Research Trends in High Performance Parallel Input/Output for Cluster Environments

Thomas Ludwig

Ruprecht-Karls-Universität Heidelberg
Institute for Computer Science
Parallel and distributed Systems
69120 Heidelberg
Germany
Email: t.ludwig@computer.org
Web: pvs.informatik.uni-heidelberg.de

Abstract

Parallel input/output in high performance computing is a field of increasing importance. In particular with compute clusters we see the concept of replicated resources being transferred to I/O issues. Consequently, we find research questions like e.g. how to map data structures to files, which resources to actually use, and how to deal with failures in the environment. The paper will introduce the problem of massive I/O from the user´s point of view and illustrate available programming interfaces. After a short description of some available parallel file systems we will concentrate on the research directions in that field. Besides other questions, efficiency is the main issue. It depends on an appropriate mapping of data structures onto file segments which in turn are spread over physical disks. Our own work concentrates on measuring the performance of individual mappings and to change them dynamically to increase performance and control the sharing of resources.

Keywords: Parallel programming, parallel I/O, cluster computing.

1. Introduction

In recent years we see an explosion of the amount of data that is stored in all sort of devices all over the world. Prices for hard disks and media like CD and DVD drop constantly thus allowing us to handle ever increasing volumes of data. Looking at applications we can easily find prominent examples in the field of database systems: Google´s data repository stores more than 3 billion web pages together with their content and provides rapid access to the information therein [10]. Data stored in database systems is often so comprehensive that you need special concepts like data mining to extract meaningful information from it. Another field of importance is natural science. In physics we see research being conducted at the CERN where terabytes (10^{12}) and even petabytes (10^{15}) of data are produced [16]. Biology provides us with almost countless data from gen sequences and with proteomics data sets will increase by orders of magnitude [15]. The Berkely report “How Much Information? 2003” recorded an increase of information stored in 2002 by 5 ExaBytes ($5x10^{18}$ bytes) where 92% of it are stored on magnetic media, primarily hard disks [22].

So the question is of how to deal with all these data. Where and how is it stored? Various technical concepts have been developed: We find RAIDs (redundant arrays of inexpensive disks), SANs (storage area networks), and NAS (network attached storage), to mention only some of the more popular approaches. To access data efficiently we use higher level abstractions like e.g. distributed file systems. For programmers and programs the most popular interface are the POSIX-compliant read()- and write()-functions, which are portable over all these different architectures.

With high performance computing, however, the situation changes. First, we now execute parallel programs. Data structures distributed over a set of processes logically form one single unit and thus should be stored as such on disks. Second, high performance computers are typically parallel computers with replicated hardware. Recently we see more and more cluster computers where commodity-of-the-shelf components (COTS) are deployed. This includes also storage and many clusters exhibit a node architecture with locally attached disks. In this paper we will investigate the question how to use these disks efficiently.

A problem arises with the quickly increasing compute performance of processors and the disks not being able to keep pace with it. We now measure compute performance in teraflops per second but I/O performance is still only several hundreds of megabytes per second. Furthermore, the aggregate amount of main memory of larger clusters often exeeds one terabyte and obviously we need clusters of large disks in order to store input/output data. The share of costs we need to spend for the storage system is now a considerable percentage of the total costs of the whole system. Evidently we need more research to use these resources efficiently and to provide the users with powerful means to handle data outside the main memory.

One such concept is parallel input/output (I/O). This covers two aspects: first, the view of the programmer onto data in files and second the distribution of files over disks. Traditionally, we find non-parallel I/O in parallel programs (sometimes also called sequential I/O). Processes send their data to a master process which in turn writes the data to disk (see figure 1.a). Obviously, this slows down the parallel application, results in hot spots in the network neighboring the master´s node and heavily charges the path from this node to the disks. With parallel I/O processes view their data as being part of a shared file and reading and writing is handled by each node. The resulting file is striped over a set of disks attached to a set of nodes of our cluster (Figure 1.b). There are many facets of parallel I/O and we will go into details later on.

