Technical Report

RAL-TR-2006-029

31 October 2006

On the Use of Parallel Input/Output in the High Performance Application Code, POLCOMS

J.V. Ashby
Computational Science and Engineering Department
CCLRC Rutherford Appleton Laboratory
J.V.Ashby@rl.ac.uk

M. Ashworth
Computational Science and Engineering Department
CCLRC Daresbury Laboratory
M.Ashworth@dl.ac.uk

Abstract

As the problems addressed by High Performance Computational Science programs increase in size and complexity, the need for efficient input and output of datasets becomes an important factor in program design. Such datasets arise naturally as initial data and final results, but frequently programs save intermediate data during a long run to enable restarting the calculation in the event of a program or machine failure. Gathering all the data on one processor forms an inherent bottleneck, and for suitably large problems may exhaust the memory available to the master processor.

We discuss the general problem of parallel I/O and the features of some of the available solutions. The checkpointing phase of a large High Performance scientific program, POLCOMS, is implemented in parallel using MPI-IO, a set of routines from the MPI-2 standard. The performance of MPI-IO is compared with that for gathering the data to one processor and writing it serially using a simple Fortran WRITE statement. We find that it is important to make sure the data representation being written out and the characteristics of access to the data are carefully selected, or serious performance degradation can be produced, as well as loss of flexibility. At its best, however, MPI-IO is comparable with the naive process, and is capable of handling much larger problems.

1. Problem definition

With the increased use of high performance computing systems providing access to capability computing, the complexity and scale of scientific problems which can be addressed is also increasing. Traditionally the performance indicators for such a system have been processor speed (or floating point performance, which is not always the same thing), communication bandwidth and latency and the size of main and cache memories. However, the larger and more complex the problem, the larger and more complex the data which must be stored at various stages during the computation. Consider, for example, a three-dimensional finite difference problem such as fluid flow or electro-magnetics with perhaps 6 double precision (8 byte) variables at each node of the finite difference grid. With a thousand processors available and a fairly modest grid of 100 by 100 by 100 on each processor, thus 10^9 nodes, the solution alone will take up 48 Gbytes. If this is to be stored at 20 time steps, to enable a movie to be made of the solution evolving, for example, we readily approach 1Tbyte of data.

Datasets of this magnitude are stored for a variety of purposes. They may be archival material, a record of the solution or program state. Often programs will write out intermediate solutions as an insurance policy against losing the results of long runs, whether by computer failure, program failure or simply running out of resources. Some algorithms, so-called out of core methods, rely on writing to disk data which is too large to be held in memory – effectively these methods perform their own memory paging. Such methods were common in the early days of computing and are experiencing a resurgence as the expectation of large problem sizes grows. Stored data, especially final solution data, becomes available for transfer to other programs which might be, for example, other simulations, visualization or data reduction software. Whatever the use to which the data will be put, it is clear that it will need firstly to be written and secondly to be read in an efficient manner. Parallel I/O systems attempt to achieve that efficiency by recognizing that the data starts off distributed among the processors of a parallel computer, and will, in all probability be read in and distributed.

The naive method of I/O has traditionally been the Gather-write/read-scatter (GWRS) one where the data to be written is all gathered onto one process (usually the master process) and then written to a single file opened by that process. Similarly the data from the file is read onto a single process and sent to the appropriate process for computation. There are two main sources of bottlenecks in this method. Firstly in a large simulation with many processors, a great many of them can be standing idle while a single processor handles the I/O. Also, if the data is being assembled from its local parts into a global form for writing (or read in as a global form for partitioning into processor sized pieces), the I/O process must be able to handle very large amounts of data in memory, and for even modest sized problems this can easily be impossible. For both these reasons a parallel solution to I/O is desirable.

2. Brief attributes of parallel I/O systems

A parallel I/O system is a combination of hardware and software which operate together to speed the storage and retrieval of information by using concepts such as striping files across RAID-style disk arrays, accessed by multiple processors via fast switches. From the users' point of view,

however, what is important is how the system is accessed, both from within the code (the API) and at the operating system level where ideally a file will appear as a single file on which all the standard file operations (mv, rm, cp, chmod, etc.) can be performed, irrespective of how the data is physically arranged on the disk or disks. Most effort has been directed at the definition of parallel file systems allowing striped access to files (which may be physically distributed) and the associated system calls. Examples are: GPFS for IBM [1], XFS for SGI hardware [2], Sun PFS [3] and the Lustre file system [4] which is used by Cray XT3 and XD1s, as well as the IBM BlueGene.

