An Approach to Parallel MxN Communication

Felipe Bertrand   Yongquan Yuan   Kenneth Chiu
Randall Bramley*

May 4, 2004

Abstract

High performance scientific applications are frequently multiphysics codes composed from single physics programs, or have a functional decomposition based on physics as well as a domain decomposition for parallelism. In either case dividing the application into independent components that can be developed and tested separately is useful. This requires a fast and efficient mechanism to share the large parallel data structures that are used in scientific applications. The "MxN problem" is the transfer of data between two scientific parallel programs with different numbers of processes on each side. We present a solution to the MxN problem based on an MPI-I/O interface. This approach builds on existing technology, emphasizes the easy migration of current applications, and simplifies the unit testing of the components while maintaining parallel high performance throughout the application. It also does not require a component to know how many processes another communicating component has.

1 Introduction

High-performance scientific computing now faces problems that span multiple scales, domains, and disciplines. The resulting multiphysics simulations are composites of highly-specialized codes developed by large, diverse, and sometimes geographically distributed research teams. For example, a global weather simulation engine could require expertise in meteorology, hydrology, geology, and oceanography. Current computational simulation projects that integrate different disciplines and codes include the Community Climate System Model (CCSM) [1] for global climate models, the Earth System Modeling Framework [2] for weather and climate simulations, the Caltech virtual shock physics facility [3] for simulating the dynamic response of materials, and the Center for Simulation of Advanced Rockets (CSAR) [4]. An emerging field for multiphysics simulation now being planned is the integrated simulation of magnetically confined fusion energy, which will require models for turbulent transport, extended magnetohydrodynamics, radio frequency heating and drives, and nonlinear gyrokinetics [5]. Over 50 major design and analysis codes are being maintained

*Work supported by National Science Foundation Grants 0116050 and EIA-0202048, and Department of Energy's Office of Science SciDAC grants.

by the magnetic fusion community, using a range of numerical techniques (finite elements, particle-in-cell, finite volumes, adaptive mesh refinement). Several of those codes, together with high-performance data systems to conduct software verification and model validation, will need to operate together to create an integrated simulation capable of diagnostics or prediction of a complete magnetic confinement fusion test.

Modern software methodology has introduced *software components* [6], which emphasize the capability for independent reuse and deployment - precisely the capabilities required to form multiphysics simulations, either from existing single-physics programs or by applying a functional decomposition design to a new application. The Common Component Architecture (CCA) Forum is an effort by universities and U.S. national laboratories to define a standard architecture for high performance computing components [7]. CCA components are independent units of software with clearly defined interfaces which can be assembled into applications at run-time. The CCA Forum concentrates on issues particular to high-performance computing: the use of multiple languages including Fortran 95, the need for low overhead in component interfaces, performance engineering, and the use of components that consist of parallel processes. This paper addresses the last issue, but as a side benefit also provides the language independence implied by the first.

Even with the use of components, there are two fundamental approaches to creating HPC multiphysics applications. The first, exemplified by the CCSM and CSAR projects, is to create a single executable by integrating all of the necessary physics into a single mathematical and software framework. Here the term "single executable" may refer to a job consisting of multiple MPI [8] tasks, sometimes with a functional decomposition among the tasks, but initiated (in the case of MPI programs) with a single MPIRUN command. This has the advantage of using a familiar high-performance communications system (the MPI API) for connecting the different simulation systems, but the disadvantage of requiring extensive rewriting of existing codes to perform data conversion, mesh mediation, synchronization, and coordination. It often requires redistributing large data objects since each of the constituent modules typically has its own optimized data decomposition.

The second fundamental approach is to leave the pre-existing code modules as separate processes, and connect the separate executables by having them communicate through files I/O. This has several advantages:

- Only minimal changes to the existing codes are required
- Data conversion can be carried out by introducing an additional filter program between the two communicating applications, rather than adding that capability to one or the other of them
- The intermediate files provide valuable restart and debugging resources
- Individual modules can be tested and debugged in isolation
- Individual modules can use different languages and runtime systems.

The most significant disadvantage of this approach is the performance penalty. Even with parallel I/O and striping, hard drive access is typically thousands of times slower than processes communicating over a fast network.

