Parallel I/O for EQM Applications

David Cronk, Graham Fagg[+], and Shirley Moore
Computer Science Department
University of Tennessee, Knoxville

As processing speeds increase, I/O is becoming a bottleneck for certain classes of applications. This is particularly true for applications that perform frequent check-pointing or frequently write out intermediate results. Traditional approaches to I/O typically result in poor performance. MPI-I/O simplifies programming and potentially improves performance by allowing the programmer to make a single I/O call to access non-contiguous memory or file data, and by striping files and/or by marshaling values to combine many small reads/writes into a few large read/writes.

This paper introduces parallel I/O with an emphasis on MPI-I/O. Two production-level EQM codes, which are experiencing I/O bottlenecks, are then presented. Solutions for reducing the bottleneck through the use of parallel I/O via the MPI-I/O interface are presented, along with a performance evaluation of these proposed solutions.

1. INTRODUCTION

"The speed, memory size, and disk capacity of parallel computers continue to grow rapidly, but the rate at which disk drives can read and write data is improving much more slowly" [1]. This is a simple, yet very important, observation. This problem is exacerbated by the fact that most users of parallel computers have little or no inclination to learn the intricacies of I/O optimization. Traditional approaches to I/O do not provide the performance needed for the problem sizes that can now be handled by today's high performance computers. This is resulting in applications whose size has traditionally been limited by the time needed to compute solutions now being limited by the time needed to read data or write results out to disk.

Storage devices perform best when they transfer large, contiguous blocks of data. This is referred to as coarse-grained access. Unfortunately, many scientific applications have fine-grained data access patterns, causing the I/O to perform many small reads or writes rather than a few large reads or writes. Fewer larger reads or writes would perform better.

[+] Project Principle Investigator. Email address: fagg@cs.utk.edu

As an example, consider a scientific application that needs to write a two dimensional array to disk. Further, assume the two dimensional array in memory has a layer of ghost cells around the block of data (see figure 1). This results in the data to be

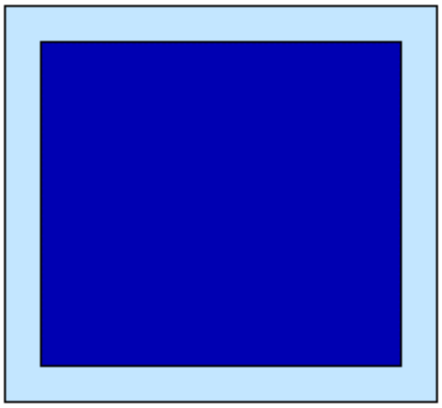

Figure 1. 2D array with ghost cells

written to disk not being stored in a contiguous block of data, and thus each column (assuming column major storage) must be written separately. To make the problem worse, most scientific code is written in Fortran and many scientists would write this data using an implied do loop, essentially causing each data element to be written separately. The most efficient way to write this data would most likely be to marshal all the data into a contiguous output buffer, and then perform a single, direct, write to disk. Why don't more scientists use this method? Simply because they are doing it the way it has always been done. Many scientists simply accept that they will not get good performance for I/O and don't look for even simple methods to improve the I/O performance. This problem gets even worse if it is running in parallel and the block of data is a sub-section of a larger two dimensional array stored on disk (see figure 2). In this situation the solution proposed above will not work because the data being accessed on disk is not stored contiguously.

This type of problem is commonly handled in one of three different ways. One solution is to for each process to write its data to a process specific file. Then, once the computation is complete, a post-processing step is needed, where a single process reads each process specific file and handles the data appropriately. Another common solution involves each process sending its data to a master process. This master process then organizes the data appropriately and writes it out to disk. A third common solution is for each process to use direct file access and write the data to the output file directly. While the first two solutions

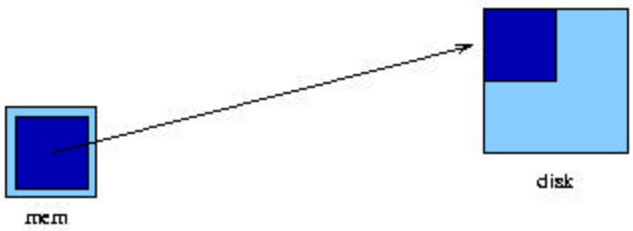

Figure 2. 2D array stored as part of a larger 2D array

clearly provide no parallelism, the third solution at least has the appearance of parallelism. In practice, however, the third solution typically provides poor performance. This is due to a couple of factors. One factor is the data typically are accessed in small blocks, and as discussed above, the best performance is attained when data is accessed in large blocks. A more important factor, however, is that the user typically is not using a parallel file system (NSF if the most common one used) and thus even though the requests are made in parallel, the file system serializes the requests.

This is a simple example using a very simple data access pattern. However, it is instructive in demonstrating the problems that may arise when performing I/O during a parallel computation. Access patterns vary greatly across different scientific applications. Further, new high performance architectures are introduced regularly, and each may have very different I/O system characteristics [1]. This would mean that a scientist wanting good I/O performance would be required to learn the I/O system of every architecture on which his code might run. The alternative is the use of a standard interface that is portable across many architectures. The use of such a standard interface may simplify coding by reducing the number of I/O calls required, and by allowing the user to become familiar with a simple interface. Further, a standard interface allows simple access to libraries that are optimized for particular architectures, maximizing the chance of significantly improved performance.

