Parallel IO support for Meta-Computing Applications: MPI_Connect IO applied to PACX-MPI

Graham E. Fagg+, Edgar Gabriel*, Michael Resch* and Jack J. Dongarra+

+Department of Computer Science, University of Tennessee, Suite 413, 1122 Volunteer Blvd, Knoxville, TN-37996-3405, USA.

*High Performance Computing Center Stuttgart, Allmandring 30, D-70569 Stuttgart, Germany

fagg@cs.utk.edu
gabriel@hlrs.de

Abstract. Parallel IO (PIO) support for larger scale computing is becoming more important as application developers better understand its importance in reducing overall execution time by avoiding IO overheads. This situation has been made more critical as processor speed and overall system size has increased at a far greater rate than sequential IO performance. Systems such as MPI_Connect and PACX-MPI allow multiple MPPs to be interconnected, complicating IO issues further. MPI_Connect implemented Parallel IO support for distributed applications in the MPI_Conn_IO package by transferring complete sections of files to remote machines, supporting the case that all the applications and the file storage were completely distributed. This system had a number of performance drawbacks compared to the more common usage of metacomputing where some files and applications have an affinity to a home site and thus less data transfer is required. Here we present the new PACX-MPI PIO system based initially on MPI_Connect IO, and attempt to demonstrate multiple methods of handling MPI PIO that cover a greater number of possible usage scenarios. Given are some preliminary performance results as well as a comparison to other PIO grid systems such as the Asian Pacific GridFarm, and Globus gridFTP, GASS and RIO.

1. Introduction

Although MPI [13] is currently the de-facto standard system used to build high performance applications for both clusters and dedicated MPP systems, it is not without some issues. Initially MPI was designed to allow for very high efficiency and thus performance on a number of early 1990s MPPs, that at the time had limited OS runtime support. This led to the current MPI-1 design of a static process model. While this model was possible to implement for MPP vendors, easy to program for, and more importantly something that could be agreed upon by a standards committee. During the late 90s a number of systems similar to MPI_Connect [1] and PACX-

MPI [3] were developed that allowed multiple MPPs running MPI applications to be interconnected while allowing for internal MPP communications to take advantage of vendor tuned software rather than using a TCP socket based portable implementation only.

While MPI_Connect and PACX-MPI solved some of the problems of executing application across multiple platforms they did not solve the problem of file distribution and collection or handling multiple specialized file subsystems. Initially this was not a problem as many parallel applications were developed from sequential versions and they continued to utilize sequential IO operations. In many cases only the root processor would perform IO. This state of affairs was compounded by the initial version of MPI not having any Parallel IO API calls, thus forcing users of Parallel IO to use non portable proprietary interfaces. The MPI-2 standard did however include some Parallel IO operations, which were quickly implemented by a number of vendors. The work presented here is built on the MPI-2 Parallel IO interface and the appication supported are expected to use the MPI-2 Parallel IO interface to maintain portability.

1.1 MPI-2 Parallel IO

Parallel I/O can be described as multiple access of multiple processes to the same, shared file. The goal of using parallel File I/O within MPI applications is to improve the performance of reading or writing data, since I/O is often a bottleneck, especially for applications, which are frequently check-pointing intermediate results. With the MPI-2 standard, an interface is designed which enables the portable writing of code, which makes use of parallel I/O.

MPI–I/O provides routines for manipulating files and for accessing data. The routines for file manipulation include opening and closing files, as well as deleting and resizing files. Most file manipulation routines have to provide as an argument an MPI communicator, thus these operations are collective.

For reading or writing data, the user has several options. MPI-I/O gives the application developer the possibility to work either with explicit offsets, with individual file pointers or with shared file pointers. The operations are implemented as blocking and non-blocking routines. Additionally, most routines are available in both collective and non-collective versions. Taking it all together, the user may choose between more than 28 routines, which fits best to his applications IO access pattern.

1.2 MPI_Connect and MPI_Conn_IO

MPI_Connect is a software package that allows heterogeneous parallel applications running on different Massively Parallel Processors (MPPs) to interoperate and share data, thus creating Meta-applications. The software package is designed to support applications running under vendor MPI implementations and allow interconnection between these implementations by using the same MPI send and receive calls as developers already use within their own applications. This precludes users

from learning a new method for interoperating, as the syntax for sending data between applications is identical to what they already use. The support of vendor MPI implementations means that internal communications occur via the optimized vendor versions and thus incur no performance degradation as opposed to using only TCP/IP, the only other option previously available. Unlike systems such as PACX-MPI, each application maintains its own MPI_COMM_WORLD communicator, and this allows each application to connect and disconnect at will with other MPI applications as shown in figure 1.