This paper proposes a new intermediate approach. Drawing on ideas from UNIX, we abstract communication channels as files by exploiting the MPI-I/O facilities in the Message Passing Interface [8]. The application codes read and write files using the standard MPI I/O interface, but instead of going to hard drives the data is streamed in parallel between module programs working together on an integrated simulation. This capability is provided by adding a new abstract device interface [9] to the ROMIO library [10, 11] which provides I/O for most major MPI implementations. The resulting approach combines advantages from the two major approaches for composing existing codes. High-performance parallelism is maintained, little or no change is needed to the existing application codes, and to some extent the redistribution of data required by the MxN problem is automated. The implementation provides a stage on the migration path towards integrating separate codes into a single executable, and allows rapid prototyping of code integration without first porting everything to work under a distributed computing framework. The model is intuitive, since it is based on a familiar file I/O model that most application users have already mastered.

2 The MxN Communication Problem

The proposed approach also solves a problem unique to high performance component software. The "MxN problem" is the transfer of a distributed data object from a module running on M processes to another running on N processes. More generally, the semantics of one component invoking a method on another component are not well-defined or standardized yet, and this is a major topic of research within the CCA Forum. Earlier work in PAWS [12] delineated several interaction possibilities, dealing with how many processes on each side participate, whether or not each side participates asynchronously, etc.

Within the CCA, MxN semantics are currently limited to two cases: all the proceses in each component[1] participate in the call, or min{M,N} processes in each component participate. This paper deals with the first case, and concentrates on the transfer of a distributed object from one component to another. Current CCA implementations involve specialized MxN components that mediate the communication between standard components. These CCA MxN components are derived from older, successful systems such as CUMULVS [13] and PAWS [12]. Figure 1 shows the architecture of a CCA MxN component. The user components communicate with each other using the communication API that the MxN components offer.

3 Existing Approaches

Previous data parallel communication libraries include CUMULVS [13], PAWS [12], HPF [14], and MetaChaos [15]. These generally take the same approach, which is to define a custom communication API to allow the definition and sharing of complex parallel data structures.

[1]In this paper we use the CCA terminology: a single component may consist of multiple parallel processes, and the decomposition of an application into components is independent of the number of processes each component instance uses.

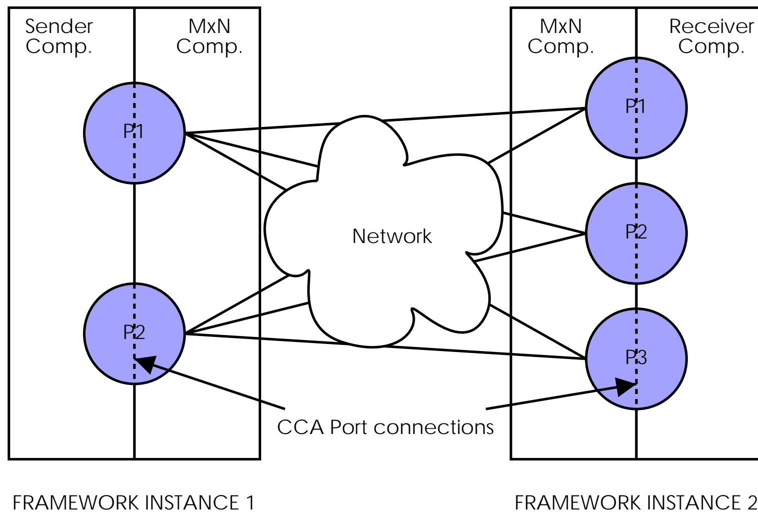

Figure 1: MxN component in CCA. Each component has its own framework instance or run-time system. Each component uses its local instance of the MxN component as intermediary. Processes are represented by round blue circles, components by the white boxes. Processes are shared by components living in the same framework instance.

Another related work is the Condor project [16], which uses transparent I/O redirection to execute processes on idle workstations. We also use transparent I/O redirection in our work. The extent of the Condor project is much broader, but is oriented towards single-process serial programs only. The Bypass project [17, 18] by the Condor team in particular allows split execution models; our work here with the MxN device extends that idea to arbitrary numbers of parallel processes on each side of the communication.

Although we focus mainly on the problem of data transfer, the MxN problem can be approached in a more general way, involving not only the mere transfer of parallel data, but also the invocation of services on remote parallel servers [19, 20, 21].

