Passion: Optimized I/O for Parallel Applications

Rajeev Thakur
Argonne National Laboratory

Alok Choudhary
Rajesh Bordawekar
Sachin More
Silvaramkrishna Kuditipudi
Syracuse University

Parallel computers with peak performance of more than 100 GFlops/second are already available to solve a variety of problems in a range of disciplines. However, the input/output performance of these machines is a poor reflection of their true computational power. This bottleneck is becoming increasingly significant because scientists and researchers are now using parallel computers for applications with huge data sets, many of which require the processing of distributed multidimensional arrays. For example, to detect faint radio pulsars (one of many scientific Grand Challenge applications), a computer must perform fast Fourier transforms of 8 to 64 gigapoints, which requires an I/O throughput of at least 1 Gbyte/second. Currently, the achievable I/O throughput is typically less than 100 Mbytes/second.

To improve the I/O performance of parallel programs with distributed multidimensional arrays, we have developed a software library called Passion (Parallel, Scalable Software for Input/Output). Passion's routines are designed to read or write either entire distributed arrays or sections of such arrays. Passion also frees the programmer from many of the tedious tasks associated with performing I/O in parallel programs and has a high-level interface that makes it easy to specify the required I/O. Because Passion accepts high-level requests from the application and translates them to the low-level interface supported by the machine's parallel file system, the Passion interface can be ported to a variety of machines. We have also incorporated a number of I/O performance optimizations, including collective I/O (in which processors cooperate to perform I/O in large chunks and in the proper order), data prefetching, data sieving, and data reuse. The "Parallel I/O Infrastructure" sidebar on p. 72 describes some of the I/O performance problems Passion is designed to address.

We have implemented Passion on Intel's Paragon, Touchstone Delta, and iPSC/860 systems, and on the IBM SP system. We have also made it publicly available through the World Wide Web (http://www.cat.syr.edu/passion.html). We are in the process of porting the library to other machines and extending its functionality.

PASSION ARCHITECTURE
Passion supports a loosely synchronous single-program, multiple-data computational model. As Figure 1 shows, its interface is a level higher than the interface to a parallel file system. The user simply specifies what section of the array must be read in terms of its lower bound, upper bound, and stride in each dimension, and Passion fetches it. In contrast, a typical interface to a parallel file system requires the user to calculate offsets in files explicitly and access contiguous portions of data.
We designed Passion to be used in one of two ways: Programmers can call Passion routines directly, as in Figure 1a, or a compiler can translate out-of-core programs written in a high-level data-parallel language (like High-Performance Fortran) to message-passing node programs with calls to Passion for I/O, as in Figure 1b.

Passion supports two basic models for storing and accessing data—the Local Placement Model and the Global Placement Model—and it provides routines to perform the required I/O on arrays and sections of arrays. In both the Local Placement Model and the Global Placement Model, Passion fetches data from a file, waits for the application program (processor) to perform the necessary computations on that data, and writes the results back to a file if necessary. This process repeats on other data sets until the end of the program.

**Local Placement Model**

Figure 2a shows how Passion works if the application program uses the Local Placement Model. In this model, the global array is divided into local arrays, each of which belongs to a particular processor, and each local array is stored in a separate file, the local array file. Using Passion routines, the node program explicitly reads from and writes to the file when required. The simplest way to view this model is to think of each processor as having another level of memory that is much slower than main memory. If the system's I/O architecture is such that each proces-

---

**Figure 1.** Passion may be used in one of two ways: (a) directly from a message-passing application program or (b) as an I/O library for a compiler for a high-level data-parallel language like High-Performance Fortran (HPF).

**Figure 2.** Models of data storage and access in Passion. (a) In the Local Placement Model, the global array is divided into local arrays stored in separate local array files for each processor. (b) In the Global Placement Model, the global array is stored in a single common global array file that all processors access.
Parallel I/O infrastructure

Almost all parallel computers provide some kind of hardware and software support for parallel I/O, usually in the form of a parallel I/O subsystem that comprises a set of I/O processors and a set of disks. Each I/O processor controls one or more disks. The I/O processors may be separate processors that do not run any compute jobs, or they may be compute processors that also perform as I/O processors. Data in each file is distributed across disks through a method known as declustering. In this method, the file is divided into a number of striped units, each of which is a logically contiguous portion of file data. The striped units are distributed among disks in a round-robin style and are physically contiguous on a disk. A striped unit can be as small as one bit or as large as the file block size.