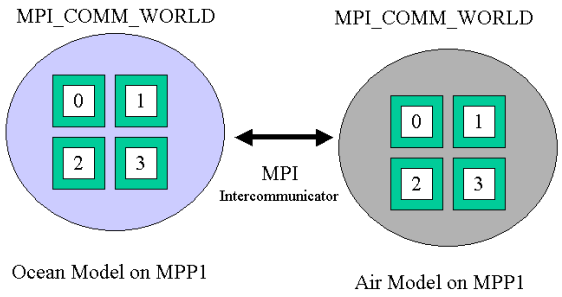

Fig 1. MPI_Connect overview

The MPI_Conn_IO API was developed to support multi-site multi-application execution using shared files. The three primary requirements were to provide:

(A) Local parallel file system access rates
(B) Global single name space naming for all files
(C) Single points of storage

To provide (A) the system had to utilize the MPI-2 PIO API for file access. To provide (B) which is important for GRID applications where location independence is required to handle uncertainties caused by runtime scheduling, a dedicated naming/resolution service was created. (C) single point of storage is where the input and output files are stored in well known locations as known by the naming service so that users do not have to distribute their files at every location where an execution is possible. This required the use of dedicated file serving daemons. An overview of the MPI_Conn_IO system is given in figure 2.

To allow applications the ability to handle this distributed access to Parallel IO the users had to add two additional library calls, one to gain access to the global/shared file and the other to release it. The call to access the file was one of two variations:
(A) MPI_Conn_getfile(globalfname, localfname, outsize)

(B) MPI_Conn_getfile_view(globalfname, localfname, my_app, num_apps, dtype, outsize, comm)

Both calls would access the naming/resolution service to locate the file server with the required file. Variant (A) would then copy the complete file onto all the systems in the MPI_Connect system. The individual applications themselves would then have to handle the portioning of the file so that they each accessed the correct sections of data. In variant (B) the call is a combination of opening a file and setting the access pattern in the terms of access derived data types and relative numbers of nodes in each part of the MetaApplication. This allows the file daemons to partition the files at the source so that file distribution and transmission costs are reduced. For example, if two applications of sizes 96 and 32 nodes opened the same file with the first setting *my_app*=0, *num_apps*=2 and *dtype* to MPI_DOUBLE, and the second opened the file with *my_app*=1, *num_apps*=2 and *dtype* = MPI_DOUBLE then the file would get stripped between the two sites. The first site would get the first 96 doubles with the second site getting then next 32 doubles and so on. The files on the local Parallel IO subsystems would be named by the user supplied *localfname* parameter.

Once the file has been globally opened it can then be opened via the standard MPI-2 Parallel IO call MPI_File_open (com, *localfname*, mode, info, fhandle).

After the application has finished with the file, the application needs to release its references to the global version of the file via the MPI_Conn_releasefile API call. If the file was created or modified locally its globally stored version would be updated by this call.

To further reduce the time it takes to transfer a file, the TCP/IP routing of the files contents can be explicitly controlled by store and forwarding through a number of IBP [15] network caches at various locations throughout the network. This scheme is used to avoid known network bottlenecks at peak times.

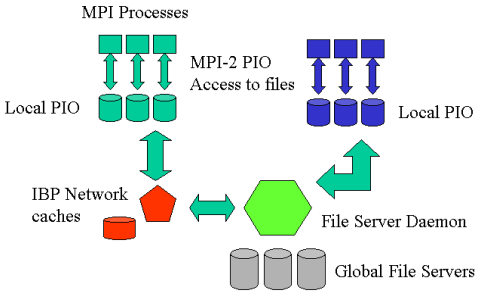

Fig 2. MPI_Conn_IO overview

1.2 PACX-MPI

PACX-MPI is an implementation of the MPI standard optimized for clustered wide-area systems. The concept of PACX-MPI relies on three major ideas: First, in clustered systems, the library has to deal with two different levels of quality of communication. Therefore, PACX-MPI uses two completely independent layers to handle operations on both communication subsystems. Second, for communication between processes on the same machine, the library makes use of the vendor-MPI, since this allows it to fully exploit the capacity of the underlying communication subsystem in a portable manner. And third, for communication between processes on different machines, PACX-MPI introduces two communication daemons. These daemons allow buffering and packing of multiple wide area communications as well as handling security issues centrally.