The rest of this paper is organized as follows: Section 2 presents an overview of parallel I/O and an introduction to the MPI-I/O interface. Section 3 presents experiences with real life applications followed by conclusions and future work in Section 4.

2. PARALLEL I/O

Parallel I/O can be simply described as multiple processes requesting concurrent access to a single, shared file. Often these processes are accessing non-contiguous data (both as stored in memory and as stored on disk) and users wish to make as few I/O calls as possible. In order to perform parallel I/O, there must first be a parallel file system to provide parallel file access.

The first thing that is required in order to be able to supply parallel file access is multiple disks. By having multiple disks, each with its own I/O node, different processes can have file access at the same time. However, if a file is still stored on a single disk, concurrent access to the file by different processes is still not possible. What a parallel file system may provide is the ability to stripe files across multiple disks. That is, a single file is actually stored in *blocks*, which span multiple disks. This provides the ability to grant different processes true parallel access to the same file (provided each process is accessing parts of the file stored on different disks).

Once a parallel file system is in place, parallel I/O is possible. More important for the purpose of this paper, it is possible to provide a parallel I/O library to the user. This library should present a simple interface and perform optimizations that lead to improved performance. There are a number of optimizations that a parallel I/O library may be able to make. In order to perform the best optimizations, the library may need some help. This help is often supplied by *hints* provided by the user. These hints may include such things an suggested striping factors (number of disks used to store a file), striping depth (size of *blocks* of data stored in round robin fashion on each disk), or other *suggestions* made to the library. These may be used for better optimization. One such optimization involves marshaling many small non-contiguous I/O requests into a single, larger, contiguous I/O request. As discussed in Section 1, storage devices perform best when they transfer large, contiguous blocks of data. Another potential optimization involves buffering. If there are separate I/O nodes, computation and I/O can be overlapped by the I/O library buffering output and returning control to the calling process. This allows the I/O node to complete writing the output to disk while the process continues to perform computations.

While these optimizations (as well as others) may provide improved performance, they also introduce additional issues in terms of file consistency and semantics. These issues may have profound effects on the correctness of some classes of parallel applications. These issues will be discussed in more detail in specific reference to how they manifest themselves in the MPI-I/O interface.

2.1 MPI-I/O

MPI-I/O is part of the MPI-2 standard, which is a set of extensions to the original MPI standard. It is important to note that MPI is nothing more than an interface specification. That is, it specifies the syntax and semantics of the various MPI routines, but is does not include any specification of how these routines should be implemented. This means different MPI implementations may be very different in terms of performance. This will be pointed out repeatedly in the following sections as the potential benefits of different features are discussed.

MPI-I/O provides routines for manipulating files and for accessing data. The hope is that MPI-I/O will become *the* standard for performing parallel I/O. It is already making inroads towards that goal as more and more vendors are providing efficient implementations of the MP-I/O interface. This allows programs that use MPI-I/O to be portable across a large number of architectures.

2.1.1 File Manipulation

MPI-I/O provides several routines for file manipulation. These include routines for deleting files, resizing files, and querying for file information. The most basic routines, however, are for simply opening and closing files.

When a user opens a file using MPI, a communicator must be supplied. This communicator defines the *group* of processes involved in accessing the file. Opening a file is a collective operation, which means all the processes in the group must open the file. One of the parameters provided to the open call is a mode for opening the file, which must be the same at all processes. These modes are similar to the standard file modes. If the create mode is included, the file will be created if it does not already exist. When a file is opened, the user may pass hints to the library via a parameter called *info*. Info may be used to give the library information such as file access patterns. The library need not use hints and different implementations will support different hints. The only requirement is that unused hints are ignored. This allows for portability between MPI implementations.

Closing a file in MPI is also a collective routine. This means that all the processes involved in opening the file must close the

file. Like many collective routines in MPI, closing a file is not synchronizing. This means that by closing a file, a process has no information regarding the state of other processes. The other processes may or may not have closed the file. While closing a file is not synchronizing in terms of other processes, it is locally synchronizing. As mentioned previously, MPI has the option of buffering output before writing it to disk. This means that just because a write operation has completed, there is no knowledge as to whether the data has been written to disk. When closing a file is described as locally synchronizing, this means that closing a file forces any buffered data from the closing process has been flushed to the disk. Again, this says nothing about other processes, only the local process. Other processes may still have output data buffered.

2.1.2 Derived Datatypes

Though derived datatypes are part of the original MPI standard rather than MPI-2, they are used extensively with MPI-I/O. For that reason a brief discussion is included here.

Derived datatypes are used for I/O both for defining the data layout in memory as well as the data layout in the file. When a datatype defines the data layout in a file, it is typically referred to as a filetype. It is important, however, to realize that a filetype is simply a derived datatype, just like any other derived datatype. It has no special meaning until it is used to set a file view. This will be discussed in Section 2.1.2.1.

