Flexible I/O Support for Reconfigurable Grid Environments

Marc-André Hermanns[1], Rudolf Berrendorf[2], Marcel Birkner[2], and Jan Seidel[2]

[1] Central Institute for Applied Mathematics
Research Centre Jülich, 52425 Jülich, Germany
[2] Department of Computer Science
University of Applied Sciences Bonn-Rhein-Sieg, 53754 St. Augustin, Germany

Abstract. With growing computational power of current supercomputers, scientific computing applications can work on larger problems. The corresponding increase in dataset size is often correlated to an increase in needed storage for the results. Current storage area networks (SANs) balance I/O load on multiple disks using high speed networks, but are integrated on the operating system level, demanding administrative intervention if the usage topology changes. While this is practical for single sites or fairly static grid environments, it is hard to extend to a user defined per-job basis. Reconfigurable grid environments, where computing and storage resources are coupled on a per-job basis, need a more flexible approach for parallel I/O on remote locations.
This paper gives a detailed overview of the abilities of the transparent remote access provided by TUNNELFS, a part of the VIOLA parallel I/O project. We show how TUNNELFS manages flexible and transparent access to remote I/O resources in a reconfigurable grid environment, supporting the definition of the amount and location of persistent storage services on a per-job basis.

1 Introduction

With the enormous increase in computational power of today's high performance computers, scientific applications can exceed old boundaries of problem size and complexity. With the increasing rate at which data can be produced, the question arises as to where to store the data in an efficient way. Result data will normally be too large to be kept in memory, while the application continues processing different data sets. Also, memory-intensive applications often need to swap data to disk, and reread it at a later point in time. With increasing size of the data sets to be loaded into memory and stored on persistent storage devices, I/O can easily become the bottleneck of modern scientific applications. To develop efficient and portable applications, it is therefore essential for the middleware to provide efficient I/O mechanisms for scientific application developers.

Almost 10 years after the specification of MPI-2 [1] in 1996, most currently available MPI implementations support the API for MPI-IO. ROMIO [2] is a publicly available MPI-IO implementation that easily integrates into the MPICH MPI

implementation [3]. ROMIO and MPICH are being developed at the Mathematics and Computer Science Department of Argonne National Laboratory. The use of the MPI API for I/O introduces a new layer of abstraction to a process's I/O accesses, increasing portability of the application. ROMIO uses the device abstraction ADIO [4] (Abstract Device Interface for I/O) to define multiple special purpose filesystem devices. In this way the MPI layer can call a generic ADIO function to handle a specific kind of I/O operation, with the ADIO layer choosing the correct device to use.

When using the MPI API for parallel I/O, it is assumed that all processes participating in each I/O call see the same underlying filesystem. In grid environments, where processes are interacting across inter-cluster boundaries with other processes, this is rarely the case and a common filesystem can introduce a high administrative overhead, like common user bases and common security policies. In static environments, where the status of available nodes in the grid is constant over a longer period of time, this is still feasible, as shown by the DEISA project [5], where a distributed GPFS filesystem is deployed over several supercomputing sites. In reconfigurable grids, where computing resources are added to and revoked from the resource pool much more frequently, this approach is a major administrative challenge.

In the VIOLA project [6] several sites with local clusters are connected by a 10 GBit/s dedicated network, while internally using either Gigabit Ethernet or Myrinet networks [6]. The MPI middleware for connecting the cluster sites is MP-MPICH [7]. MP-MPICH is an enhanced version of MPICH, providing a process environment for meta computing. The major advantage of MP-MPICH is the transparent use for different communication devices for inter- and intra-cluster communication. Thus clusters can still use the usually faster special purpose interconnect for intra cluster communication, instead of having to choose the least common denominator of all of the available communication devices. While earlier versions of MP-MPICH realize the inter-cluster connection with transparent router processes, providing a store-and-forward routing for messages between processes of different clusters, recent prototypes provide a secondary device to each process to handle intra- and inter-cluster communication by separate devices [8].