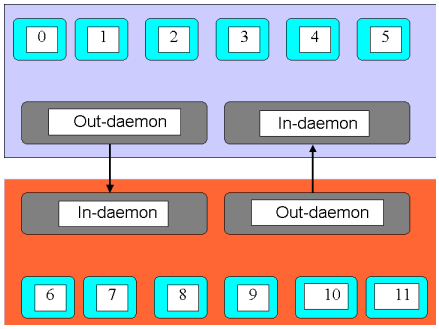

Fig 3. Concept of PACX-MPI

2. Distributed and Grid Parallel IO issues

The requirements and implementation issues of Distributed and Grid based file systems overlap heavily. Although the topic is very complex and broad it can be simplified to a number of key issues.

Distributed File Systems (DFS) such as AFS [12] primarily focus on performance through mechanisms such as caching, and specialized file locking techniques at the OS layer and are typically aimed at supporting sequential (POSIX style) access patterns across a wide area network rather than an unreliable global one. They assume that restrictions such as single user IDs and a single coherent file name space.

Grid based tools are primarily aimed at simplified transmission of file data, where the actual data IO is done locally from some temporary storage location. Here the issues become those of single point authentication, efficient handling of different backend file services, mapping of multiple user Ids, and handling the naming and resolution of multiple replicated copies of data [6][11]. The replicated issue becom-

ing more important as more systems utilize caching and pre-staging of data to reduce access latencies to remote storage. Systems such as gridFTP and Global Access to Secondary Storage (GASS) [8] exemplify this. GridFTP provides an FTP abstraction that hides multiple access, transmission and authentication methods, and GASS which provides a simple POSIX file IO style API for accessing remote data using URL naming. GASS then utilizes local caching to improve repeated IO access rates. One advantage of GASS over AFS is that the caching is user configurable via an exposed cache management API.

PACX-MPI PIO aims to support functionality in-between these two levels and is very close to systems such as RIO [9], Stampi [4] and the Grid Data Farm [10]. RIO was aimed at supporting MPI Parallel IO access patterns via a server client model. Unlike traditional server client models, the clients are parallel applications and the servers can directly access multiple parallel IO file servers, thus ensuring that performance of the parallel file system was passed onto the client side. The implementation contained forwarder nodes on the client side that translate local IO access requests and pass these via the network to the server side which then accesses the real file servers in parallel. The Grid Data Farm is a Petascale data-intensive computing project initiated in Japan as part of the Asian Pacific Grid. In common with RIO it has specialized file server daemons (known as gfsd's) but it supports a specialized API that uses a RPC mechanism much like that of NFS for actual file access by remote processes. This was needed as it did not directly support the MPI API. An interesting feature of the system is that the file server nodes can also perform computation as well as IO, in which case they attempt to process only on local data, rather than more expensive to access remote data.

MPI_Conn_IO is equivalent to a mix of GASS and RIO. It has additional calls to access remote files, which in effect are pre-staging calls that cache the complete requested data fully onto a local Parallel File System via MPI-IO operations. It has special file handling daemons like the RIO server processes, although it does not support their on demand access operation.

The target implementation of PACX-MPI is hoped to be a system that allows for a mode of operation anywhere between the GASS/MPI_Conn_IO fully cached method and the RIO on demand method. The authors believe that there are many different styles of access pattern to file data within parallel and meta/grid applications and that only by supporting a range of methods can reasonable performance be obtained for a wide range of applications [2].

