate SCF energy with four-index integrals being expanded in terms of the three-index quantities. The computational effort for the Coulomb contribution scales as $O(N^3)$ with a moderate prefactor. Früchtl et al. [18] have used the GA and DRA libraries to develop an RI-SCF implementation. They use disk resident arrays to store transformed three-index compressed integrals. The amount of available main memory limits the number of expansion functions contributing to the approximate solution that can be computed in-core at a time. Processing is done in batches, with integrals for the next expansion function being read asynchronously while the Fock matrix contributions are being calculated. The main components of the calculations are two matrix multiplications which require $4N^3$ floating-point operations per expansion function. The I/O access patterns exhibited by this application are as follows. First, a disk resident array is written once in multiples of full rows at a time. Then, multiple full columns are read many times, based on the number of expansion functions held in core and the number of SCF iterations. We received positive feedback from the developers concerning the DRA functionality and performance. For details we refer the reader to [18].

RI-MP2 is another large parallel computational chemistry application with significant I/O requirements. This code was developed on top of Global Arrays and simple Fortran I/O [19] before the DRA library was available, and is currently being converted to use DRA. In both RI-SCF and RI-MP2, the DRA abstractions has been found to simplify implementation of out-of-core algorithms. In addition, its asynchronous interface has contributed to reducing the I/O bottleneck by providing the ability to overlap I/O with computation.

6 Related Work

Although out-of-core algorithms date to the early days of computing [20], they have traditionally been developed without assistance from libraries or compilers. Virtual memory can also be used to manage data structures too large to fit in memory. However, the complexity of I/O systems on modern parallel computers and the performance requirements of scientific applications have so far prevented this approach from being effective. Application-specific virtual memory management [21] may address some of these concerns, but it has not yet been proven in large-scale systems.

Parallel I/O techniques and libraries that provide collective I/O operations on arrays include disk-directed I/O [22], Panda [23], PASSION [8], MPI-IO [24], and Vesta [25]. Most support lower-level interfaces than the DRA library, allowing, for example, non-collective operations or requiring that the user define disk striping. Our work is distinguished by the integration of the DRA and GA libraries, which means that data is transferred collectively from disk to global or shared memory, rather than to private processor memory. One important consequence of this model is that data transfers typically are large, and the implementation can optimize for large transfers. We prefer this approach to more flexible models that allow for independent I/O from different processors. I/O is expensive and should be done in large blocks, and our API is designed for large block transfers. In addition, the asynchronous interface provided to I/O operations encourages users to design I/O algorithms using prefetching or a two-phase approach.

The DONIO library [26] caches a copy of a file in memory, distributed across available processors. Application processors can then seek and perform non-collective read/write operations on this file, as if they were accessing disk. When the “shared file” is closed or flushed, the data is moved to disk. This model combines some aspects of the GA and DRA libraries, but provides less control over the movement of data between different levels of the storage hierarchy.

Vitter and Shriver [27] and Shriver and Wisniewski [28] describe out-of-core matrix multiplication algorithms but do not present performance results.

Compiler-based approaches to out-of-core I/O have been investigated in the context of the data-parallel languages C* [29] and High Performance Fortran [15,30]. However, these languages do not support the MIMD programming model provided by the GA library and apparently required by the computational chemistry programs for which the GA and DRA libraries were originally designed.

7 Conclusions

We have described a high-performance parallel I/O library that provides programmers with explicit control over the movement of data between different levels in the memory hierarchy. The Disk Resident Arrays (DRA) library described in this paper manages the movement of data between secondary storage and distributed data structures called global arrays; its integration with the Global Arrays (GA) library, which manages the movement of data between global arrays and local memories, allows programmers to stage array data structures from disk, through global memory, to local memory.

We believe that the simple conceptual model provided by the DRA library can significantly simplify the task of developing efficient, portable out-of-core programs. To date, this assertion is supported by positive experiences with two applications, an out-of-core matrix multiplication program and a large computational chemistry program. Further applications will be evaluated in the future. We note that the integration of the DRA library with the GA library’s array-oriented communication infrastructure not only simplifies the development of out-of-core applications, but also significantly reduced DRA development time. Focus on large data transfers allowed some performance optimizations that would not be possible in a general-purpose I/O library.

We are currently working on a variety of performance optimizations. One important direction relates to increasing the degree of asynchronism in DRA operations. Currently, asynchronous I/O is achieved by using asynchronous I/O mechanisms in the underlying file systems. Threads may well be provide a more portable and flexible alternative, although their performance characteristics are not yet well understood. The simplicity of the DRA framework means that it can also be extended easily in other directions. For example, the matrices constructed by computational chemistry codes are often quite sparse and hence may benefit from the use of compression [7]. Compression is easily incorporated into the DRA framework, as are other optimizations proposed for collective I/O [31,8,11].

Acknowledgments

Work at PNNL was performed under the auspices of the High Performance Computing and Communications Program of the Office of Computational and Technology Research, U.S. Department of Energy under contract DE-AC066-76RLO1830 with Battelle Memorial Institute which operates the Pacific Northwest National Laboratory. Work at ANL was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38, and by the Scalable I/O Initiative, a multi-agency project funded by ARPA, DOE, NASA, and NSF.