Within the parallel I/O subproject of VIOLA, we develop two ADIO devices for MP-MPICH to support the special needs for efficient parallel I/O in grid environments [9]. In Sect. 2 we introduce the overall design of the TUNNELFS client/server architecture. Section 3 presents possible I/O distribution strategies, used with TUNNELFS. In Sect. 4 we then state some of our results with the early version of the prototype using the VIOLA network. Sect. 5 places our current efforts into the context with other work, involving remote I/O for MPI. Section 6 concludes with a brief summary and future prospects in our work.

2 Design

To support special aspects of I/O access with different filesystem, ROMIO uses the ADIO layer [4] to decouple the implementation of MPI function calls and

filesystem specifics. The MPI-IO functions work on driver functions of the ADIO layer that select the correct ADIO device to be used for a specific filesystem (e.g., UFS, NFS, PVFS2). To integrate transparent I/O for MPI applications, we defined two new ADIO devices, TUNNELFS and MEMFS [9]. TUNNELFS provides the service for transparent remote access (see Fig. 1, and MEMFS creates a multi-node shared virtual filesystem accessible with MPI-IO calls. Together they provide easy and efficient access to I/O on distributed memory resources.

All client/server communication regarding remote I/O is handled transparently by the ADIO device for TUNNELFS. The user therefore does not need to deal with explicit communication calls to the server. In MPI, file namespaces are implementation depended, as it may be required to provide additional information about the file, such as the type of the underlying filesystem. ROMIO uses prefixes separated by a colon, to explicitly identify a filesystem type (e.g., `nfs:file.dat`). TUNNELFS and MEMFS use the same prefix scheme for selection of the corresponding ADIO device (e.g., `tunnelfs:file.dat` or `memfs:file.dat`). For TUNNELFS the location of a file is not pinpointed to a specific server but rather to a class of servers belonging to the same *filesystem domain*, additionally coded in the filename. Filesystem domains and their usage are discussed in detail in Sect. 2.4.

2.1 Client/Server Architecture

TUNNELFS uses a parallel client/server architecture. As shown in Fig. 1, the user process, being the I/O client, issues an I/O call, descending through the MPI layers until the ADIO device layer is reached. Instead of calling system functions for disk I/O on the client's node, the TUNNELFS ADIO device transmits the request parameters and possible I/O data to the server, using MPI point-to-point communication. The I/O server receives the request of the client and acts upon it, using local MPI-IO calls. If the request is a read request, the server will issue the call locally and transfer the data to the client. If the request is a write request it will wait for the data to be sent by the client in a second message and then issue a local write call. Communication between clients and servers as well as between multiple servers is handled with point-to-point communication to avoid dead-

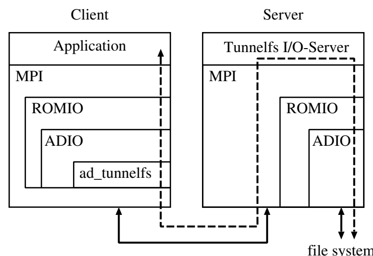


Fig. 1. Layers of the TUNNELFS device and I/O server.

locks and excessive blocking of server processes waiting for messages of other servers.

To maintain portability, flexibility and efficiency, the TUNNELFS I/O server uses only MPI function calls for communication and file I/O. Request header and data buffers are transferred in separate messages, to avoid repacking of buffers before and after transfer. The client/server protocol defines a single master I/O server that is derived from the middleware configuration files, common to all processes started. All client/server and server/server communication is issued upon an implicitly defined communicator, containing all user processes as well as all I/O server processes. As the communication involved in I/O requests is hidden from the user program, the user has no knowledge which server is appropriate for a specific file. Thus, initial requests are always sent to the global master I/O server, which either processes the request locally or delegates it to a different server. In case of request delegation the client is informed of the I/O server now responsible for this file.