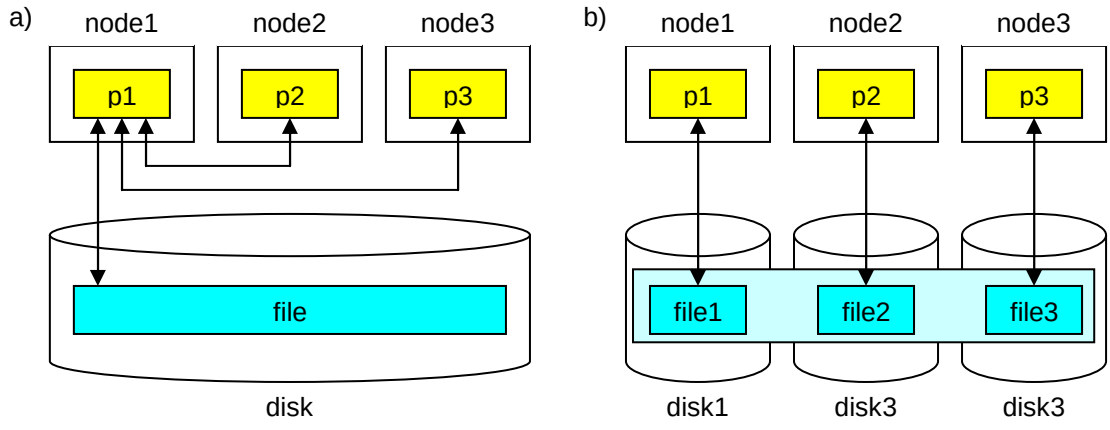

Figure 1: a) Sequential I/O via a dedicated master b) parallel I/O from parallel processes to a set of disks.

The low-level mechanisms that we deploy here are provided by a so called parallel file system. In what is it different from the above mentioned RAID systems and from distributed file systems? Most importantly a parallel file system is independent of storage hardware components. It spans over a set of I/O devices which can be hard disks, RAID systems or any other component that provides random read/write access to files. A parallel file system stores (logical) files in a distributed manner in (physical) files on a set of devices. It usually uses concepts like striping to determine the distribution and by that employs RAID-level 0. Other distributions are possible and we will discuss the question of their influence onto performance. Distributed file systems like e.g. NFS, AFS, and DFS also use a set of devices to store data, however, they do not subdivide a single file into individually stored segments. Instead, their main purpose is to support location transparency. For details on the latter see [7]. Parallel file systems always provide us with a parallel file access semantics that is defined by the application programming interface (API) of that system. In most cases however, we do not use these native APIs directly, instead map portable APIs like that of MPI onto them.

Research on parallel I/O started in the mid-90s but 5 years later there were still only few groups being active in that field. Only recently the field attains more attention. In 2000 the Euro-Par conference introduced this topic into the list of their workshops [5], there is now a new conference series on file and storage technology [6], and starting with 2004 the Supercomputing Conference, which is the biggest in that field, adds the keyword “storage” to its title [14]. A remarkable special initiative will be introduced at SC2004: StorCloud will showcase HPC storage technology and provide 1 PetaByte of randomly accessible storage to StorCloud and conference participants [32]. It was not before 2001 that a book on parallel I/O was published. Its author John May gives a comprehensive overview over research in that field [24]. There now also appeared a book edited by Daniel Reed which presents a deep insight into current research issues [31].

The remainder of the paper is organised as follows: Section 2 describes the categories of parallel input/output in typical applications. Section 3 shows how applications can use parallel I/O and section 4 lists the most recent approaches in parallel file system technology. In section 5 we will discuss the research questions that can be found in this field and will relate them to projects that are ongoing. Section 6 describes our own work, which focusses on load management in parallel file systems. Finally, we give a conclusion and make an outlook onto future work.

2. Categories of High Volume Input/Output

Applications with high volume of input and output data can be roughly divided into two classes: numerical and non-numerical applications. Typical examples of the latter are database and multimedia applications. Both belong furthermore to a program class that is always I/O-intensive, as I/O is the actual purpose of the application. We do not focus on these applications as they are not or not yet typical candidates for high performance compute clusters (For further details on their I/O behavior see e.g. [37]). Instead we concentrate on numerical applications. We will analyse the characteristic I/O situations of these programs and make statements on their I/O access patterns.