3. PACX-MPI PIO aims

The current aims of PACX-MPI-PIO are:

(A) Support a single MPI-2 Parallel IO file view across any MPI communicator (assuming it is distributed across multiple MPPs)

(B) Allow access to files via the MPI-2 API using the efficient vendor supplied Parallel IO libraries where possible

(C) Do not force the application developer to use any additional API calls unless they are in the form of performance hints
(D) Reduce any file management to a minimum
(E) Support a number of caching and transmission operations

As PACX allows communicators (A) to be across multiple machines the IO layer has to move file data between sites automatically if they are to support reduced file management (D). A consequence of (D) is that one of the execution sites is considered the 'home' site where the data files will reside but we will not restrict all files to be at the same home site. Allowing PACX-MPI-PIO to work without forcing the use of any additional calls (unlike MPI_Conn_IO) requires the MPI-2 file PIO calls to be profiled using the MPI profiling interface so that the file movement is hidden from the users application. (B) forces the profiled library to call the MPI-2 PIO API directly. (C) and (E) mean that any application level requests to change the system transfer options and caching methods should be in the form of MPI attributes calls, or even compile time flags.

To simplify initial development a number of restrictions are placed on the user applications use of the MPI Parallel IO API. Firstly, the applications open files for reading, writing or appending, but not for mixed read and write operations. Secondly once a file has been opened, it can set its file view but not modify this file view without reopening the file. This restriction maybe removed in a future version if there is a need, but currently the authors have not found any application that need this functionality. Lastly, shared file pointers between multiple MPPs are not supported.

3.1 PACX-MPI PIO file manipulation and transmission modes

Manipulation of files and transmission of file data can be broken into a number of issues.

(A) Naming. As the target files exist at one of the execution sites, the need for any global naming scheme is negated and the local names can be used instead. The requirement is that users specify which site is the home site to the PACX system. This can be done via an entry in the PACX host file, or over ridden by an MPI attribute call. Any data stored in a cache at another site uses a one time temporary name.
(B) Caching of data. Is the complete file replicated across all MPPs, or just sections of it. Is the data stored on the local PIO system or purely in memory.
(C) Who does the remote access? Are additional daemons required or can the PACX communication daemons do the work.

The initial test implementation of PACX-MPI PIO only supports two modes of remote file access with various options:

1. Direct access
2. Proxy access

3.2 PACX-MPI PIO Direct access

Direct access is where the data on the remote (i.e. non home) site is stored on the PIO system as a data file on disk and the MPI-2 PIO application calls translate directly to MPI-2 PIO library calls. This is just the same as the original MPI_Conn_IO library, except there is not a specialized File server and thus less data needs to be transferred, and there are no API changes or global names needed.

In this mode of operation, when the applications make the collective MPI_File_set_view call, the local (home site) application reads the file data in via parallel MPI IO calls, and then sends the data via the communications daemons to the remote nodes, which then write the data via the MPI IO calls to their local temporary parallel files. The applications can then proceed as normal handling the data with the profiled MPI Parallel IO API calls. The use of MPI IO calls to store the data is important as the authors have found that on some systems such as the Cray T3E, the data is not automatically stripped if just a single processes writes it out and thus performance drops to that of a single disk for IO access.

The data storage and access at the remote sites can be in one of three forms:

1. Complete File Copy (CFC). In this mode all sites have an identical copy of the complete file. This is useful for small files where the cost of moving all or part of the file is negligible compared to the overall execution time. It is also useful for supporting multiple changes to the file view during a single application run. This mode of distribution is show in figure 4.
2. Partial File Copy (PFC) with full extent. This is where only the data needed on each of the distributed systems is copied to them. This reduced the amount of data to be copied. The file is stored with the same extent as the original copy, i.e. it is sparse storage. This mode is useful as it does not require any remote file view translation allowing for multiple file view support. The only disadvantage is that it requires the same amount of storage at each site as the original file which maybe considerable. An example of this is shown in figure 5.
3. Partial File Copy (PFC) with compacted storage. This is the same as above except that the storage is compacted so that no gaps in the remote files exist as shown in figure 6. This has three consequences. The first is the file uses much less space. The second is that it requires all the file view API calls on the remote machines to be translated so that they correctly address the new data layout. This translation is automatic and only performed once. The third consequence is multiple file views are not supportable, and that the temporary file would have to be completely rebuilt each time the file view was changed.