The advantage of declustering is that if multiple processors request different parts of the file, the requests may lie on different disks and hence may be served in parallel. A number of parallel file systems have been developed that allow parallel access to files.

Although parallel computers are provided with hardware support for large-scale parallel I/O, the I/O performance observed at the application level is usually much lower than what the hardware can support. There are several reasons. First, the data access patterns of many parallel programs are such that they result in a large number of small requests to the file system. Because the I/O latency is very high, poor performance results. Second, none of the interfaces of current parallel file systems let programmers specify stored accesses using a single read or write call (although some recent proposals address this). Third, the interfaces do not provide support for processors to make a collective I/O request. Therefore, file systems cannot perform any optimizations on the basis of their collective knowledge of all processor access requests. Finally, programmers cannot specify access requests using a high-level description, but must instead explicitly manipulate file pointers. This makes it difficult for the programmer to optimize I/O without computation, because of the complexities involved in managing buffer sizes and file pointers.

Global Placement Model

In this model, shown in Figure 2b, the global array is stored in a single file, the global array file, and no local array files are created. Consequently, the model saves the phase in the Local Placement Model that creates the local array files. Of course, the Global Placement Model requires explicit synchronization when a processor must access data that another processor may have previously modified.

In the Global Placement Model, a processor generally needs access to the global array file either to retrieve a section of its own local array or to retrieve a section of the local array of another processor. The latter case corresponds to accessing off-processor data. Thus, a processor may need to access an arbitrary portion of the global array. The model provides routines to access sections of global arrays from files. Each processor specifies the section it needs in terms of its lower bound, upper bound, and stride in each dimension. The routines for accessing out-of-core array sections use the two-phase method for collective I/O.

References
- In this feature... Parallel I/O Systems and Interfaces for Parallel Computers (Birkhauser, Boston, 1993). ISBN 0-8176-3399-1
Two-phase method

In data-parallel programs, all processors perform similar operations but on different data sets. If one processor must read data from disks, it is highly likely that a group of processors, or maybe all the processors, will need to read data from disks at the same time. Hence, the requesting processors could cooperate in reading or writing data—a process known as collective I/O.

If processors perform I/O independently, many low-granularity requests may arrive from different processors in any order. If the processors use collective I/O, they can cooperate to perform I/O efficiently in large chunks and in the proper order.

Passion implements collective I/O using the two-phase method, a technique our group developed. To illustrate, suppose you want to read an entire two-dimensional array from a file into a distributed array in main memory. In distributed-memory parallel programs, two-dimensional arrays are commonly distributed among processors as row-block, column-block, block-block, row-cyclic, column-cyclic, and cyclic-cyclic. The file containing the array is typically striped across disks as determined by the striping unit. The “Parallel I/O infrastructure” sidebar explains striping units in more detail.

When data is distributed among processors in a way that conforms to the way it is stored on disks, each processor can read its local array directly in a single request. This distribution is called the conforming distribution. For example, if an array is stored in a file in column-major order, a column-block distribution is the conforming distribution. For any other distribution, Passion stores a processor's local array noncontiguously, and if each processor tries to read its local array directly (the direct method), the result will be many low-granularity requests. Hence, in the direct method, the I/O performance is best for the conforming distribution but degrades drastically for any other distribution.

Our method makes it possible to have a performance level very close to that of the conforming distribution, yet still use the desired distribution. Figure 3 illustrates. In the first phase of the method (bottom of the figure), the processors always read data assuming the conforming distribution, which requires the least I/O time. In the second phase, Passion redistributes data among processors to the desired distribution using interprocessor communication. Because interprocessor communication costs orders of magnitude less than I/O, the extra cost of the second phase is negligible. The procedure reverses for writing a distributed array to a file.

Out-of-core array sections

In the preceding description, we focused on accessing arrays that can fit in memory. Often a processor must access a section of a global array. For example, suppose each processor must read some regular section of a two-dimensional array stored in a file in column-major order. The section needed by each processor is represented in terms of its lower bound, upper bound, and stride in each dimension \( l, u, s_1, l, u, s_2 \), given in global coordinates.