When accessing a parallel file system, various strategies can be employed. Processes may write and read files either independently or collectively. If access is independent, then there must be mechanisms to ensure coherency of data, just as in shared memory architectures. Since it is possible that data will be written and read by different processors, many systems use pre-fetching to move data to areas of the disk array best connected to the processor which will be using it. Caching can also be used to improve performance by, for example, writing the data to temporary storage (either on disk or in cache memory) and releasing the program to continue with other work while the data is committed to permanent disk storage.

Systems which are hardware based are generally tied to one specific manufacturer, or even to one subset of a manufacturer's product range. For the application programmer, this represents a problem with the portability of any code which uses such a system. The usual approach to overcome this is to use a standard library which provides the desired functionality to the program, and which can be interfaced to the system level specific layers for each machine. The most commonly used example of such a library is MPI-IO, now part of the MPI-2 standard. Others include ChemIO (designed for computational Chemistry), Panda, PASSION, ViPIOS and the Jovian parallel IO library. Brief details of these can be found in [5, and references therein]

MPI-IO

For this study we chose to use MPI-IO as the most commonly available parallel IO library. MPI-IO is implemented as a set of routine calls which follow the ethos established in MPI. There are bindings to several languages including C, Fortran and C++. We used the Fortran binding. We shall discuss below the basic functionality available in MPI-IO. For more details, see [6]

From within an application it is necessary to be able to open, close and move around (seek) a file. MPI-IO provides routines to do all of these: MPI_File_open, MPI_File_close and MPI_File_seek. The difference between these routines and the ordinary operations is that they operate in parallel. Thus a normal Fortran OPEN statement would open one file per process. On some systems these might be the same file, particularly if the filesystem was NFS mounted, on others with local disks they might be different, particularly if the file was opened in /tmp (often processors will each have their own /tmp directory). If the file is being read, there is no great problem, but if the processes are writing to the file, confusion can reign as there is no way of knowing in which order they write to the file. MPI_File_open associates the same file with each process in the communication group that calls it. The routine returns a file handle (in Fortran this is an integer, in C it has type MPI_File) and this is passed to subsequent routines to direct I/O.

Data in MPI-IO is organized in units called etypes and filetypes. This is possibly the most important concept, together with the file view, that must be understood when programming in MPI-IO, as it

determines how calls to the basic MPI routines are interpreted. Let us start with an etype, which is an MPI basic type (such as MPI_Real, MPI_Int, etc.) or an MPI derived type created using one of the MPI_Type routines. A filetype is similarly an MPI basic or derived datatype, but it must be built of (possibly multiple instances of) a single etype. Thus a valid filetype could be built from five MPI_Ints, for example, but not from three MPI_Ints and two MPI_Reals. The file view consists of a triplet of entities, an etype, a filetype built from that etype and a displacement. The displacement states how many bytes are to be skipped from the beginning of the file. It must be specified as a special type called in C an MPI_Offset. In Fortran90 it is declared as INTEGER (KIND=MPI_OFFSET_KIND). In most implementations this will ensure that it is allocated 8 bytes and can therefore address the large files in which we are interested.

Also declared as MPI_Offset are the file pointer locations and explicit offsets passed to functions which use them. In these cases, however, the offset is determined in etypes, not in bytes. When a file is first opened the displacement is 0 and the etype and filetype are both MPI_Byte. We found it sensible for our application to retain this and perform all offset calculations in bytes. It is important that variables used as MPI_Offsets are declared as such and not as simple INTEGERS. If a four byte integer is passed, MPI will take the next four bytes as well and interpret all eight as an offset. When writing data, this can lead to enormous files, most of which are empty space. The results may also vary wildly depending on what is in the next four bytes. This also applies to constants passed in – either declare a variable and assign it rather than putting a constant in the calling sequence or pass the constant explicitly cast (in Fortran this can be achieved by using the underscore notation 1234_MPI_OFFSET_KIND).