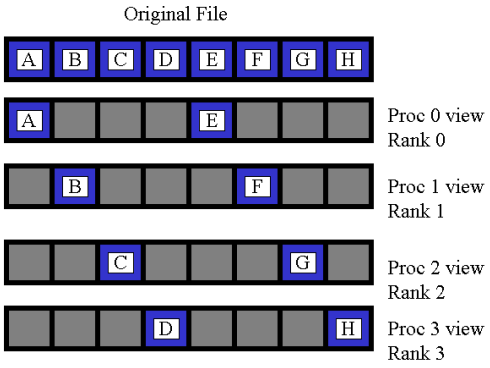

Fig 4. File view for data stripping on MPPs with complete data files

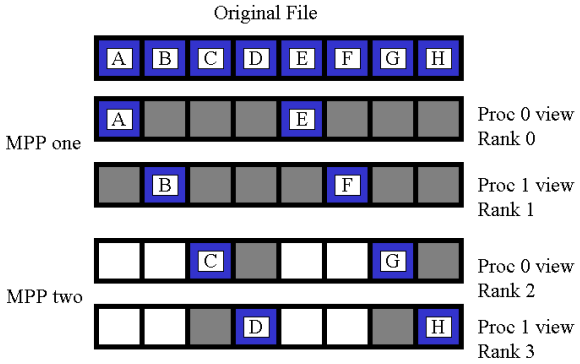

Fig 5. File view for stripping of data across two MPPs with full file extent.

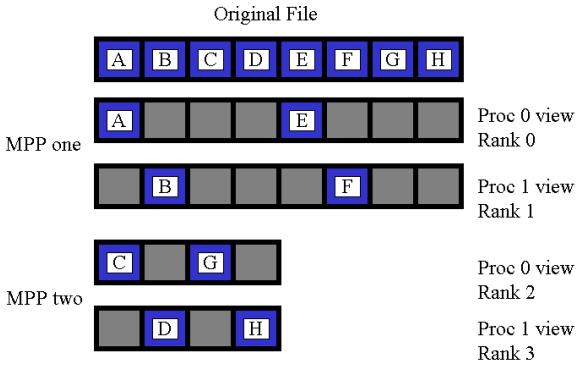

Fig 6. File view for stripping data across two MPPs with compacted file storage (reduced extent)

3.3 PACX-MPI PIO Proxy access

Proxy access is when the file data only exists on the home/local site and all other applications pass their IO access requests to the primary application, which then performs the accesses on their behalf. The overall structure is similar to RIO and the Data Grid Farm as shown in figure 7. Due to its nature it is known as On Demand Access (ODA) as IO calls are made dynamically as demanded by remote processes, or Cached as the remote system reads and writes data from a memory buffer in the form of a message queue. The number and size of pending IO accesses can be varied dynamically. Another feature is that pre-fetching of read requests can be performed as the file view is known in advance, which leads to more efficient latency hiding.

A number of issues exist which effect overall performance. Firstly, the local application has to perform all the IO operations which can reduce the level of parallelism within the Parallel IO subsystem. Secondly the amount of intersystem communication dramatically increases, and even with pre-catching, intersystem bandwidth become a dominating factor.

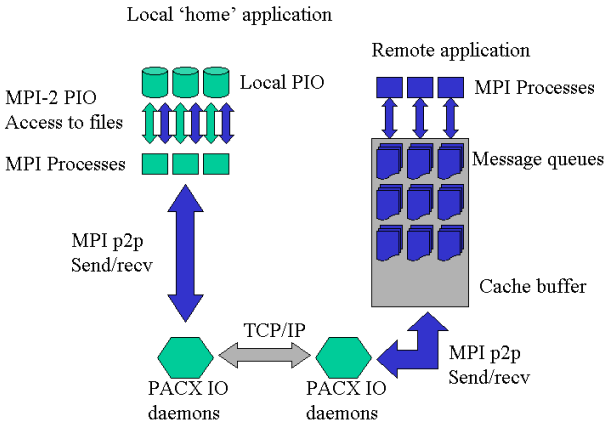

Fig 7. Proxy, On Demand Access (ODA) cached access structure.

4. PACX-MPI IO experiments

Initial experiments were conducted on a Cray T3E-900 with 512 nodes at the University of Stuttgart. Unfortunately the version of MPI-2 PIO installed was based on a version ROMIO that utilized only the POSIX file IO calls and thus the raw PIO performance was poor [5].

We ran several different classes of experiment to characterize the IO subsystem:

(A) Basic Parallel IO benchmarks using only collective blocking calls (MPI_Read/Write_ALL).