As an example of why one would want to use a derived datatype to access memory when doing I/O, refer back to Figure 1. Suppose the user wants to write just the resident (non-ghost) cells to disk. Using Unix style I/O, the user would need to make separate I/O calls for each column (assuming column major ordering). However, by using derived datatypes, the user can define the memory access such that a single element represents just the resident cells. This would be done with a call to MPI_TYPE_SUBARRAY. By providing the starting points and size of the resident subarray, the user can define the type and subsequently write the entire subarray with a single MPI-I/O call. As will be seen in the following section, even if the data are not being stored on disk in a contiguous block of storage, the user need only make a single I/O call. The ways this can improve performance will be covered in 2.1.3. Derived datatypes for accessing memory in I/O are used the same way as they are for message passing.

2.1.2.1 FILE VIEWS

A *file view* in MPI defines how a process sees a file. That is, it defines which portions of the file are visible to the process [2]. A process may only read from and write to those portions of the file visible, as defined by the view of the file. In fact, when reading (or writing), the parts of the file that are not visible to the process are automatically skipped. File views are created by first creating a derived datatype that defines how the process sees the file. This derived datatype is then used in a call to MPI_FILE_SET_VIEW. Following the call that sets the view, the process accesses the file as if this view represented the entire file. Additionally, views are set with a datatype that represents the elementary element types stored in the files. This is the smallest element that may be read from or written to the file. For example, if this type is an integer, then individual bytes can not be accessed in the file.

Returning to Figure 2, it is now possible to access the file with a single I/O call. First a datatype must be defined that represents the process' view of the file. This can once again be done as a subarray. By defining a subarray that represents just the shaded area of the larger array, the view has been defined. This datatype is then used as part of the call to set the view. Once this is done, the process can only access the shaded area of the file. There are several ways that this file can be accessed. The user may simply read single elements from the file. MPI will skip the non-shaded portions of the file automatically. Alternately, the user may access the shaded portion one column at a time. The difference from the way this would have been done with Unix style I/O is that the user need not calculate where each column resides in the file. Setting the view has done this. After reading the first column, MPI automatically adjusts the file pointer to skip the rest of the column and point to the start of the next column. A third, and possible most efficient, way to access this data is to use both derived datatypes discussed above. A single element defined to represent the memory portion of Figure 2 can be read from the file. Since the datatype defined for the left side of Figure 2 tells MPI how to store the data inside the ghost cells, the user only needs to use a single element of this type with the file view discussed here. This will cause a read operation to read just the shaded area from the file and store it in just the shaded area of the memory. This not only makes the code easier to write and maintain, but as will be seen in Section 2.1.3, may lead to greatly improved performance.

There are a number of common file access patterns that can be supported in this same way. Some examples include block-cyclic data distribution (use MPI_TYPE_VECTOR to define the file view) and even irregularly distributed arrays. By using a variety of datatype constructors, a user should be able to define about any memory and file layout. This should allow the user to access non-contiguous data in an easily recognized and straightforward manner.

2.1.3 DATA ACCESS

There are three aspects to data access: Positioning, synchronism, and coordination [3]. Figure 3 represents these three aspects.

Positioning. MPI provides three types of routines in regards to positioning: explicit offsets, individual file pointers, and shared file pointers.

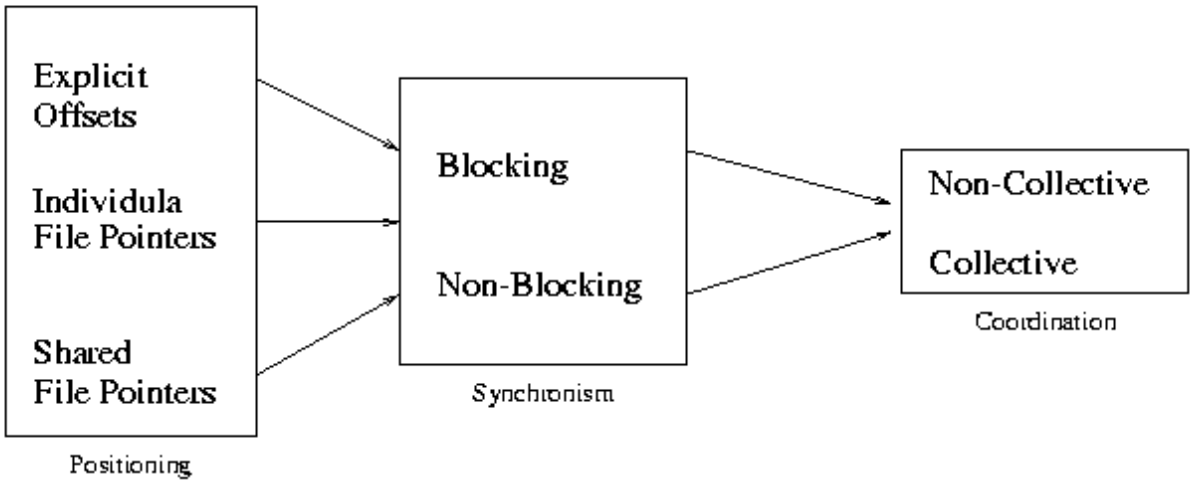

Figure 3. Data access methods

Explicit offset operations perform data access at the specific location given directly as an argument passed into the routine. These calls do not update any file pointers. These accesses are in relation to the file view discussed in Section 2.1.2.1 and the offsets are in terms of the elementary datatype used when setting the view.