Let us first see, which are the typical situations when I/O is required. We can identify the following categories:

1. Reading of input data, writing of output data
2. Writing of checkpoint data
3. Reading and writing of scratch data during program execution
4. Out-of-core execution

The first and second are the most important ones for high performance applications. Reading input and writing output is already costly and data intensive when only the main memory has to be filled or emptied. In addition we see many applications where the actual parallel program is part of an execution pipe (just like the one you start on a command line in a Unix-based computer): they input one data set after the other and output result data sets one after the other. Some of them do both. Prominent examples are e.g. computational fluid dynamic programs that produce sequences of pictures while calculating a steady flow. An important scenario here includes post-processing of data, in particular of high amounts of result data. Even when those data are written with a parallel file system they either will be processed in a sequential computer or be transferred to another storage device. We find here a new problem space which deals with the moving of massive amounts of data between parallel and non-parallel file systems. As moving is complicated and expensive we should study concepts where the data can be left in place.

With checkpoint data the situation is less complex. We mostly write this data and its volume is equal or less the size of the main memory. For securing a single application it is sufficient to store two checkpoints. In case we want to foster preemptive execution of batch jobs in cluster environments we need space for the data of each interrupted job that has to be resumed.

Writing scratch data does not require a parallel file system as long as each node has its own local disk and can access it during execution and the data to be written is only read by its writer. In the latter case we do not need the semantics of a single file that can be accessed by many processes and in the first case we do not need a parallel file system that gives us access to non-local disks.

With out-of-core computation the situation is controversial. Out-of-core computation is always deployed when the size of the data to be computed exceeds the physical memory. In modern operating systems we have two options: We can ignore it and just rely on virtual memory. The performance penalty comes by swapping. Alternatively, we use out-of-core computation where the data is stored in files and the program manipulates the data via file access operations. This gives us a chance to integrate own optimization algorithms and we will see a better performance as with swapping. However, with high performance computing this concept is not reasonable as it makes bad use of the processor. Most programs adapt the size of their data set to the size of the physical memory not using virtual memory at all. (There are also environments where it is not even provided.)

For the first category it is now interesting to explore how the I/O activity spreads over time and space. We would like to learn details about I/O request size, about their frequency and also about the locations in logical files that get accessed. This insight can be used to optimally adapt the logical file request to the actual position of the physical file on disk. Research in this field was and is conducted by the groups of Daniel Reed (see e.g. [33]) and Rajeev Thakur, William Gropp, and Ewing Lusk (see e.g. [34]). We will come back to this issue in section 5. Before that we will examine how parallel applications perform parallel I/O and how this is technically handled by parallel file systems.

3. Application Programming Interfaces and Semantics

In this section we will analyze which are the available interfaces for parallel programs to file I/O and what semantics is provided by them. Currently we can identify three levels of abstraction, going from basic APIs to highly abstract ones (see figure 2).

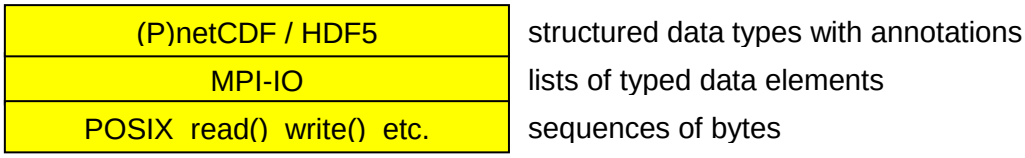

Figure 2: Hierachical abstraction layers with parallel I/O.

In fact these levels form a stack where higher levels rely on implementations of lower levels. Real parallel file systems usually introduce at least one more level which is defined by their own native API. However, if portability is an issue then our world is restricted to these three levels.

With POSIX functions we can only read and write byte streams and open and close files and position the file pointer. There is no data abstraction layer and we have to manually coordinate the I/O of complex data types as well as the portability of the resulting file. However, given an underlying parallel file system, we could profit of the striping of the stored files which will give us a better performance. Parallel access to one file by several processes is also possible, though not recommended.