(B) Partition to partition TCP performance (to test interconnection speeds)

(C) Single machine, single partition tests

(D) Single machine, multiple partition tests (each partition runs a separately initiated MPI application under PACX-MPI) with a single coherent file view across the whole PACX-MPI job.

Out of the different caching, file transfer and storage methods described in section 3 we present only the partial file copy (PFC) with compacted storage (modified file view) and (ODA) on-demand access (system cached) for the distributed multiple MPI application cases as these best represent the current issues in a final implementation. These are compared against the single application in terms of both aggregate IO bandwidth and application execution time.

The data files used were all over 150 Mbytes to avoid multi-level system caching effects, and from previous experience with real CFD applications [2] we varied the granularity of individual node IO accesses from small/fine, i.e. 100 doubles per node per access to very large/course, i.e. 100,000 doubles per node per IO access.

Figure 8 shows the aggregate IO bandwidths for accessing a large (150+ Mbyte) using the MPI-2 parallel IO API using a total of 32 nodes.

The first line is for reading on a single partition with 32 nodes and peaks at around 25 MB/Second. The second line shows the write bandwidth which is much lower with a peak of only 10.5 MB/Second. The third line shows the bandwidth of accessing the file with two separate 16 node partitions using the PFC method. This method includes all file transfers and its performance is dominated by the slow parallel file write IO operations. The forth line shows twice the interconnection bandwidth. This is show as it is the limit for file transfers. (Twice is shown as we are utilizing the compacted file view so only half the file is transferred and we are measuring the aggregate bandwidth being sum of all individual node bandwidths).

The forth line shows the on demand access (ODA/cached) method which performs much better than the PFC method at around 10.5 MB/Second for large accesses. As the IO performance goes up the effects of interconnection will dominate this method. The effects of latency for this method, can be hidden by increasing the pre-fetch buffer size. This technique becomes infeasible for course grain accesses on larger numbers of nodes due to memory constraints.

Figure 9 shows the single application single partition version compared to PFC and ODA/cached methods in terms of time rather than bandwidth for different granularities of access. Here the degradation in performance of using the PFC version is more apparent especially for smaller accesses, although the ODA version appears to perform remarkably well.

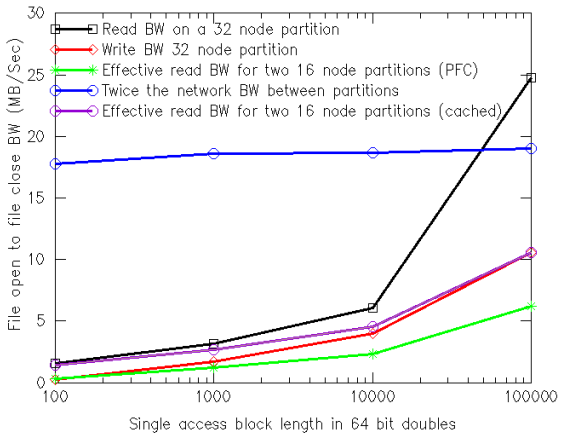

Fig 8. Aggregate Bandwidth of different method as a function of access granularity

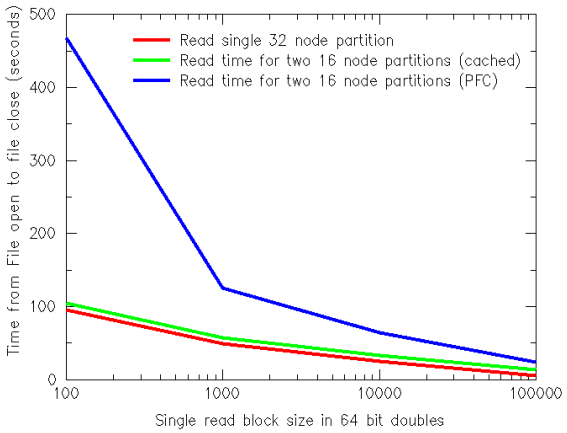

Fig 9. Overall application IO times for various access methods.

5. Conclusions and future work

The PACX-MPI PIO initial implementation appears to be a good vehicle to experiment with various caching, data access and storage mechanisms while supporting the portable use of the MPI-2 Parallel IO API. The PIO system is transparent in use and requires no code changes from that of a standalone application even when executing transparently on multiple MPPs. The system also utilizes the concept of a home site for the storage of files, which reduces the work load of the application users as they no-longer have to handle file staging manually when performing multi-site application runs.