Remote I/O (RIO) is another ROMIO/ADIO back-end described in [22]. This work also leverages MPI-I/O by intercepting the calls to the I/O system, but its purpose is to allow MPI applications to access remote file systems, rather than to provide a way of communication between simultaneously executing, collaborating parallel components.

The Message Passing Interface (MPI) [8, 23] is a widely used standard for writing message-passing programs. The interface establishes a practical, portable, efficient, and flexible standard for message passing. MPI-2 allows separate parallel programs to communicate even if they were not started in the same MPI instance. MPI strongly couples the two programs, preventing fully decoupled development and testing. However, the MPI interface is familiar and widely-used by applications scientists, and one goal of our work is to allow them to work strictly within an MPI application interface without needing to explicitly use a new framework or run-time system.

4 MxN parallel data transfer

4.1 Goals

One goal for an MxN solution is the application-level "invisibility" that Bypass provides, where no changes are required in the individual component code. In particular, a component need not be aware of the number of processes in another component connected to it. This implies the language interoperability that CCA's SIDL enables, and maximizing the decoupling of the communicating applications. If the programs can be run independently, most of the testing effort can be done at the unit testing level, and the integration phase is simplified. A second goal is to allow a natural and quick migration from file-based coupling to real-time MxN communication. A common initial step in migrating towards an integrated simulation is to have the codes read and write files to transfer data, and we leverage this paradigm. The third goal is to develop a system for MxN communication that uses systems and API's familiar to most HPC scientists.

4.2 Implementation

The system we have developed for the transfer of data between parallel programs is based on defining a new device for the ROMIO implementation of MPI-I/O [24]. A key element of the design of ROMIO is an abstract-device layer (ADIO) [9]. This layer consists of a relatively small set of basic functions for parallel I/O. Adding a new back-end for ROMIO consists simply of implementing the ADIO interface for a new hardware device. Current devices for ROMIO include a default device for the UNIX file-system, a device for NFS file-systems, and a device for the parallel virtual file-system [25]. We created a new back-end, the *MxN* device. This device allows the application to transfer data using the regular MPI-I/O interface, as if writing to a file.

An MPI-I/O file operation is directed to another component or to a regular file (perhaps using the ROMIO PVFS device) depending on the file name. A file name starting with "mxn:" is connected to the MxN device, and thus be sent to a remote component. Files starting with "pvfs:", "nfs:", "ufs:" are sent to the PVFS, NFS and Unix file system modules. Since only the file name changes, development is easy: components are tested stand-alone, using files for input and output. Once they are running correctly, a change in the file name directly connects the components. If the file name is read in from a control parameter file, the application executable need not be changed or recompiled and coverage tests done during the unit-testing phase will remain valid during the integration phase. In summary, the strategy that we have devised brings a high level of transparency because the application is not aware of the MxN communication; it reads and writes data through a regular MPI-I/O interface.

Another advantage of this approach is the ability to transfer data between programs in different MPI instances, as long as ROMIO is the MPI-I/O implementation (MPICH, LAM-MPI, HP MPI, SGI MPI, and NEC MPI are examples of this). Communication across different architectures is also possible if the file is opened using MPI's "external32" data mode. In general the communication will always work if the target component can read a file generated by the source component.

Data is transfered using a stream paradigm (Figure 2). This is, however, a *logical* view of

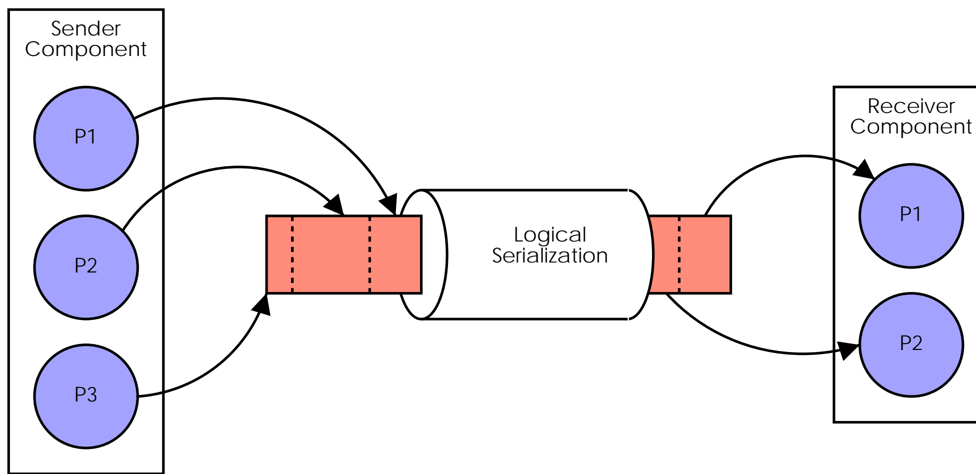

Figure 2: Data is logically ordered, but transfered in parallel.