References

1. High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, Houston, TX, January 1993.
2. J. Nieplocha, R. Harrison, and R. Littlefield. Global Arrays: A portable "shared-memory" programming model for distributed memory computers. In *Proc. Supercomputing 1994*, pages 340–349. IEEE Computer Society Press, 1994.
3. J. Nieplocha, R. Harrison, and R. Littlefield. Global Arrays: A non-uniform-memory-access programming model for high-performance computers. *The Journal of Supercomputing*, to appear.
4. B. Bershad, M. Zekauskas, and W. Sawdon. The Midway distributed shared memory system. In *Proc. '93 CompCon Conference*, pages 528–537, 1993.
5. N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.
6. D. Bernholdt et al. Parallel computational chemistry made easier: The development of NWChem. *Intl J. Quantum Chem. Symp.*, 29:475–483, 1995.
7. K. E. Seamons and M. Winslett. A data management approach for handling large compressed arrays in high performance computing. In *Proc. 5th Symposium on the Frontiers of Massively Parallel Computation*, pages 119–128, February 1995.
8. R. Bordawekar, A. Choudhary, and R. Thakur. Data access reorganizations in compiling out-of-core data parallel programs on distributed memory machines. Technical Report SCCS-622, NPAC, Syracuse, NY 13244, September 1994.
9. G. Gibson and D. Stodolsky. Issues arising in the SIO-OS low-level PFS API. Technical report, Carnegie Mellon University, 1993.
10. I. Foster, C. Kesselman, and S. Tuecke. The Nexus task-parallel runtime system. In *Proc. 1st Intl Workshop on Parallel Processing*, pages 457–462. Tata McGraw Hill, 1994.
11. C. Freedman, J. Burger, and D. DeWitt. SPIFFI – a scalable parallel file system for the Intel Paragon. *IEEE Trans. Parallel and Dist. Systems*, 1996. to appear.
12. C. Miller, D. Payne, T. Phung, H. Siegel, and R. Williams. Parallel processing of spaceborne imaging radar data. In *Proc. Supercomputing '95*. ACM Press, 1995.
13. R. Thakur, W. Lusk, and W. Gropp. I/O characterization of a portable astrophysics application on the IBM SP and Intel Paragon. Preprint mcs-p534-0895, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1995.
14. R. van de Geijn and J. Watts. Summa: Scalable universal matrix multiplication algorithm. Technical Report TR-95-13, Dept of Computer Sciences, The University of Texas, 1995.
15. R. Bordawekar, A. Choudhary, K. Kennedy, C. Koelbel, and M. Paleczny. A model and compilation strategy for out-of-core data parallel programs. In *Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–10, July 1995. Also available as the following technical reports: NPAC Technical Report SCCS-0696, CRPC Technical Report CRPC-TR94507-S, SIO Technical Report CACR SIO-104.
16. R. Harrison and R. Shepard. Ab initio molecular electronic structure on parallel computers. *Ann. Rev. Phys. Chem.*, (45):623–58, 1994.
17. P. Crandall, R. Aydt, A. Chien, and D. Reed. Input/Output characteristics of scalable parallel applications. In *Proc. Supercomputing '95*. ACM Press, 1995.
18. H. Früchtl, R. A. Kendall, R. J. Harrison, and K. G. Dyall. A scalable implementation of RI-SCF on parallel computers. Submitted to *Intl J. Quantum Chem.*, 1996.
19. D. E. Bernholdt and R. J. Harrison. Large-scale correlated electronic structure calculations: The RI-MP2 method on parallel computers. *Chem. Phys. Lett.*, 250:477–484, March 1996.
20. R. Floyd. Permuting information in idealized two-level storage. In R. Miller and J. Thatcher, editors, *Complexity of Computer Calculations*, pages 105–109. Plenum Press, 1972.
21. K. Krueger, D. Loftesness, A. Vahdat, and T. Anderson. Tools for the development of application-specific virtual memory management. In *Proc. 8th Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 48–64, 1993. SIGPLAN Notices 28(10).
22. D. Kotz. Disk-directed I/O for an out-of-core computation. In *Proc. 4th IEEE International Symposium on High Performance Distributed Computing*, pages 159–166, August 1995.
23. K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proc. Supercomputing '95*, December 1995. To appear.
24. P. Corbett, D. Feitelson, S. Fineberg, Y. Hsu, W. Nitzberg, J.-P. Prost, M. Snir, B. Traversat, and P. Wong. Overview of the MPI-IO parallel I/O interface. In *IPPS '95 Workshop on Input/Output in Parallel and Distributed Systems*, pages 1–15, April 1995.
25. P. Corbett, D. Feitelson, J.-P. Prost, and S. Baylor. Parallel access to files in the Vesta file system. In *Proc. Supercomputing '93*, pages 472–481, 1993.
26. E. F. D'Azevedo and C. Romine. DONIO: Distributed object network I/O library. Report ornl/tm-12743, Oak Ridge National Lab., 1994.
27. J. Vitter and E. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2/3):110–147, August and September 1994.
28. E. Shriver and L. Wisniewski. An API for choreographing disk accesses. Technical Report PCS-TR95-267, Dartmouth College, 1995.
29. T. H. Cormen and A. Colvin. ViC*: A preprocessor for virtual-memory C*. Technical Report PCS-TR94-243, Dept. of Computer Science, Dartmouth College, November 1994.
30. R. Thakur, R. Bordawekar, and A. Choudhary. Compilation of out-of-core data parallel programs for distributed memory machines. In *IPPS '94 Workshop on Input/Output in Parallel Computer Systems*, pages 54–72. Syracuse University, April 1994. Also appeared in Computer Architecture News 22(4).
31. D. Kotz. Disk-directed I/O for MIMD multiprocessors. Submitted to TOCS, January 1995.