For each process, MPI maintains two file pointers for each open file. One of these file pointers is the individual file pointer. This is unique to each process and updates to it in one process do not affect the individual file pointers in other processes. One can liken individual file pointers to file pointers used in Unix style I/O. Individual file pointer operations access the file according to the implicit offset provided by the calling process'

individual file pointer. The calling process' individual file pointer is updated accordingly upon completion of the operation.

MPI also maintains a shared file pointer for each open file. This pointer (as its name suggests) is shared between all processes. Shared file pointer operations access the file according to the implicit offset provided by the shared file pointer. The calling process' individual file pointer is neither used nor updated. Upon completion of the operation, the shared file pointer is updated, with this update being reflected at all processes that were part of opening the file.

Synchronism. Synchronism in MPI has nothing to do with other processes, but rather the ability to use data buffers involved in an operation. It is important to keep this in mind. MPI-I/O offers both blocking and non-blocking I/O routines. Non-blocking operations return immediately. It is not safe to reuse user buffers until the non-blocking operation has been completed with a test or a wait, just like non-blocking communication. Blocking routines do not return until a user buffer can be safely re-used. In terms of reads, this means the data has been read and is available in the user buffer. Blocking operations performing writes to files do not return until the data has been copied out of the user buffer. It is important to point out here that this does not mean the data has actually been written to disk when a blocking write returns or a non-blocking write is completed (with a test or wait). MPI is free to copy the data to a buffer and write it out to disk at a later time. In fact, this method is often used for optimization. The user must make no assumptions about data availability simply because a write operation has completed. This is similar to the completion of a blocking send in MPI giving no information about whether the message has been received. Methods for assuring data have been written to disk will be discussed in Section 2.1.5.

Coordination. MPI offers both collective and non-collective I/O routines. Non-collective operation depend on no other processes and work how one would expect them to work. Collective routines, on the other hand, require the coordination of all the processes that opened the file. While these routines are collective, they are not synchronizing. Even if a process has completed a collective I/O call, no information is known about the status of the other processes.

MPI provides a complete combination of these different data access aspects. That is, each data access routine relating to positioning (explicit offset, individual file pointers, shared file pointers) has both a blocking and a non-blocking version. Further, each of these routines (explicit offset – blocking, explicit offset – non-blocking, etc) has both a non-collective and a collective version. It should be noted that non-blocking collective routines are referred to as split-collectives and require both a start call and a finish call.

The semantics of data access in MPI are fairly loose. This gives implementers a lot of leeway for the purpose of performance optimization. There are two primary ways improved performance is achieved. The first is through buffering. Buffering data that is to be written to disk offers two ways to optimize. One way is to

keep the data in a buffer in case subsequent writes allow the library to marshal data into contiguous blocks of storage data. As has been mentioned, storage devices perform best with large, contiguous blocks of data. Another way buffering can improve performance is by hiding the time needed to perform I/O. If there are dedicated I/O nodes, buffering can allow these I/O nodes to perform I/O while the compute nodes are busy continuing with the computation.

One of the most important ways performance can be improved is through collective operations. By using collective routines, the user is letting the library know that all the processes are going to be performing I/O at about the same time. The library can use this knowledge to better marshal data. Returning to Figure 2, if the library knows that all four process are going to write to disk (and this constitutes the entire file) it can marshal all the data into a single output buffer and make a single write to disk. Without this knowledge it would most likely write each process' data column by column, making many more disk access than necessary.

As mentioned previously, buffering can cause difficulties in terms of file consistency between processes. These issues will be covered in more detail in Section 2.1.5.

2.1.4 FILE INTEROPERABILITY

MPI says nothing about how routines should be implemented. This extends to files created through MPI-I/O. MPI says nothing about how these files should be stored. Files may be stored as ordinary files, compressed, stripped across a disk array, or some other way. The only constraint MPI places on file format is that if the file is not stored as a linear sequence of bytes (like a typical file), then there must be a utility provided for converting the file to a linear sequence of bytes, as well as utilities for common file operations such as copying and deleting. This ensures that any non-MPI program will be able to read the file, though it may first need to be converted [2].

Some users may wish to create files on one architecture and then be able to read them on another architecture. This is generally not possible due to different data representations on different architectures. In order to do this there must be some sort of data conversion. This is supported by MPI. When a process sets a file view, it specifies if the data representation is to be *native*, *internal*, or *external32*.

When native representation is used, the data are stored in the file exactly as they are in memory. This means that files created with this representation cannot be read on a different architecture. This file can be read by programs using the same MPI implementation on the same architecture. There is no guarantee it can be read using other MPI implementations because the file structure may be different. This is a non-portable representation, but typically provides the best performance.