what is going on. In reality, data is transfered in parallel and is never explicitly serialized. At the lowest level, the MxN device connects the parallel components using raw Unix sockets.

In MPI-I/O, data access can be done in a number of modes. Each mode is defined by a combination of three orthogonal aspects [23]: *positioning*, *synchronism* and *coordination*. Positioning defines the way the offset with respect to the the beginning of the file is calculated. It can be given explicitly as an argument to the read/write operation, or it can be implicit with the file pointer kept internally and each read/write operation continuing from where the last operation ended. This leads to two possible scenarios: either each process that shares an open file updates its own pointer independently, or all the processes update the same shared file pointer. Synchronism refers to the possibility of the operations being blocking or non-blocking. Coordination refers to the operations being collective or non-collective. In collective operations all the processes of the parallel program are involved in the I/O operation, whereas in the non-collective case, each process performs its own I/O operation independently. The MPI-I/O standard defines specialized functions for each possible combination of the above parameters.

Not all ADIO devices support every possible I/O mode. For example, a tape-based device will likely only support shared file pointers. The MxN abstract device supports only blocking, implicit shared file pointer access. This means that for inter-component communications the application only uses MPI-I/O operations with suffixes _SHARED and _ORDERED. Also, to enforce a one-way flow of data, files must be open in read-only or write-only modes only.

MPI-IO calls ending in _SHARED are non-collective. For files, the data is read or written at the current position of the shared file pointer. In the MxN device, each read operation simply returns whatever available data is next in the logical stream. These shared read and write operations are atomic in the sense that two concurrent reads or writes can not interleave, but, as allowed by the MPI-2 standard, the ordering is not deterministic.

On the other hand, calls ending in _ORDERED are collective. In this mode, each read or write is assigned a section of the file or logical stream. The order in which the sections are assigned is determined by the rank of the invoking process. Once each process's section of

the stream is computed, the MxN device transfers the data in parallel, hardware permitting.

The sending and receiving modes are independent. For example, the sender can be using the _SHARED mode and the receiver the _ORDERED mode. The general idea of the MxN device is that the semantics of the read and write operations in the MxN device are exactly the same as if using an intermediate file.

On the writer side, the MxN abstract device gets a reference to the data written by the client application. Write operations block until the requests coming from the reading side have consumed all the data. Once that happens, the reference to the buffer is released and the call returns. During read operations, a request is sent to the writer side. Data is collected from the remote processes until the request is completed. This scheme works because we do not allow random access to the data: the only mode of operation supported uses implicit, shared, file pointers. Note that the readers can obtain the data they seek as soon as it has been made available in the buffers of the writers. This implies that if the writer application writes a large chunk of data in each write operation, it will be easier for the readers to access the data in parallel. On the other hand, if the writers only pass a few bytes in each write operation, the readings will be serialized because the data will not be readily available (the readers will have to wait for their data to be available in a subsequent write operation). In summary, writing data in large quantities enhances parallelism among the readers.

The MxN device does not manage the data buffers. It transfers the data from the buffers at the writers' side to the readers' buffers directly. These buffers are managed by the ROMIO layer above ADIO and might or might not correspond to the original application buffers, because ROMIO must collect the original data in contiguous buffers. The main reason for not supporting random access is because, in that case, the MxN device would have to keep a local copy of all the data being written. We decided not to do this for performance reasons.

If the original application accessed the files linearly, then minor modifications (if any at all) will allow both file and MxN I/O. On the other hand, if the application uses intensive random access to files, then it will be more challenging to transform it to a linear file access scheme. Because the goal is to provide a staged approach to applications working as components with MxN communications, this restriction is generally not onerous. The files to which it applies are just the ones read or written by other programs, and most such files are accessed in a logically linear fashion. Other I/O the application performs (intermediate results, check-pointing, postmortem visualization and analysis) can remain unchanged.