To read these out-of-core array sections, Passion first assigns ownership to portions of the file such that a processor can directly access only the portion of the file it owns. The portion of the file a processor can directly access is called its file domain. For a file stored in column-major order, for example, the file domain is some set of array columns. Passion selects file domains dynamically according to the access requests, effectively partitioning the I/O workload among processors dynamically. This partitioning is logical, not physical.

Next, processors exchange their own access information (the indices \( l, u, s_1, l, u, s_2 \)) with other processors so that each processor knows the access requests of the others. Each processor then calculates which portions of its own request and the requests of other processors lie in its file domain. Figure 4 shows a file domain for an access pattern involving four processors: 0, 1, 2, and 3.

Each processor must then read all the required data
from its file domain. A simple way to do this would be to read all the data that processor 0 needs, followed by what processor 1 needs, processor 2 needs, processor 3 needs, and so on. But this method often results in too many small accesses that are not in sequence. For the processor to read the data efficiently, it must access the file in sequence and contiguously.

We use a very general method to ensure this procedure. Each processor calculates the minimum of the lower bounds and the maximum of the upper bounds of all sections requested from its file domain. This effectively determines the smallest section that contains all the data that must be read from the file domain (ABCD in Figure 4). Because this section may also include data that no processor requires, Passion uses an optimization called data sieving (described later) to read the useful data from the section. Thus, the processor can read large contiguous portions of data at a time from the file, extract the useful data, and discard the unwanted portion.

When Passion has read all the required data from the file domain, the first phase of the two-phase method is complete. The second phase consists of communicating the data read to the respective processors. The algorithm for writing sections is essentially the reverse of the reading algorithm.

**Performance**

We have tested the performance and scalability of this two-phase method extensively and found it to be far superior to the direct method for reading and writing both entire arrays and sections of out-of-core arrays. Figure 5 shows the performance improvement of the two-phase method over the direct method for reading an entire array (one that fits in memory) of 10K × 10K single-precision real numbers using 64 processors on the Touchstone Delta for four common distributions. The array is stored in column-major order, so the column-block distribution is the conforming distribution. Hence, for the column-block case, no redistribution is required, and the two-phase and direct methods are essentially the same. For any other distribution, the two-phase method performs considerably better. For the row-cyclic distribution, with the direct method, each processor must make a separate read request for each element of its local array, resulting in a very large number of low-granularity requests. This accounts for the extreme time the direct method takes for this distribution.

By contrast, the two-phase method takes nearly the same time for all distributions. The difference in time is due only to the difference in time for the redistribution phase, and the time required for redistribution is at least an order of magnitude lower than that required for I/O.

Figure 6 shows the time taken by the two methods on the Touchstone Delta, for reading sections of an out-of-core array of 16K × 16K single-precision real numbers. Each processor, \( p \), must read a section of this array, specified by \((1 + 20p: 16 + 20p: 1, 4000: 12,000: 1)\). The number of processors varies from 4 to 128. As the figure shows, the two-phase method performs considerably better than the direct method. The performance improves even more when there are more processors. When the number of processors doubles, the amount of I/O performed also doubles, but the time taken increases only slightly.

**EXAMPLE**

To illustrate how Passion works in a practical example, consider the High-Performance Fortran fragment in Figure 7, which solves Laplace's equation using a Jacobi iteration method. In High-Performance Fortran, programs use a single global address space, with directives to explicitly specify the parallelism. In this example, arrays A and B are distributed as block-block on a 4 × 4 grid of processors, shown in Figure 8a, but they are too large to fit in main memory. To execute them on a distributed-memory machine, a compiler must first translate the program to a node program with calls to Passion for I/O using either the Local Placement Model or the Global Placement Model.
Local Placement Model

Figure 8b shows the out-of-core local array on processor P5. This array is stored in the processor’s local array file, which is divided into slabs, as Figure 8d shows. Each slab represents the data that Passion can fetch at a time into processor memory for computation (“Actual data”), which is the in-core local array in Figure 8c. When the application program executes, Passion fetches a slab from the local array file to the in-core local array, waits as the processor performs the computation on that slab, and stores the results to the appropriate place in the local array file. This process repeats for each remaining slab.