2.2 I/O-Transfer-Staging

Though I/O calls usually handle large amounts of data, allowing arbitrary sized blocks for TUNNELFS is not feasible. The TUNNELFS protocol defines a two step data transfer for I/O calls. In a write call the buffer is transferred to the server process first, and then written to disk. Keeping large buffer sizes will result in a serialization of those two operations, maximizing the time needed for the complete operation. As MPI also defines that buffers involved in ongoing MPI operations must not be accessed until the operation involving the buffer has completed, transfers have to be broken down into smaller parts. Therefore the TUNNELFS servers stage data transfers to and from the client with its local I/O calls, to maximize overlapping of those operations.

For write requests, the client sends a request header, followed by a number of messages containing the I/O data. The number of I/O data messages is coded in the request header. After receiving the first I/O data package, the server can start writing the data to storage, while the rest of the I/O buffer is still in transfer. The write call is acknowledged by the server through a reply message. For read requests, the client sends the request header and the server replies directly with the parameters of the issued call, in particular the number of following I/O data packages. After the client processes the server reply, it will start receiving the messages containing the requested I/O data.

2.3 Multiple Distributed Server Support

The first prototype of the TUNNELFS and MEMFS devices supported any number of clients but only a single I/O server to be used with the application. This was partly due to the router concept of earlier versions of MP-MPICH, where avoidance of bottlenecks would have meant introducing a router for every I/O server defined. This would have implied an enormous overhead of additional processes. With the modification of MP-MPICH to support direct process to process inter-cluster

communication without interference from use of router processes [8], multiple I/O server support becomes feasible for the MP-MPICH environment. With multiple I/O servers present in the application's MPI environment, I/O load sharing and I/O request delegation, as well as transparent file access becomes possible. The TUNNELFS client/server protocol was extended to support the desired flexibility and transparency with the use of multiple I/O servers.

2.4 Filesystem Domains

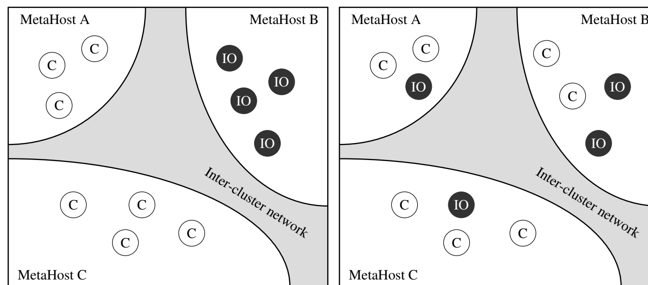
As TUNNELFS is not a filesystem itself, but an interface to different remote filesystems, it has to provide a means for distinguishing those filesystems at runtime. The TUNNELFS I/O servers need information in addition to the filename, what the location of that file in the grid is. As nodes are assigned to the user application by the scheduling system, it is possible but not always practical, to request specific resources of a cluster for execution. With a uniform node architecture, i.e., no special purpose nodes available for reservation, it is not necessary to run the I/O server on one specific node, but rather on any node which has access to the desired filesystem. Within the grid, processes can then be assigned to specific classes, where each class has access to the same filesystem (e.g., the local filesystem on one cluster of the grid). These classes are called *filesystem domains*.

MP-MPICH defines so-called *metahosts* to classify all processes of a single cluster sharing a common internal interconnect. These metahosts comprise the nodes of the global meta computer. As nodes within one cluster usually share a global filesystem, filesystem domains contain all processes of one or more metahosts. The identifiers of a filesystem domain are implicitly defined by the user, as they are derived from the grid configuration defined by the user for this specific job. If the user defines a metahost with name "`metahostA`", clients have remote access to that filesystem via the prefix "`tunnelfs:metahostA:`".

To prevent the clients from governing too much information on data location, the logic for deciding which server is used for opening a file resides on the global master I/O server. Thus the client itself does not process the filename any further than the first separating colon. The TUNNELFS prefix is truncated from the filename and the rest of the filename is sent to the master I/O server for further processing. The master I/O server then decides whether the file is to be opened locally or the open request has to be delegated to a server of a different filesystem domain.