This approach fulfills the goals outlined in the beginning of this section. The decoupling of applications during the software development cycle is achieved by directing the I/O to regular files. In this way the components can be tested separately, and unit tests can be prepared using custom-generated files. Migration is almost automatic. If the application already uses the I/O modes supported by the MxN device, a simple change in the file name is the only modification required. Also, the stream-based paradigm on which this mode of communication is based is easy to understand because of the similarity with I/O to a regular file.

A final advantage of this approach as compared to other current solutions [13, 12, 21] is that it does not require the user to define data structures in terms of custom data types provided by the library. Instead, it accepts MPI data types, which are routinely used in most parallel application programs.

5 Performance Measures

To measure the performance of this MPI MxN device we used a parallel solver system for a differential equation. The first component is a discretizer, which in parallel discretizes a given time-dependent partial differential equation using a simple form of domain decomposition, creating in parallel a large sparse linear system of equations. The resulting coefficient matrix and right hand side vector is forwarded to a solver component, which uses the Aztec library [26] of parallel iterative solvers. For an $n \times n$ mesh the amount of data transfered is about $96n^2$ bytes. The solver has a significantly greater workload so it is assigned sixteen processors while the discretizer has four. Tests were run on a dedicated cluster, each node consisting of dual 2.80GHz Xeon processors with 2GB of memory per node. The cluster has nine nodes, connected by 100Mb Ethernet. Each node is connected to a SAN system by Fibre Channel, and PVFS [25] was built to use four I/O nodes. MPICH version 1.2.5 was built with the additional MxN device described earlier.

Three versions were implemented: using files read and written with standard MPI I/O on top of PVFS, using a socket opened between MPI processes of rank 0 on each side, and using the MxN device. The file I/O version obtains parallelism through PFVS using direct file offsets to access files. Sequential access and shared file pointers are not supported by PVFS. The PVFS default 64 KB stripe size is used. The socket version serializes the transfer, by having the process with rank 0 in one component receive the data from its cohort and send the data over a socket to the process with rank 0 in the other cohort, which then distributes the data among its own cohort. This is the standard approach that would be used for communicating components when neither is aware of how many processes are participating in the other component. The MxN device solution is as described above, and allows arbitrary parallel transfers.

Figure 3 shows the elapsed wall clock time for the required data communication of each version; for each version we timed the complete transfer from the start of writing by the discretizer to the end of reading by the solver component. In the file I/O version the writing and reading are completely sequential, since without other runtime support, a component must wait for the write to complete before it can start reading a file. So the time shown is the sum of the discretizer writing time and the solver reading time.

All of the results used four processors for the discretizer and sixteen for the solver. So the amount of parallelism available to both the file I/O and MxN versions is four - even though the solver uses sixteen processors, the MxN device requires running the discretizer at the same time which limits the data transfer processing to its four processors. Although the number of readers is a multiple of the number of writers in this case, this is not a requirement for the MxN device and we have also tested its correctness for relatively prime numbers of processors. The times shown are averaged over five runs for each datum, with error bars corresponding to one standard deviation. The times are roughly linear in the datasize being sent, with the slope of the socket version 6.7 times larger than that of the MxN device. A factor of four comes from MxN using four connections simultaneously while the socket code only uses one; the additional performance improvement can be attributed to most of the socket version's data needing to make three hops instead of just one. The socket version also has higher variance than the other two methods, which we have observed on independent timings of using sockets on the network. The PVFS file I/O version is slower even though

it uses the faster Fibre Channel connections; this is in part because some of the data makes an extra hop going from a compute node to an I/O node, where it is then written out to the SAN device. That process is then reversed on the read phase. The times are shown on a log scale in Figure 3,