The internal representation provides some degree of portability. The implementation may store the data in any format it chooses and will perform type conversions if necessary. The environment in which the file can be re-used will be implementation dependent and must be documented [3]. A particular MPI implementation may use a data representation that allows the file to be read on any architecture using the same MPI.

The external32 representation causes the data to be stored in a specific data format as defined by MPI. This format is basically IEEE big-endian format. This ensures that a file written with this data representation can be read by any MPI implementation on any architecture. This is, of course, assuming external32 format is supported by the MPI implementation being used. Since data conversion is being performed, there may be a loss of data precision and poorer performance can be expected.

2.1.5 CONSISTENCY SEMANTICS

As discussed previously, there are many instances where buffering data can improve performance. This is particularly true with regards to writing data to disk. Because of this possible buffering, file consistency becomes an issue. One must remember that just because a write operation has completed, there is no guarantee the data are actually written to disk and available to other processes. Keep in mind also that collective routines are not synchronizing.

Consistency is only an issue if multiple processes are accessing the same file and at least one of the processes is writing to the file. Further, if no two processes access the same location in the file, consistency is maintained. Therefore, consistency semantics are of interest when multiple processes access the same location of the same file, with at least one process writing to the file.

Generally, there is only one way for a process to be guaranteed access to data written by another process. The reading process must be sure the writing process has completed writing the data and some file synchronization has been performed. There are three methods to ensure file synchronization: MPI_FILE_SET_ATOMICITY, MPI_FILE_SYNCH, and closing a file. These are all collective operations and a process should complete the operation and know the other processes have completed the operation to be guaranteed consistency.

MPI_FILE_SET_ATOMICITY is a collective operation that ensures all subsequent writes cause the data to be flushed to disk. This means, if atomicity is true, once a process has returned from a blocking write (or completed a non-blocking write) the written data have been transferred to disk and are available to other processes. Although this is a collective operation, it is not synchronizing.

MPI_FILE_SYNCH is a collective operation that acts much like a flush. A synch call causes any buffered output data to be written to disk. Since it is collective, all processes must call it and this insures data written by other processes are accessible. Although this is a collective operation, it is not synchronizing.

Closing a file is a collective operation that acts much like a synch, except it causes the file to be closed. That is, it causes all buffered data to be written to disk before the file is closed. Thus, if a process closes a file and knows another process has closed the file, it also knows it can access all the data written by said process. Although this is a collective operation, it is not synchronizing.

It is important to remember these are all collective operations and they must be used appropriately. Knowing another process has called one of these routines is not enough to ensure consistency. The reading process must have called the synchronizing routine as well. This is because MPI can buffer input data as well as output data. MPI may pre-fetch data from the disk. If the data on disk change after the pre-fetch, a subsequent read will get the out-of-date data unless the consistency semantics have been followed.

3. APPLICATIONS

This section presents two EQM (Environmental Quality Modeling) applications that have been suffering from I/O bottlenecks. The I/O characteristics of the code as well as the original approach

to I/O are explained. This is followed by a description of the approach taken for converting the existing I/O to MPI-I/O as well as a discussion of performance results where available.

3.1 LBMPI

LBMPI is a Contaminant Dispersion model EQM challenge code being used by Robert Maier at the Army HPC Research Center. The code performs computation using what are conceptually four 3-dimensional arrays. In practice, however, the four 3-dimensional arrays are stored as a single 4-dimensiona array. This is done to achieve a better cache-hit ratio. Each of the 3-dimensional arrays consists of interior resident cells as well as a layer of ghost cells. The application uses a block-block data distribution. This code is experiencing a severe I/O bottleneck. Running on an IBM SP2 using 512 processors, a typical run requires approximately 12 hours of compute time and an additional 12 hours of I/O time.

3.1.1 I/O CHARACTERISTICS

This code writes each of the conceptual 3-dimensional arrays out to a separate file. The result is four global files that each contains the appropriate data from all the processes. The resulting files store the data in their natural order. That is, the data are written the same regardless of the number of processes.

The code uses Fortran's column major ordering with the first dimension being the number of conceptual 3-dimensional arrays (four in this case). This results in no two adjacent values in one of the 3-dimensional arrays being stored in adjacent memory locations. Thus the 3-dimensional arrays that need to be written are not stored in contiguous memory, and no two elements stored in adjacent memory locations. Furthermore, since a block-block data distribution is used, the data from a particular process is not stored in a contiguous block on disk. Each column from the 3-dimensional array is stored in a contiguous block on disk, but adjacent columns are not.

3.1.2 INITIAL SOLUTION

The original solution attempted to get parallel performance using standard Unix style I/O calls. When data was to be written to disk, each process would open the output file. Once the file was

opened each process would calculate where in the file its data were to be stored. Each process then wrote the data directly to the calculated offset using a standard “write” call. A nested do loop was used to loop through every column in the 3-dimensional array, and an implied do loop was used to write each element of the column. As was mentioned above, the column elements are not stored contiguously in memory, so it was impossible to write each column out as a single block of data.