2.5 Distribution Schemes

During job execution user processes and I/O server processes are distributed in the grid. The configuration and placement of user processes can have direct influence on efficient placement of I/O server processes. Additionally I/O server processes have to be placed according to the required filesystem access of the application. Each filesystem domain that needs to be accessed during execution has to define at least one I/O server. During runtime of the application, the server placement should then be taken into account when clients are assigned to



(a) All I/O servers on a single metahost

(b) I/O servers distributed over metahosts

Fig. 2. Different placement scenarios for TUNNELFS

a specific server. As a part of the transparent access scheme, the distribution is completely handled by the TUNNELFS servers, with only minimal intervention by the user. The user can define hints in an `MPI_Info` object, referenced on special I/O calls like opening or setting a file view. The server will then use the given information for optimal server assignment.

The servers are assigned to the clients by several different distribution schemes. The most simple distribution scheme is *single server*, where all clients use one server for a specific file. This is very close to the behavior of our first I/O server prototype, where only a single server was allowed in a job specification. One difference to having only one server in the system is that even though a single file is handled by one server exclusively, other files opened can be handled by other servers, providing a fairly balanced distribution of files over all available I/O servers. Another simple distribution scheme is *balanced global distribution*, where all defined servers are equally involved in sharing the I/O load. This scheme is not available for all types of files yet, as it is relying on a global filesystem that can handle multiple file handles on a single file without interfering with each other. This distribution scheme is used mostly for the MEMFS virtual filesystem, which can handle this efficiently. Figure 2(a) depicts a possible distribution of I/O servers, where all special I/O processes are started on a single metahost which is the only one providing I/O services to the compute grid.

A third distribution that might not result in a balanced distribution is *filesystem domain distribution*, as shown in Fig. 2(b). This is a specialization of the *balanced global distribution*, where clients are only assigned to servers of their filesystem domain. The goal of this distribution is to have an I/O server available via the faster local interconnect of the local cluster. Especially write operations can then be handled more efficiently, as the data can be transferred via local interconnect to a server, where it can be processed without further waiting time for the client. The user can influence the default mapping of clients to servers by

providing additional information in an MPI_Info object, restricting the mapping to servers of a specific filesystem domain.

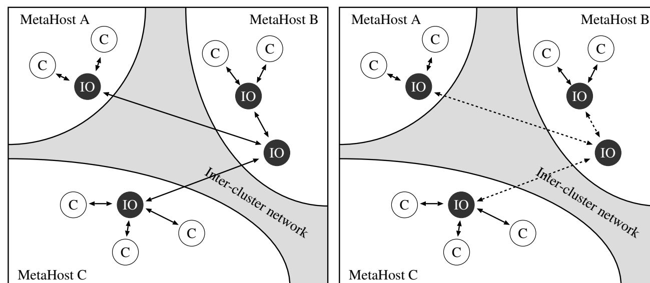
3 Distributed I/O Strategies

The main objective in development of the TUNNELFS ADIO device and server is efficient support for remote filesystems. The supported filesystems are primarily dependent on the other ADIO devices present in the ROMIO library. The MEMFS virtual filesystem is designed to be global over all TUNNELFS servers. Additionally, it supports multiple independent file handles on the same file. Colliding accesses are handled within MEMFS itself, so TUNNELFS can have multiple distributed handles open at the same time, without the need of collective operations on the file among all servers. Filesystems on the operating system level usually do not provide this service to an application, therefore the TUNNELFS servers have to use different strategies to support I/O load sharing among the servers when working on persistent filesystems.