Currently future work is aimed at automatically improving performance, especially for the class of applications that perform regular user directed check-pointing.

6. References

1. G. E. Fagg, K. S. London, and J. J. Dongarra, "MPI_Connect: managing heterogeneous MPI applications interoperation and process control", in V. Alexandrov and J. Dongarra, editors, *Recent advances in Parallel Virtual Machine and Message Passing Interface*, volume 1497 of *Lecture notes of Computer Science*, pages 93-96. Springer, 1998. 5th European PVM/MPI User's Group Meeting.
2. D. Cronk, G. Fagg, and S. Moore, " Parallel I/O for EQM Applications", Department of Computer Science technical report, University of Tenneesee, Knoxville, July 2000.
3. E. Gabriel, M. Resch, T. Beisel and R. Keller "Distributed Computing in a heterogeneous computing environment", in V. Alexandrov and J. Dongarra, editors, *Recent advances in Parallel Virtual Machine and Message Passing Interface*, volume 1497 of *Lecture notes of Computer Science*, pages 180-188. Springer, 1998. 5th European PVM/MPI User's Group Meeting.

4. T. Imamura, Y. Tsujita, H. Koide, and H. Takemiya, "An architecture of Stampi: MPI library on a cluster of parallel computers", in J. Dongarra, P. Kacsuk, and N. Podhorszki, editors, *Recent advances in Parallel Virtual Machine and Message Passing Interface*, volume 1908 of *Lecture notes of Computer Science*, pages 200-207. Springer, 2000. 7th European PVM/MPI User's Group Meeting.
5. Rolf Rabenseifner and Alice E. Koniges, "Effective File-I/O Bandwidth Benchmark", in A. Bode, T. Ludwig, R. Wissmüller, editors, Proceedings of Euro-Par 200, pages 1273-1283, Springer 2000.
6. B. Allock et all, "Secure, Efficient Data Transport and Replica Management for High-Performance Data-Intensive Computing", submitted to IEEE Mass Storage Conference, April 2001.
7. R. Thakur, W. Gropp, and E. Lusk, "On Implementing MPI-IO Portably and with High Performance", in *Proc. Of the Sixth Workshop on I/O in Parallel and Distributed Systems*, May 1999, pages 23-32.
8. J. Bester, I. Foster, C. Kesselmann, J. Tedesco, S. Tuecke, "GASS: A Data Movement and Access Service for Wide Area Computing Systems", in *Sixth Workshop on I/O in Parallel and Distributed Systems, 1999.*
9. I. Foster, D. Kohr, R. Krishnaiyer, J. Mogill, "Remote I/O: Fast Access to Distant Storage", in *Proc. Workshop on I/O in Parallel and Distributed Systems, (IOPADS)*, pages 14-25, 1997.
10. O. Tatebe et all., "Grid Data Fram for Petascale Data Instensive Computing", Elechtrotechnical Laboratories, Technical Report, TR-2001-4, http://datafarm.apgrid.org.
11. B. Tierny, W. Johnston, J. Lee, M. Thompson, "A Data Intensive Distributed Computing Architecture for Grid Applications", Future Generation Computer Systems, volume 16 no 5, pages 473-481, Elsevier Science, 2000.
12. J. Morris, M. Satyanarayanan, M. Conner, J. Howard, D. Rosenthal and F. Smith. "Andrew: A distributed personal computing environment", *Communications of the ACM*, 29(3):184-201, 1986.
13. William Gropp, et. Al., MPI – The Complete Reference, Volume 2, The MPI Extensions, The MIT Press, Cambridge, MA 1999
14. William Gropp, Ewing Lusk, and Rajeev Thakur, Using MPI-2, Advanced Features of the Message-Passing Interface, The MIT Press, Cambridge, MA 1999
15. James S. Plank, Micah Beck, Wael R. Elwasif, Terry Moore, Martin Swany, Rich Wolski "The Internet Backplane Protocol: Storage in the Network", NetStore99: The Network Storage Symposium, (Seattle, WA, 1999)

[Notes to reviewers
Even though the paper is long we need to add results on multi-file check-pointing of user data as this is the most common bottleneck we have found in real applications. I.e where the applications every Nth iteration dumps its entire vector fields for either restart or visualization use.]