Although each process can make these calls concurrently, the application was not using a parallel file system. Thus the I/O calls were serialized, resulting in no real parallelism. Furthermore, since the data were stored non-contiguously in memory and no marshaling of data was used, performance was poor.

A final note on the original solution relates to convention. The output is written to four separate files because it has always been done this way. Since the data are stored in a single 4-dimensional array, it would possibly be more efficient to write the data to a single output file. This would allow contiguously stored blocks of memory to be written to disk.

3.1.3 MPI-I/O SOLUTION

There are two issues here that can be addressed through MPI. These are the memory access and the file access. In both cases the application is accessing a portion of a large array. The application accesses each conceptual 3-dimensional array separately. These are really just sub-arrays of the larger 4-dimensional array. On disk, each process is accessing a 3-dimensional block that it a part of the larger 3-dimensional global file. Both situations lend themselves nicely to the MPI SUBARRAY datatype. Thus SUBARRAYS are the basis for the MPI solution.

Four distinct solutions were developed. First, a solution writing to a single file and a solution writing to four separate files were developed. Each of these solutions has a variation using non-collective I/O operations and a variation using collective I/O operations. The basic steps taken for each solution are as follows:

1. Set up arrays defining the sub-arrays in memory
2. Set up arrays defining the sub-array on disk
3. Construct new datatypes defining memory (mtype) and file (ftype) access (use MPI_TYPE_CREATE_SUBARRAY)
4. Commit these new datatypes
5. Open the file using MPI_FILE_OPEN

6. Set the file view using the datatype ftype, as defined in step 3
7. Make a single call to MPI_FILE_WRITE to write a single element of type mtype, as defined in step 3
8. Close the file using MPI_FILE_CLOSE
9. Free the datatypes using MPI_TYPE_FREE

3.1.4 PERFORMANCE

Table 1 presents the benchmarking results of each of the four solutions. These results were obtained by taking the average time for five execution runs on the Cray T3E located at the US Army Corps of Engineers ERDC in Vicksburg, MS. These runs used the Cray T3E's native MPI implementation. The first column lists the number of files written to (one and four). The second column lists the time in seconds used for I/O using non-collective operations. Finally, the third column lists the time in seconds used for I/O using collective operations. As a comparison, the original (non-MPI) solution used 5402 seconds to complete the I/O.

Number of files	Non-collective (time)	Collective (time)
1	889.4 seconds	110.2 seconds
4	4144 seconds	58.64 seconds

Table 1. Total I/O time for LBMPI

Some of these numbers may seem a bit surprising at first glance. Looking first at the numbers for non-collective operations, there is very little improvement when writing to four separate files, as compared to the non-MPI results. This indicates that MPI is not marshaling the output data. Better performance is achieved writing to a single file, but this can be attributed to writing large contiguous blocks from memory as opposed to completely non-contiguous memory access as in the four-file case. However, since collective operations are not used, MPI does not appear to marshal any data to allow larger blocks of data for file access.

The numbers in the third column are very instructive. It has been stated several times that collective operations present the best opportunity for I/O optimization. This is reflected in these numbers. Since collective operations were used, it appears MPI used the knowledge that each process would be performing I/O to marshal data and allow fewer disk accesses using larger blocks of data. In the single file case, it may be possible to make just one disk access by marshaling the data from every process into a single output buffer.

The fact that using collective operations to write to four files outperforms using collective operations to write to a single file is also instructive. No *hints* were used when creating these files. It appears the MPI being used creates files on a single disk if no hints are offered. This means there is no true parallelism available for the I/O (there is only a single read/write head per disk). The reason writing to four files is faster is that each of the files may reside on different disks. This means each file can be accessed concurrently. The next step to be taken will be to use hints to get MPI to have the file system stripe the single file across multiple disks. This should introduce true parallelism to the I/O and performance should improve. This performance improvement should be seen for both the single file case and the four file case.

3.2 CE-QUAL-ICM

CE-QUAL-ICM is an EQM code developed at the Army Corps of Engineers ERDC in Vicksburg, Mississippi. The code performs computation using a large number of arrays with the parallel version distributing the arrays in an irregular fashion. These arrays also contain a high percentage of ghost cells (typically 20%-35%). Typical production level executions perform ten-year simulations.

3.2.1 I/O CHARACTERISTICS

A typical execution run of this application writes intermediate results after every simulation month. Each successive write of intermediate results is appended to the previous intermediate results. These intermediate results consist of all the resident cells of all the arrays being used in the computation. Since the arrays are irregularly distributed, no assumptions can be made about any two cells being stored in contiguous locations in the output file. In fact, it is possible that two resident cells are stored in a different relative order on disk than in memory. Figure 4 shows an example of what this might look like. There is a mapping available that defines where each resident cell in each process gets stored in the file. This mapping is the same for each output array and for each output iteration. That is, for a particular process, the same mapping is used many times.

3.2.2 INITIAL SOLUTION

The original solution requires post-processing. Each process writes results to a process specific file. During an output

phase, each process opens its own file, and writes every array to that file. The entire arrays are written including ghost cells. Additionally, the arrays are written in the same order as the are stored in memory.