As stated in Sect. 2.4, the TUNNELFS servers assume that every process within a filesystem domain has equal access to the filesystems of that metahost. Clients as well as servers are classified into these filesystem domains. Upon first registration during the initialization of MPI, the clients provide information about their filesystem domain to the main server. The main server can then use this information to compute an optimal distribution of clients to servers. For example, clients can be assigned to the nearest server. Having a server in reach of intra-cluster communication can speed up data transfer, as it is used in routed and cached I/O with intermediary I/O servers. With this filesystem domain distribution, the TUNNELFS servers support two sorts of I/O behavior, in regard to the configured views on the file.

3.1 Routed I/O

Routed I/O is a transfer of the router concept of MP-MPICH [7] from routing processes to the I/O server processes. Clients are assigned to I/O servers in a distributed fashion in a way that they are local to their metahost and therefore reachable via the local, typically faster internal network. After completion of the data transfer, the intermediary servers reply to the clients and then send the data to the responsible I/O server for persistent storing of the data. The clients can then continue with computation much earlier than having to wait for explicit storage of the data on the remote I/O server. The intermediary server uses the same file view the client would have used to write to the file, therefore no additional offset and file view calculation has to be done, and the server simply reuses the information. While write access is being accelerated by this strategy, read access is degraded, because of the store-and-forward data transfer and the additional hop of the intermediary server. Therefore this I/O strategy is not the best for all kinds of file accesses and is currently only used if MPI_MODE_WRONLY is specified upon opening the file. The clear benefit is the usability for arbitrary file views



(a) Routed I/O (b) Cached I/O

Fig. 3. Different TUNNELFS I/O strategies.

in comparison to cached I/O discussed in the next section. Figure 3(a) shows I/O message interchange between clients, intermediary servers and the responsible file server. This I/O strategy is still reasonable for scientific applications, as often the major I/O load is created by writing intermediary and result data.

3.2 Cached I/O

Cached I/O goes further and speeds up the read access, by keeping a local cache file of the clients view. This will imply coherence problems, if several clients have access to the same region in a file. To avoid complicated handling of replicated data, cached I/O is restricted to file views with disjoint access patterns.

When a message passing library like MPI is used, the application problem domain is often already restricted to algorithms that do not need random access on global data, as this can be a big challenge to provide efficiently without hardware support of some kind. Thus disjoint access to global data is an access pattern very common in scientific applications using MPI. In Fig. 3(b) the message exchange for cached I/O is shown. It is very similar to the routed I/O scenario, but now the intermediary servers only synchronize with the responsible file server on special I/O requests, such as `MPI_File_close`, `MPI_File_set_view` and `MPI_File_sync`. This postponed synchronization, in regard to routed I/O, is illustrated by the dotted lines between the server processes.

4 Status

As the implementation of distributed TUNNELFS I/O servers has been completed recently, we cannot state solid performance data for it yet. With multiple servers balancing I/O load we expect to significantly exceed the already promising performance data of the single server TUNNELFS environment [9] presented in Fig 4. In this single server test, we are able to deliver I/O rates that are restricted

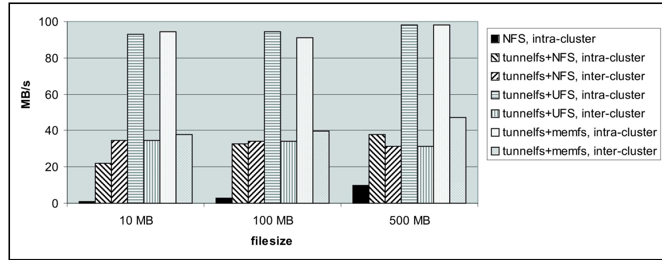


Fig. 4. Bandwidth with 12 I/O client processes and 1 I/O server on a shared file.

by the network card of the server (1 GigE) rather than the processing on the server. The data was obtained using an MPI benchmark program that measured 8 different MPI I/O operations (individual/shared fp, explicit offset, collective/non-collective). Results are given as average bandwidth numbers over these operations. Two clusters were used, connected with a dedicated 100 km distance 10 GigE fibre network and 1 GigE network cards in each cluster node connected to a cluster switch. All cluster nodes running TUNNELFS I/O server processes are 4-way SMP nodes with 10.000 rpm SCSI disks accessed locally on a node by a UFS device, and an NFS-mounted RAID-5 based shared filesystem hosted by a cluster file server.