Calls to the basic I/O functions or read and write come in several varieties. They can be collective or non-collective. In collective I/O it is asserted that all processes will access the file simultaneously – the underlying I/O system can thus optimise the way data is processed and moved around. The calls for collective operations end in the word “all”, as in MPI_File_write_all. For non-collective I/O the processes are treated as independent and no such optimization is possible These are the simple calls such as MPI_File_write. These calls are blocking, the process calling them can do nothing until they have finished the I/O operation. Non-blocking versions also exist, characterised by the use of “iwrite” and “iread” in their names. There is also split collective I/O which is a form of non-blocking I/O for collective calls. To use this the user must call a “begin function”, e.g. MPI_File_read_all_begin, the start the operation and a corresponding “end” function, e.g. MPI_File_read_all_end, to complete it. Between these two calls the program can continue with any computation. A program cannot have two split collective operations active for a particular file handle simultaneously – that is, “begin” and “end” functions cannot be nested.

Finally as we alluded to above, either MPI can keep track of its own position in the file or the programmer can supply the position explicitly. These forms of the routines have “at” in their name, for example, MPI_File_write_at, and an offset in their argument lists.

We have mentioned above the use of defined datatypes as etypes and filetypes. While this is not confined to MPI-IO, it is a useful tool and can greatly simplify the code by treating large areas of memory as single entities to be accessed in one operation. It is particularly useful for accessing arrays distributed across the processes. A darray can be constructed which creates a derived datatype defining the location of the local part of a linearised multidimensional global array for

some common regular distributions. The supported distributions are block, cyclic and general block-cyclic or cyclic(k) distributions as defined in High Performance Fortran (HPF) [7]. Because of the specific details of these distributions, care must be taken when using darrays that the distribution in the program matches exactly. If the distribution does not match, then it may be possible to use the subarray datatype instead. Here the start coordinates in the global array and size in each dimension are given. Note that the start coordinates are always given assuming that the array is indexed from zero, even for Fortran arrays. It is assumed that the local array is an area of contiguous memory.

In many applications the local array will have a ghost or halo region. In a Finite Difference type calculation this will represent nodes of the mesh which belong to neighbouring processes. In this case the active part of the local array will not be contiguous. One solution to this is to create a subarray of a subarray.

Another useful datatype is the vector which has a base type (real, integer, etc.), a blocklength, b, a stride, s and a count, c. Then c groups, each consisting of b contiguous elements of the base type are located with their start positions s apart. There are also generalisations of this to less regular arrangements.

The simplest datatype, which we shall use in our tests, is the contiguous type, which defines a datatype of a number of base types arranged contiguously in memory.

MPI-IO has a host of other features which support the basic functionality described above. Most of these are concerned with setting and interrogating various attributes of files and other I/O systems. An example is MPI_File_get_amode. When a file is OPENed using MPI_File_open, the access mode or amode must be specified. The amode is a combination of constants which include MPI_MODE_RDONLY (read only), MPI_MODE_WRONLY (write only), MPI_MODE_CREATE (if the file does not already exist, create it), MPI_MODE_EXISTS (return an error if the file does not already exist), or'd together in C or added in Fortran.

It is possible to pass hints to the MPI-IO system which might improve performance for a particular implementation or a particular program

As with any parallel program, consistency and synchronisation is an important issue. For consistency purposes we can consider an MPI-IO file as equivalent to a region of shared memory. There are two cases in which consistency does not matter – when all processes are merely reading from a file and when each process is accessing a separate file (this latter is akin to each process having a private portion of the shared memory). If separate processes both read and write a file, there are three ways to ensure consistency. The amode of the file can be set to atomic using MPI_File_set_atomicity. In this mode MPI guarantees that when written by one process, data becomes immediately available for other processes to read. Alternatively the file can be closed and re-opened between writing and reading. Because the MPI_File_close will wait until all processes have finished accessing the file this is a brute force way of synchronising the file. Finally we can explicitly synchronise at various points in the code using MPI_File_sync and MPI_Barrier.

4. Performance Evaluation of MPI-IO

Clearly the speed at which MPI-IO can write and read data is important when deciding whether to

incorporate it in an application. There have been studies of the raw speed of data transfer, such as Breitmoser [8], and Holden, Breitmoser and Hein [9]. While these are useful for identifying strategies for code optimisation, they can fail to represent the complexity of I/O in an application code. Several studies of the use of MPI-IO in applications have been done using the FLASH code, e.g. Ross, Nurmi, Cheng and Zingale [10], but these use MPI-IO as a parallel layer under either HDF or NetCDF APIs. There is little readily available work on the use of “raw” MPI-IO in applications, and we have found none where parallel I/O through MPI-IO is compared with traditional I/O solutions.