Following the execution run, a sequential post-processing step is required. The post processor reads two mapping files. The first mapping file defines which cells from each array, for each process, are resident cells. The second mapping defines where

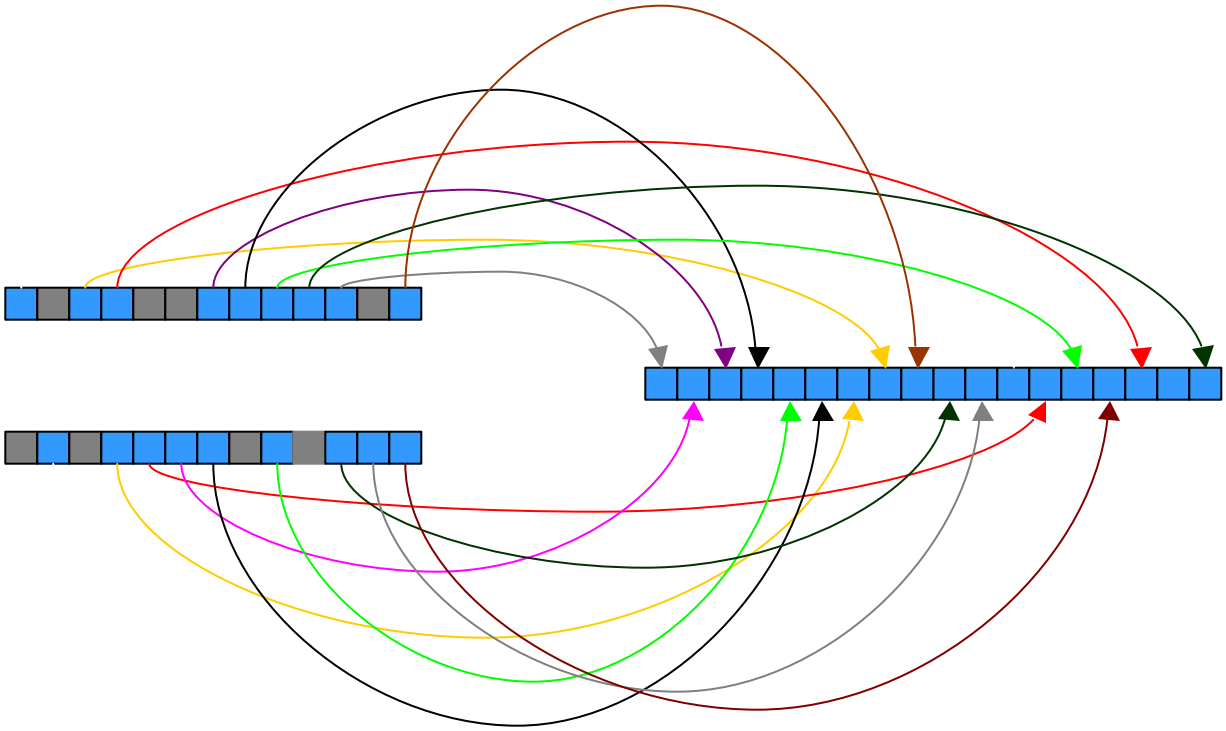

Figure 4. Irregular data distribution

each resident cell, for each process, is to be stored in the global output file. These mappings repeat for each array and for each iteration.

Once the mappings have been read, the post processor opens a large global output file. This is the file that will store the final results. It then enters an outer loop to process each sequence of intermediate writes. For each intermediate write, the post processor loops through the number of processes used to create the output. For each process, the process' output file is opened and all the data written for the particular iteration are read. Then

each element of each array is checked to see if it is a resident cell. If it is a resident cell, the second mapping is used to determine where in the global output the value belongs. The value is stored in this location in the appropriate large global output array. There is a global output array for each of the arrays written to the file.

Once this has been done for all the processes, the global output arrays are full. The post processor then simply writes each of the global output arrays to disk. Once all the write iterations have been processed, the post processor closes the global output file. Once this is complete, the process specific files may be discarded.

3.2.3 MPI-I/O SOLUTION

The challenge is to find an efficient solution using MPI that eliminates the post-processing step. It must be a solution that does not increase the execution time substantially.

Again there are two main issues that can be addressed by MPI. These are memory access and file access. Since the arrays are irregularly distributed, file access is going to be irregular. Also, since there are ghost cells throughout the arrays (not just the beginning and end) memory access is irregular. Irregular access lends itself very well to MPI's indexed datatype. MPI2 offers a new datatype called INDEXED_BLOCK that defines a datatype with blocks of data all of the same size. This is ideal for this problem. One problem not mention previously is that any datatype used for defining a file view must provide for non-decreasing access to the file. This means an indexed datatype that defines accesses like those shown in Figure 4 cannot be used for a file view. No such restriction exists for memory access.