The computation in this example is a stencil computation, in which each processor calculates the value of each element \((i,j)\) using the values of its corresponding four neighbors: \((i-1,j),(i+1,j),(i,j-1)\) and \((i,j+1)\). Also, the computation in the current iteration uses values computed in the previous iteration. Therefore, to calculate the values at the four boundaries of the local array, P5 needs the last row of the local array of P1, the last column of the local array of P4, the first row of the local array of P9, and the first column of the local array of P6. Before each iteration of the program, P5 must get these rows and columns.

If the array had been in core, Passion would have placed these rows and columns in the overlap areas shown in Figure 8b, because such an approach would enhance performance for this particular example. However, because the array is out-of-core, Passion stores these overlap areas in the local array file. Thus the local array file consists of the local array elements interspersed with overlap areas. Passion stores the local array in either row-major or column-major order, as specified in the out-of-core array descriptor. The in-core local array also needs overlap area for the same reason as the out-of-core local array.

At the end of each iteration, all the processors must exchange boundary data with their neighbors in each direction, a shift-type communication routine. For an in-core array, this routine consists of directly communicating data from each processor’s local memory. For an out-of-core array, such communication also requires I/O. In the latter case, each processor reads the boundary data from its local array file and communicates the data to its neighbors. The processor also receives the data sent by neighboring processors and stores it in appropriate locations in the local array file. Passion provides routines to perform the I/O required to read and write the slabs, as well as that for out-of-core communication.

Global Placement Model

If the program uses the Global Placement Model, each processor must access a section of its local array from the global array file along with the required overlap area. Passion routines for doing this use the two-phase method described earlier. An explicit out-of-core communication phase is not
needed in this case. However, the processors must explicitly synchronize before each iteration of the program.

**PERFORMANCE OPTIMIZATIONS**

In addition to two-phase collective I/O, Passion incorporates three techniques specifically to enhance I/O performance: data sieving, data prefetching, and data reuse.

**Data sieving**

At present, no interface to a parallel file system supports access to sections of arrays with strides. Consequently, reading the section \((l_1, u_1, s_1, l_2, u_2, s_2)\) of an out-of-core local array from the local array file using the direct method requires explicitly moving the file pointer to each element and reading it individually. This requires as many reads as the number of elements in the section, and because of high I/O latency, results in a very high cost. With one processor and one disk on the Touchstone Delta, for example, it takes 1,948.00 ms to read 1,024 integers individually; it takes 16.06 ms to read them as a block.

Passion uses data sieving to support the reading and writing of sections of strided arrays. To read a strided section, Passion reads a large contiguous portion of data, starting from the first element, into a temporary buffer in main memory using one read call. This, of course, may include unwanted data, so Passion extracts the required data from the buffer and discards the unwanted data dynamically. This process repeats until Passion has read the entire section. The amount of data read each time depends on the amount of memory available.

Data sieving is a way to combine several small I/O requests into fewer, larger requests to reduce the effect of high I/O latency time. The main advantage of this method is that it requires very few I/O calls; most of the time is spent transferring data within main memory. The disadvantage is that it requires extra memory and reads more data from disk than is actually required. However, we have found that the performance increases from the reduced number of I/O calls far outweigh the penalty of reading and extracting extra data.

In the direct method, even though a separate read call is required for each element, the call may not result in a disk access each time. The processor usually does some form of caching at the I/O nodes, and if the requested data lies in the cache, the processor can read it from the cache itself. Even so, we find that reading individual elements is a lot more expensive than reading one large chunk because the processor must make several requests to the I/O nodes and look up the software cache at the I/O node each time. Data sieving is similar to performing caching at the compute node itself. The processor effectively caches an entire chunk of data, starting from the first element in the section, at the compute node, and all required elements are supplied from this cache. The file system sees only a single request or at most a few requests, depending on the amount of memory available for the cache.

To write strided sections using data sieving, Passion first reads a large chunk of data from the file into a temporary buffer, then stores the strided section into the buffer, and finally writes the buffer back to the file. If it does not read the buffer first, data in the buffer between the strided elements will overwrite the corresponding data elements in the file. Thus, writing strided sections requires twice the amount of I/O needed to read strided sections.

Table 1 shows the performance improvement provided by data sieving over the direct method on the Touchstone Delta, for both reading and writing sections of an array of \(2K \times 2K\) single-precision real numbers distributed among 64 processors in one dimension along columns. As the table shows, data sieving performs considerably better than the direct method in all cases. The large number of I/O requests in the direct method results in more time, even though with data sieving, the processor actually accesses more overall data.