We wish to explore the ease of use of MPI-IO in a high performance application, i.e. one designed to use in excess of 32 processors with copious amounts of memory, and to assess the impact of MPI-IO on the code's performance. We would like to know how the use of MPI-IO affects the speed of the code, measured by time to completion for a particular problem. The use of MPI-IO should not adversely affect the code's other functionality, and preferably should enhance it, making possible the solution of larger or more complex problems.

5. POLCOMS

The POLCOMS code simulates the hydrodynamics of the marine environment, coupled with the effects on and of temperature, salinity, etc. It also can be used to predict biological production and pollution transport. The code solves the incompressible, hydrostatic Boussinesq equations of motion. The velocities are written as a depth independent part plus a depth varying part, which allows time-splitting. The variables are placed on a staggered grid, the vector quantities such as the velocities living on a mesh displaced half a step to the south-west from the mesh for the scalar quantities. The mesh is two plus one dimensional, the two dimensional mesh spanning the geographical surface (and note that the systems the code simulates are large enough that the curvature of the surface must be taken into account) while the remaining dimension spans the depth of the water column from sea-bed to surface (actually just below the sea-bed to just above the surface to allow for the imposition of boundary conditions). The vertical mesh is also staggered.

The code has been parallelised by using a two-dimensional partitioning of the domain in the horizontal plane. The initial domain is a regular rectangular grid. Because of the presence of land masses, where no calculations need be done, a simple partitioning can lead to gross load imbalance. In the extreme case partitions could fall entirely within land masses and their associated processes do no work at all. To overcome this, a recursive bisection algorithm is used which divides alternately in x (E-W) and y (N-S) directions, keeping the number of sea points in each domain as equal as possible. As a result the domains a) have different numbers of grid points (since they are themselves defined as rectangles) and b) do not cover the whole of the original (global) mesh, though they only leave out parts which are definitively land. Each domain has a halo associated with it of nodes which belong to another domain. The halo nodes are updated during boundary exchange with the neighbours in the usual fashion

The main quantities which the code outputs as a result of its calculations are the surface elevation, velocities in the east-west and north-south directions, temperature and salinity. Also the turbulence parameters are output such as the eddy viscosity and diffusivity, and the turbulent kinetic energy. Finally, the sediment concentration can be written. There is the facility to write these as time series,

for subsequent generation of moving images.

When a long calculation is being performed, which may take several hours or even days to complete, it can be well worth while pausing every so often to save the state of the program in a form from which it could be restarted. This checkpointing or restart facility usually stores more than the basic output variables to save re-computing the additional data on restart. In POLCOMS we can store the following items:

- Salinity
- Temperature
- Velocity at the current and previous timesteps
- Sea surface height (as deviation from the mean surface height)
- Bathymetric depth (from mean height)
- Eddy viscosity and diffusivity
- Mixing length
- Turbulent Kinetic Energy
- Depth-mean velocities
- Turbulent components of velocities
- Surface and bottom stresses
- Vertical velocities

6. Checkpointing

To test the use of MPI-IO in POLCOMS, we chose a medium sized problem, the Medium Resolution Continental Shelf (MRCS), modelling the continental shelf around the British Isles with an domain covering the north-west European shelf seas at a resolution of 1/10 degree by 1/15 degree. This results in an underlying grid of 251 by 206 by 20. This in turn leads to an active grid of 33514 wet points, each with 20 vertical levels. The number of points in a sub-domain varies as the partitioning algorithm tries to keep the number of wet points in each sub-domain as equal as possible. For 32 processors, the sub-domains have between 684 and 747 wet points, and between 702 and 2898 points, both wet and dry. We ran the model for 15 simulated days, writing a checkpoint file every 24 hours. Timings from five runs were averaged, and the average time taken to run the same problem without checkpointing was subtracted to give a time for the checkpointing I/O phase. The simulation was performed using 32, 64 and 128 processors.