The MPI solution is based on the INDEXED_BLOCK derived data type for both memory and file access. The map arrays are already available as discussed in Section 3.2.2. However, the mapping defining where each resident cell goes in the global file must be non-decreasing. This is not guaranteed. What is done is this map array is sorted. At the same time, the map array that defines which cells are resident cells is permuted to match this sorting. At this point memory access may not be non-decreasing, but file access is. These map arrays are used to construct a derived datatype using MPI_TYPE_CREATE_INDEXED_BLOCK. The map array defining the resident cells is used to construct the datatype for memory access (mtype), and the map array defining where resident cells go in the global file is used to construct the datatype for

file access (ftype). The file type (ftype) is then used to set the file view. This file view needs to be updated after every write to reflect the new position in the file so previously written data is not overwritten.

The MPI solution has a master process read some global data needed to calculate displacements into the file. These data are broadcast to the rest of the processes. The master process then reads the map arrays for each process and sends them to the appropriate processes. Once this is complete, each process has all the information needed for I/O. Each process must sort the appropriate map array while permuting the resident cell map array. Once this is done, each process constructs the two datatypes as describe above. The first time any data are written, the file starts at the beginning. On subsequent writes, however, the data previously written must be skipped. This is possible by keeping track of the total number of bytes written. This number is used when setting the file view.

The basic outline of the MPI solution is as follows:

1. Master process sends necessary information to other processes
2. Each process constructs INDEXED_BLOCK datatypes as described above
3. Each process opens the global output file
4. Each process sets displacement to 0
5. Each process sets the file view using ftype as described above
6. For each array to be written,
    7. Each process writes a single element of type mtype as described above. This causes all the resident cells to be written to the file in the appropriate location
    8. Displacement is updated by adding the total number of bytes written in step 7
    9. The file view is set again using the new displacement. This causes all the previously written data to be skipped
10. Steps 6 though 9 are repeated for each series of writes (typically every simulation month)

Unfortunately, MPI_TYPE_CREATE_INDEXED_BLOCK is not supported by all implementations of MPI. For this reason this type was replaced with the more general MPI_TYPE_INDEXED. The INDEXED type is the same as the INDEXED_BLOCK type except the blocks can have different sizes. By using an array of sizes where every size is 1, the INDEXED type is the same as the INDEXED_BLOCKED type with block size of 1. This is sufficient for the above solution.

3.2.4 PERFORMANCE

The above solution has been completely implemented. Unfortunately, the implementation is not yet in working order. The current status is that the code core dumps during

initialization. While this is in a part of the code added to support the MPI-I/O solution, it is prior to any actual MPI-I/O activity. It seems to be an obscure bug and the testers simply ran out of time prior to this writing. Work on getting the code to run successfully continues.

4. CONCLUSIONS AND FUTURE WORK

The use of parallel I/O potentially offers significant improvement in I/O performance. This can be accomplished by a number of methods, including striping files across multiple disks and marshaling data to achieve coarser-grained data access. However, different architectures provide different parallel file systems, each with different characteristics. In order for a user to be able to get good performance on many different architectures, the intricacies of the I/O subsystem of each architecture would need to be understood. This is an unrealistic expectation for a typical user. Therefore, a standard interface is needed. MPI-I/O offers a simple API that is portable across many different architectures. If a user takes the time to study the proper use of MPI-I/O, that user gains the ability to write code that performs parallel I/O on all architectures supporting the MPI-I/O subset.

In addition to improved performance, programs that use MPI-I/O may be easier to maintain. The use of derived datatypes with MPI-I/O makes it easier to understand what data and file access patterns are being used. This makes it easier to understand the program and thus easier to maintain the program.

Once the concepts of MPI-I/O are understood, it is not difficult to convert applications using traditional Unix style I/O to use parallel I/O through MPI. Once the I/O characteristics had been studied and understood for the application discussed in Section 3.1 (LBMPI), it took less than one week to add the MPI-I/O calls. Likewise, once the I/O characteristics of the application discussed in Section 3.2 were understood, it took less than a week to insert the necessary code to use MPI-I/O.

Future work will involve continuing work on the applications discussed in this paper as well as some new initiatives. Work on getting the MPI-I/O version of the CE-QUAL-ICM code running will continue. Additional work on the LBMPI code will also continue. There are plans to investigate the effects of adding hints to the LBMPI code. This should allow stripping of the output files, which should lead to improved performance.

There a re also plans to investigate the use of different data formats to support better file interoperability. One such format is the Hierarchical Data Format (HDF). The HDF5 format will be studied and there are plans to investigate parallel I/O support for HDF5 [4].

Acknowledgement

This work was supported in part by a grant of HPC time from the DoD HPC Modernization
Program.

References

1. John M May, *Parallel I/O for High Performance Computing*, Academic Press, San Diego, CA, 2001
2. William Gropp, Ewing Lusk, and Rajeev Thakur, *Using MPI-2, Advanced Features of the Message-Passing Interface*, The MIT Press, Cambridge, MA 1999
3. William Gropp, et. Al., *MPI – The Complete Reference, Volume 2, The MPI Extensions*, The MIT Press, Cambridge, MA 1999
4. The NCSA HDF Home Page, *http://hdf.ncsa.uiuc.edu*