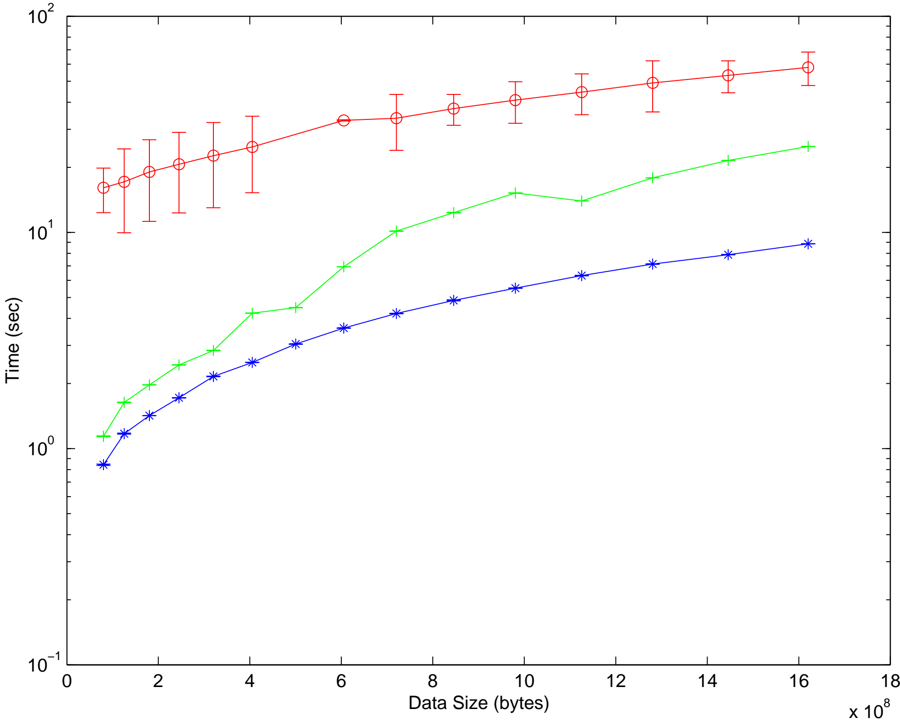

Figure 3: Discretizer-Solver Communication Times: MxN(*), PVFS(+), Socket(o)

6 Conclusion and Future Work

The MxN problem involves transfering scientific data structures between parallel components, an increasingly important task as multiphysics applications are developed by combining existing simulation codes. In this paper we created a system for parallel data transmission between parallel components which can be in different languages and even using different MPI implementations. This system is based on the logical serialization (or streaming) of data, while still allowing fully parallel transmission of the data. This communication using a new MPI-I/O abstract device allows users to write and run their application codes without having to be aware of how many processors a communicating component has, or even if the data is coming from an MPI file or a live parallel process. This provides an upgrade path for new efforts at developing composed simulations from existing stand-alone programs, where the eventual goal is either a single integrated code or the programs turned into CCA high-performance components.

A shortcoming with the MxN device is that it is not fully scalable. Since a socket is opened from each participating process to every process in the other component, this can require

M*N sockets for an MxN connection. To address this we are experimenting with moving the MxN device connection to the PFVS layer, so that the number of sockets required will be limited to the number of I/O nodes. From the PVFS layer this eliminates the time for actual transfers to physical storage units, and can also allow using block sizes better suited for the network than for a hard drive.

Currently, we are incorporating the MxN communication system into a CCA component. The strategy we are using is the same as that of existing MxN components based on CUMULVS [13] and PAWS [12], as depicted in Figure 1. In this scheme, two MxN components must be instantiated, each one connected with a CCA port to one of the communicating components. The application programming interface (API) exposed to the communicating processes is just a wrapper to the relevant MPI I/O services. The benefits of making a component out of this service include independent development and upgrade of the service without involving any recompiling or relinking of the applications. If a standard API for MxN components is created in the CCA Forum, then there would be an increased added value. In that case, MxN components could be exchanged at run-time, allowing the user to switch to the most efficient for the particular task at hand.

References

[1] Climate and UCAR Global Dynamic Division. Community Climate System Model. http://www.cgd.ucar.edu/csm/.

[2] University Corporation for Atmospheric Research. Earth System Model Framework. http://www.esmf.ucar.edu/.

[3] Caltech Center for Simulation of Dynamic Response of Materials. http://www.cacr.caltech.edu/ASAP/index.html.

[4] University of Illinois at Urbana-Champaign. Center for Simulation of Advanced Rockets. http://www.csar.uiuc.edu/.

[5] Jill Dahlburg et al. Fusion simulation project: Integrated simulation and optimization of magnetic fusion systems, 2002. Final report of the FESAC ISOFS Subcommittee, submitted to J. Fusion Energy.