The experiments were all run on HPCx, an IBM Regatta machine. The main compute service of this consists of 96 nodes or nodes, each with 16 POWER5 processors tightly coupled in a hierarchical structure (2 processors to a chip, 4 chips to a module, 2 modules to a node).

Gather-write / read-scatter

The original version of POLCOMS handles its checkpoint file using the gather-write / read-scatter model. That is to say that on the master processor a global array is allocated, when writing the

partitioned arrays are sent from each slave processor and received by the master, where they are stored in the appropriate locations in the global array. When this is complete, the entire array is written to an unformatted Fortran file. The global array is then deallocated. For reading the process is reversed. The global array is allocated and read from the unformatted file. Then the master sends portions of the global file to the slave processors where they are received into the local partitioned arrays.

The data model that underlies this is a flat file of global arrays (headed by a single real containing the time at the checkpoint) in a set order which must be adhered to in both writing and reading routines. The average and standard deviation of the times to write the arrays is shown in Table 1:

Number of Processors	***I/O time***	***Error***
32	33.87	3.01
64	44.37	8.48
128	53.66	10.83

Table 1: Average times to write checkpoint data in gather-write mode

The increased variability seen as the number of processors increases is likely due to using increased numbers of nodes, leading to more inter-node communication. As nodes are filled and reserved exclusively for one job by the job scheduler, communication between processors on the same node is relatively independent of what is happening on the rest of the machine. Inter-node communication, on the other hand, has to compete for bandwidth with other jobs, and thus can be quite variable.

Having established a baseline for I/O using the original version we then implemented a variety of MPI-IO strategies.

Partition Stacking

The first version we produced packed the partitioned arrays as compactly as possible. Each process calculated the size of the (local) arrays in the processes of lower rank than itself, and then started writing its own local array from the next available position. Because of the different sizes of these arrays of different processors, we were not able to define a datatype to use as an etype in the file view. We therefore chose to use the routines which explicitly address the file by passing in an offset defining where to write or read (the basic routines are MPI_FILE_WRITE_AT and MPI_FILE_READ_AT). The default file view was used so that all offset calculations were carried out in bytes.

Number of Processors	***I/O time (standard)***	***Error***	***I/O time (MPI-IO)***	***Error***
32	33.87	3.01	367.46	23.18

Number of Processors	*I/O time (standard)*	*Error*	*I/O time (MPI-IO)*	*Error*
64	44.37	8.48	358.82	37.81
128	53.66	10.83	375.34	22.97

Table 2: Average times to write checkpoint data in simple MPI-IO mode (non-collective)

These results, shown in Table 2, are (to say the least) disappointing. They represent a worsening of I/O times by factors of 10.8, 8.1 and 7.0 for 32, 46 and 128 processors respectively. However, this uses the non-collective calls. Collective calls are said to be more efficient, since MPI can make some optimizations. We therefore changed the calls from MPI_FILE_WRITE_AT to MPI_FILE_WRITE_AT_ALL, its collective counterpart.

Number of Processors	*I/O time (standard)*	*Error*	*I/O time (Collective MPI-IO)*	*Error*
32	33.87	3.01	119.03	19.7
64	44.37	8.48	149.71	14.08
128	53.66	10.83	208.63	22.38

Table 3: Average times to write checkpoint data in simple MPI-IO mode (collective)

Now the ratios are 3.5, 3.4 and 3.9, a dramatic improvement over the non-collective version. Profiling the new version with Vampir (now known as the Intel Trace Analyzer) [11] showed that much of the time was being absorbed in synchronisation stages between each array. These had been included following the model of example programs, but further scrutiny showed that they were unnecessary, especially for collective calls. Removing the synchronisation steps resulted in the times shown in Table 4:

Number of Processors	*I/O time (standard)*	*Error*	*I/O time (non-collective MPI-IO no sync)*	*Error*	*I/O time (collective MPI-IO no sync)*	*Error*
32	33.87	3.01	121.29	38.69	36.67	9.59
64	44.37	8.48	135.44	36.73	56.13	8.57
128	53.66	10.83	165.41	31.64	123.95	23.23

Table 4: Average times to write checkpoint data in simple MPI-IO mode (collective, no synchronisation)

We can see that the non-collective code is now as good as or better than the collective code with synchronisation. Even more importantly, the collective version is comparable with the original method, at least at low processor numbers, though its behaviour as the number of processors increases is not as good. Since the main advantage of using parallel I/O, the avoidance of gathering a large amount of data onto the master processor with the large memory requirement that places on the code, is most effective for massively parallel runs, this loss of performance is disappointing.