5 Related Work

Data access to remote locations from within MPI applications has been previously addressed with the RIO device, where the communication for I/O is directly bound to TCP/IP socket communication. The work was continued with RFS [10], focusing on client side caching to improve I/O performance. To the best of our knowledge, efficient remote data distribution using MPI datatypes for optimal placement on multiple servers has not been extensively addressed yet.

6 Conclusion and Future Work

We defined and implemented a client/server architecture for flexible, transparent and efficient I/O in grid environments. With our approach to distributed I/O server support, we have designed a very flexible infrastructure that can be adapted to applications' I/O needs on a per-job level. Users can define the number of I/O processes supporting their application as well as their placement in the grid. This enables substantial influence of the I/O infrastructure presented to applications by external configuration prior to runtime. The lifetime of TUNNELFS I/O server processes is limited to the scope of an applications execution time and are integrated into the application's MPI environment by the middleware. We introduce the term of filesystem domains to assist the user in a flexible specification for

data location. TUNNELFS uses a client/server architecture, using MPI communication and I/O calls to maintain a maximum of portability and flexibility, while allowing the use of special purpose MPI devices for inter process communication.

With completion of TUNNELFS and MEMFS prototypes, further efforts will be aimed at an improved performance and robustness of the implementation, while using it with simulation applications in the VIOLA testbed.

Acknowledgments

This work was supported within the VIOLA project by the German Ministry of Education and Research under contract number FKZ 01AK605L. Thanks to Martin Pöppe, Boris Bierbaum and Carsten Clauss of the University of Technology Aachen for their very close support on MP-MPICH.

References

1. Message Passing Interface Forum (MPIF): MPI-2: Extensions to the Message-Passing Interface. University of Tennessee, Knoxville (1996) `http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html`.
2. Thakur, R., Ross, R., Lusk, E., Gropp, W.: Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation. Technical Report ANL/MCS-TM-234, Mathematics and Computer Science Division, Argonne National Laboratory (2004) `http://www-unix.mcs.anl.gov/romio/`.
3. Gropp, W., Lusk, E., Doss, N., Skjellum, A.: A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. Technical report, Mathematics and Computer Science Division - Argonne National Laboratory (1996) `http://www-unix.mcs.anl.gov/mpi/mpich/`.
4. Thakur, R., Gropp, W., Lusk, E.: An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces. In: Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation. (1996) 180–187
5. The DEISA Project Group: Distributed European Infrastructure for Supercomputing Applications (2005) `http://www.deisa.org/`.
6. The VIOLA Project Group: Vertically Integrated Optical testbed for Large scale Applications (2005) `http://www.viola-testbed.de/`.
7. Pöppe, M., Schuch, S., Bemmerl, T.: A Message Passing Interface Library for Inhomogeneous Coupled Clusters. In: Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS 2003), Workshop on Communication Architecture for Clusters (CAC 2003), Nice, France (2003)
8. Bierbaum, B.: Implementation of a Multi Device Architecture for MetaMPICH. Master's thesis, Chair for Operating Systems, RWTH Aachen (2005) `http://www.lfbs.rwth-aachen.de/~boris/diplomarbeit.pdf` (In German).
9. Berrendorf, R., Hermanns, M.A., Seidel, J.: Remote Parallel I/O in Grid Environments. In: Proc. of the Sixth Conference on Parallel Processing and Applied Mathematics (PPAM). Lecture Notes in Computer Science, Poznan, Poland, Springer (2005) to appear.
10. Lee, J., Ma, X., Ross, R., Thakur, R., Winslett, M.: RFS: Efficient and Flexible Remote File Access for MPI-IO. In: Proceedings of the International Conference on Cluster Computing. (2004)