[6] Clemens Szyperski. *Component Software*. Addison-Wesley, 1998.

[7] Rob Armstrong, Dennis Gannon, Al Geist, Katarzyna Keahey, Scott R. Kohn, Lois McInnes, Steve R. Parker, and Brent A. Smolinski. Toward a common component architecture for high-performance scientific computing. In *HPDC*, 1999.

[8] Message Passing Interface Forum. MPI: a message-passing interface standard. *International Journal of Supercomputer Applications and High Performance Computing*, 8(3/4):159–416, Fall-Winter 1994.

[9] Rajeev Thakur, William Gropp, and Ewing Lusk. An abstract-device interface for implementing portable parallel-I/O interfaces. In *Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computation*, pages 180–187, October 1996.

[10] Rajeev Thakur, Rob Ross, Rob Latham, Rusty Lusk, and Bill Gropp. ROMIO: A High-Performance, Portable MPI-IO Implementation. http://www-unix.mcs.anl.gov/romio/.

[11] Rajeev Thakur, Ewing Lusk, and William Gropp. Users guide for romio: A high-performance, portable mpi-io implementation. In *Technical Memorandum ANL/MCS-TM-234*. Argonne National Laboratory, July 1998.

[12] Peter Beckman, Pat Fasel, William Humphrey, and Sue Mniszewski. Efficient coupling of parallel applications using PAWS. In *Proceedings of the High Performance Distributed Computing Conference*, Chicago, IL, July 1998.

[13] G. A. Geist, II, James Arthur Kohl, and Philip M. Papadopoulos. CUMULVS: Providing fault tolerance, visualization, and steering of parallel applications. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(3):224–235, Fall 1997.

[14] Rice University. High performance fortran.

[15] M. Ranganathan, A. Acharya, G. Edjlali, A. Sussman, and J. Saltz. A runtime coupling of data-parallel programs. In *Proceedings of the 1996 International Conference on Supercomputing*, Philadelphia, PA, May 1996.

[16] Jim Basney and Miron Livny. Deploying a high throughput computing cluster. In Rajkumar Buyya, editor, *High Performance Cluster Computing*, volume 1, Architectures and Systems, pages 116–134. Prentice Hall PTR, Upper Saddle River, NJ, 1999. Chap. 5.

[17] Douglas Thain and Miron Livny. Bypass: A tool for building split execution systems. *Proceedings of the Ninth IEEE Symposium on High Performance Distributed Computing*, pages 79–85, August 1-4 2000.

[18] Douglas Thain and Miron Livny. Multiple bypass: Interposition agents for distributed computing. *Journal of Cluster Computing*, 4:39–47, 2001.

[19] Jason Maassen, Thilo Kielmann, and Henri E. Bal. GMI: Flexible and efficient group method invocation for parallel programming. In *Sixth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, Washington DC, March 2002.

[20] Kate Keahey, Pat Fasel, and Sue Mniszewski. Paws: Collective interactions and data transfers. In *Proceedings of the High Performance Distributed Computing Conference*, San Francisco, CA, August 2001.

[21] Kostadin Damevski. Parallel RMI and M-by-N data redistribution using an IDL compiler. Master's thesis, The University of Utah, May 2003.

[22] Ian Foster, David Kohr, Jr., Rakesh Krishnaiyer, and Jace Mogill. Remote I/O: Fast access to distant storage. In *Proceedings of the Fifth Workshop on Input/Output in Parallel and Distributed Systems*, pages 14–25, San Jose, CA, 1997. ACM Press.

[23] Message Passing Interface Forum MPIF. MPI-2: Extensions to the Message-Passing Interface. Technical report, University of Tennessee, Knoxville, Tennessee, US, 1996.

[24] Rajeev Thakur, William Gropp, and Ewing Lusk. Data sieving and collective I/O in ROMIO. In *Proceedings of the Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189. IEEE Computer Society Press, feb 1999.

[25] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. PVFS: A parallel file system for linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, 2000. USENIX Association.

[26] S. A. Hutchinson, J. N. Shadid, and R. S. Tuminaro. *Aztec Users' Guide: Version 2.0.* Sandia National Laboratories, http://www.cs.sandia.gov/CRF/aztec1.html, 1998.