Partition Steering

While this model is conceptually simple it has the drawback that it is only possible to restart the simulation with the same number of partitions as the simulation that wrote the checkpoint file. It may be that a reason for the simulation needing to be restarted is that a different partitioning is needed for efficiency, so this model would not be appropriate for this. We therefore needed to find a way within MPI-IO to write an image of the global array from the local arrays. Each processor knows the start addresses of its local arrays within the global array. Thus, say, processor 17 knows that its array, which is of size IESUB by JESUB starts at (IELB, JELB) in the global array. It also knows that the global array has size (L, M). The position is illustrated in the figure below, where the mapping of array locations to memory is assumed to run first horizontally, then vertically downwards. Also shown with vertical hatching is a halo region to the local array, which we do not wish to write, both to avoid redundancy and to avoid contention with two or more processes trying to write to the same part of the file. The union of the vertically and horizontally hatched regions is contiguous in memory on one of the processors; we wish to perform the write such that the global array is contiguous on disk. (In practice, because of the way the grid is partitioned and because dry regions are excluded, not all locations in the file will be written.)

With this version, which we call Partition Steering since the calculation of offsets steers the data into the correct location in the file, we can use the contiguous datatype defined by the MPI call to MPI_Type_contiguous and write a row of the array at a time. This means that the call to the MPI_File_write_* routine must be contained within a loop which has some performance implications, but the results in Table 5 show these are not significant.

Number of Processors	***I/O time (standard)***	***Error***	***I/O time (steered MPI-IO, non-collective)***	***Error***	***I/O time (steered MPI-IO, collective)***	***Error***
32	33.87	3.01	145.2	11.11	34.78	5.83
64	44.37	8.48	126.82	11.15	45.43	11.81
128	53.66	10.83	163.23	37.39	60.82	11.78

Table 5 Average times to write checkpoint data in partition stacking MPI-IO mode (no synchronisation)

In this case there is no improvement with the non-collective calls over the unsteered version, but the collective steered version is operating more or less at parity with the original code. Given the ability to checkpoint much larger problems with the parallel method, this represents a step forward for the

capability of the POLCOMS code. There remains the question of the scaling behaviour of the two versions. Consider an amount of data, A, distributed evenly among N processors. Let T_c be the time to send one element of this data between two processors and T_{lat} the time to initiate a message. Thus the time for a message of length B is $T_{lat} + B * T_c$. When MPI does a gather operation it can do so in several ways. Early implementations simply queued all messages onto the master process. In this case the time to send the data was $A * (N-1)/N * T_c + (N-1) * T_{lat}$, there being N-1 messages to send. For large N the dominant term is O(N). It is possible to arrange the gather operation in a binary tree so that several communications can be performed simultaneously. In this case we can write the time taken as:

$$T_{total} = \sum_{i=0}^{\log_2(N)-1} \left[\frac{A}{N} 2^i T_c + T_{lat} \right] = \frac{A}{N} T_c \frac{\left(1-2^{\left(\log_2(N)\right)}\right)}{(1-2)} + T_{lat} \log_2(N) = A \frac{(N-1)}{N} T_c + T_{lat} \log_2(N)$$