In data sieving, the total amount of data transferred depends only on the lower and upper bounds of the section; it is independent of the stride. Hence data-sieving time does not vary much across sections. There is, however, a wide variation in time for the direct method because a processor reads only the elements it needs. The time is lower for small sections and higher for large sections. As the table shows, even with a combined read and write time, data sieving performs better than the direct method for writing array sections.

| Table 1. Performance of direct read/write versus data sieving on 64 processors on the Intel Touchstone Delta. Global array size is \(2K \times 2K\) single-precision real numbers. Time is in seconds. |
|---|---|---|---|
| Array Section | Passion_read_section | Passion_write_section |
| | Direct Read | Sieving | Direct Write | Sieving |
| \((12,048,2, 132,2)\) | 52.95 | 1.970 | 49.96 | 5.114 |
| \((12,048,4, 132,4)\) | 14.03 | 1.925 | 13.71 | 5.033 |
| \((10,024,3, 32,3)\) | 8.070 | 1.352 | 7.551 | 4.825 |
| \((100,2048,6, 32,4)\) | 7.881 | 1.606 | 7.293 | 4.756 |
| \((10,024,2, 048,2, 132,3)\) | 18.43 | 1.745 | 17.98 | 5.290 |

| Table 2. Performance of an out-of-core median filtering program on the Intel Touchstone Delta. |
|---|---|---|
| Array Size | Time (seconds) |
| Without Reuse | With Reuse |
| \(2K \times 2K\) | 75.12 | 75.12 |
| \(4K \times 4K\) | 210.71 | 210.71 |
Data prefetching

As we mentioned earlier, I/O and computation are distinct phases, which means that a processor must wait while the I/O system reads or writes each data set. Overlapping computation with I/O would thus reduce a program's overall execution time. Passion uses data prefetching to accomplish this, in which it issues an asynchronous I/O read request for the next data set immediately after it reads the current one. Because the read request is asynchronous, the reading of the next data set can overlap the computation being performed on the current data set. If the computation time is comparable to the I/O time, performance will significantly improve.

Figure 9 shows the performance of an out-of-core median filtering program on the Touchstone Delta for an image of 2K x 2K pixels using a 3 x 3 window. Median filtering is frequently used in computer-vision and image-processing applications to smooth the input image. Each pixel is assigned the median of the values of its neighbors within a window of a particular size. In our implementation, the image was distributed among processors in one dimension along columns and stored in local array files. As the figure shows, prefetching improves performance by as much as 40 percent in some cases.

Data reuse

One way to reduce the amount of I/O is to reuse the data already fetched into main memory instead of reading it again from disk. To illustrate, consider the Laplace’s equation-solver program described earlier.

Suppose the array is distributed as column blocks in one dimension. The computation on each column requires one column from the left and one column from the right. The computation on the last column requires one column from the overlap area, and the computation on the column in the overlap area cannot be performed without reading the next column from the file.

Hence, instead of reading the column in the overlap area again with the next set of columns, Passion can reuse it by moving it to the first column of the array and moving the last column to the overlap area before the first column. Without this move, Passion would have to read the two columns again from the file along with the next data set. In this example, reuse eliminated the reading of two columns; the amount of savings from reuse generally depends on the size of the intersection of the sets of data needed for computations involving two consecutive slabs.

Table 2 shows the performance of data reuse for a Laplace equation solver on the Intel Touchstone Delta using 64 processors. We used two global array sizes: 2K x 2K and 4K x 4K. The performance improvement is limited because only two columns can be reused. We expect improvement to be greater in applications with more data reuse.

Acknowledgments

This work was supported in part by the National Science Foundation Young Investigator Award CCR-9357840; by the Scalable I/O Initiative, a multiagency project funded by the Advanced Research Projects Agency under contract DABT63-94-C-0049; the US Department of Energy; the National Aeronautics and Space Administration; the National Science Foundation; and a grant from Intel Scalable Systems Division. We performed this work in part using the Intel Touchstone Delta System operated by the California Institute of Technology on behalf of the Concurrent Supercomputing Consortium. Access to this facility was provided by the Center for Research on Parallel Computation.

References


June 1996