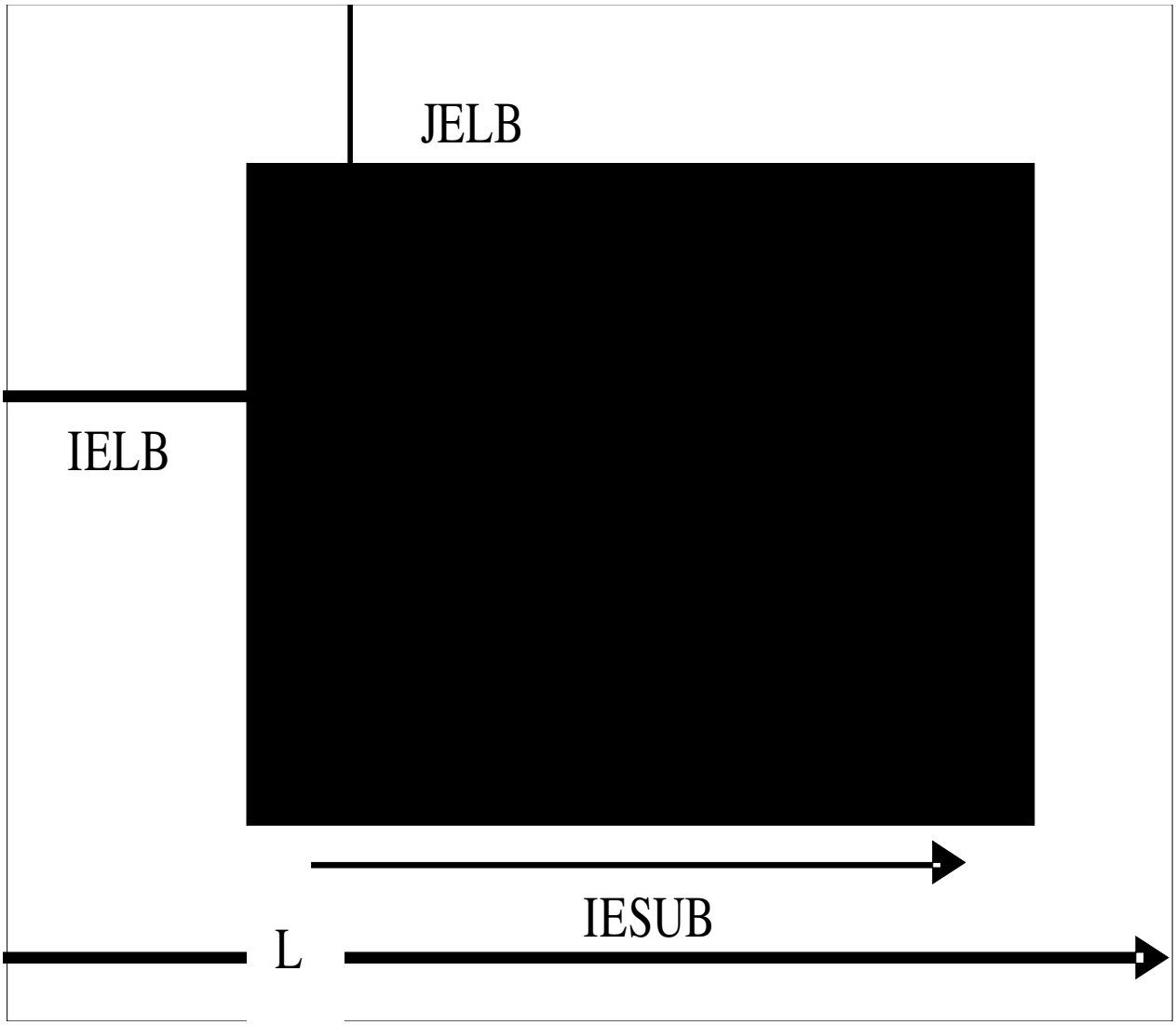

Figure 1 A local array (horizontal hatching) with its halo region (vertical hatching) shown in position within the global array.

so that the dominant term is now $O(\log_2 N)$ The time to actually move the data remains the same, since the same amount of data needs to get to the master process – the saving is achieved by having fewer sequential messages, though the individual messages are larger. This is of most concern in a

high latency environment.

For the MPI-IO case, if we had a perfect parallel file system so that there was no contention for resources and no blocking of messages, and if we could write all the local data at once, then the time to write the data would be $A/N\ T_{write} + T_{open}$ where T_{write} is the time to write one item of data and T_{open} is the time to initiate the I/O operation. However, we are writing the local arrays a row at a time, so we are performing $(A/N)^{1/2}$ writes, each of $(A/N)^{1/2}$ items of data. Thus the time becomes $A/N\ T_{write} + (A/N)^{1/2}\ T_{open}$. This is not borne out by the results above where the time increases as N does, rather than decreasing, suggesting that MPI-IO is doing something akin to a message gathering. In the worst case, where the individual writes are queued onto a single bottleneck, the time will become $A\ T_{write} + (AN)^{1/2}\ T_{open}$. The results of Table 5 for collective calls are a good fit to this model, with $A\ T_{write} = 8.74$s and $A^{1/2}\ T_{open} = 4.6$s.

8. The Prospects for Large Problems

We have shown that with care, parallel I/O can be implemented for the checkpointing phase of the POLCOMS code. This gives increased capability of the code, in that larger problems become tractable in principle, with little detriment to performance. However, it is not the whole answer. The problem size that can be run is still limited by the same criteria, as all final I/O is performed using the gather-write model. Since in this case many of the files are formatted ASCII files required to be read by other programs (e.g. for visualisation), MPI-IO is not appropriate as a long term solution, though it could be used with an intermediate translation program.

Instead the ultimate aim is to move to NetCDF for interoperability with other codes. Consideration should also be given to the use of pNetCDF, a parallel version which it is hoped would give the same improvements as demonstrated here with MPI-IO.

9. Conclusions

We have implemented a parallel input/output strategy in the checkpointing and restart phases of POLCOMS, an oceanographic model. Two representations of the distributed data have been used, referred to as partition stacking and steered, and the effect of using collective and non-collective parallel I/O has been investigated. In partition stacking the local arrays which make up the global data are arranged contiguously in the file, and each process works out where in the file to start by calculating how much data the processes of lower rank than itself will be writing. In steered I/O the data is written to produce an image of the global array in the file. This means that the local array is no longer arranged contiguously within the file and must be written row by row. This is more complicated to implement, but the advantage is that it is then possible to restart a computation with a different number of processors from that which wrote the checkpoint file.

For this case there was seen to be a clear advantage to using collective I/O calls. By using these, the MPI subsystem is aware that all processors will be accessing the file and can make optimisations based on that. It was also clear that synchronisation could form a major bottleneck, and since this was unnecessary with collective calls, this made their use preferable. It would, however, be unwise to extrapolate from this fairly simple example to suggest that unsynchronised collective I/O would always be the preferred option. Some codes, particularly codes which use parallel I/O to operate

out-of-core, may require explicit synchronisation to operate correctly.

In summary, the use of steered I/O with collective MPI calls gave comparable performance to the original version of the POLCOMS code which used a gather-write / read-scatter model. With the ability to write larger problem sizes by using parallel I/O, it represents a possible way forward to enhance the capability of POLCOMS.

References

1. GPFS Primer for AIX clusters, IBM, http://www.ibm.com/servers/eserver/pseries/software/whitepapers/gpfs_primer.pdf (accessed 20/July/2006)
2. XFS: A high-performance journaling filesystem, SGI, http://oss.sgi.com/projects/xfs/index.html (accessed 20/July/2006)
3. SUN PFS Whitepaper, http://whitepapers.zdnet.co.uk/0,39025945,60011890p-39000591q,00.htm (accessed 20/July/2006)
4. Lustre scalable storage, Cluster File Systems Inc., http://www.clusterfs.com/index.html (accessed 20/July/2006)
5. Greenough C., Fowler R.F. and Allan R.J., Parallel IO for High Performance Computing, http://www.sesp.cse.clrc.ac.uk/Publications/paraio.pdf (accessed 20/July/2006)
6. Gropp W., Lusk E. and Thakur R., Using MPI-2, Advanced Features of the Message-Passing Interface. MIT Press, Cambridge Mass. and London England (1999)
7. The High Performance Fortran Forum, High Performance Fortran Language Specification, http://hpff.rice.edu/versions/hpf2/hpf-v20.pdf (accessed 4/Oct/2006)
8. Breitmoser E. Investigating MPI-IO on HPCx, HPCx Technical Report HPCxTR0304 (2003) http://www.hpcx.ac.uk/research/hpc/technical_reports/HPCxTR0304.pdf (accessed 27/July/2006)
9. Holden M., Breitmoser E and Hein J., I/O Performance on the HPCx Phase 2 System, Technical Report HPCxTR0504 (2005) http://www.hpcx.ac.uk/research/hpc/technical_reports/HPCxTR0504.pdf (accessed 27/July/2006)
10. Ross R., Nurmi D., Cheng A. and Zingale M. A Case Study in Application I/O on Linux Clusters, SC2001. http://www.sc2001.org/papers/pap.pap166.pdf (accessed 27/July/2006)
11. Intel Trace Analyzer and Collector6.0 Features. http://www.intel.com/cd/software/products/asmo-na/eng/cluster/tanalyzer/231038.htm?prn=y (accessed 22/Aug/2006)