MPI-IO gives us a real parallel file access semantics. MPI-IO was for some time developed independently (that´s why it has its own name) but finally became a part of MPI-2 [11]. The basic idea of MPI-IO is: Make file I/O similar to message passing. Reading is equivalent to receiving messages and writing is equivalent to sending. Derived data types can be used with I/O and we find blocking, non-blocking, and collective calls. The advantage of such an approach is evident: MPI programmers will easily transfer available know-how from message passing onto data I/O. The concept fits perfectly with what is already existing in MPI. On the other hand, the reproach that MPI is all too low level then also applies to MPI-IO. You can perform sophisticated tuning with MPI-IO but you must also learn how to do it and actually apply it in order to achieve maximum performance. Some details on MPI-IO will follow later in this section. It remains to be emphasized that using MPI-IO does not require to have a parallel file system underneath. The MPI library (e.g. MPICH [26]) is in charge of mapping file access onto available file systems, e.g. onto a standard file system on a RAID device. A widely used library implementation for MPI-IO is ROMIO [35]. It provides its own parallel access semantics and is often used as a basic layer when implementing higher abstraction layers like the two discussed next.

Although MPI-IO allows complex data types to be transferred between memory and files this is not yet sufficient. In order to achieve higher abstraction and better portability we would like to also include the description of our data types into the files themselves. By that, a subsequent reader of a file will be able to extract exactly those data types that were also written. Two such abstraction layers are currently frequently used in scientific applications:

- netCDF (network Common Data Form) [27] and
- HDF (Hierarchical Data Format) [13].

Both were developed in the late 80s. It´s beyond the scope of this paper to discuss details of this approach (see e.g. [24]), but it should be mentioned that there is active further

development with both of them: for netCFD there is now a parallelized version PnetCDF [17] and HDF5 is a major improvement over preceeding versions and also supports parallelism. Implementations are available for both APIs that map these higher abstraction layers onto MPI-IO and in particular onto ROMIO.

We now have to answer the question what parallel file access really *is* and why we want to have that. We use again MPI-IO as an example although during the years we have seen many proprietary parallel file systems that offered their own APIs which were different to MPI-IO. As MPI became a standard for message passing, MPI-IO will be the standard for parallel file I/O. So why use parallel I/O? The main reason is that by doing so we can achieve higher performance and that it is a natural way of doing I/O in parallel programs where processes compute data that usually are part of one single file. We cannot go into details with MPI-IO but we want to illustrate its principal concepts. Consider an example where a 2x2 matrix of any elements is stored in a file. Usually this is done in a row major order, i.e. we store it linewise. Figure 3.a shows this mapping. Assume that we have two processes which need to access the matrix. Process A needs the first column, process B the second. Thus they access noncontiguous segments of the file (Figure 3.b).

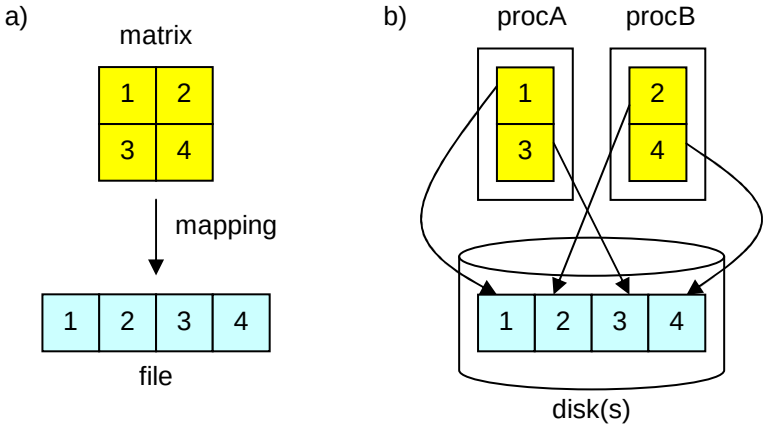

Figure 3: Simple MPI example with a 2x2 matrix, a) mapping of data into file, b) access to file by processes.

By using MPI-IO we can distinguish four different levels of how to actually access these data segments:

0. Each process reads each segment one by one.
1. All processes perform a collective call and read the segments one by one.
2. Each process reads all segments at once in a noncontiguous request.
3. All processes perform a collective call and read all their segments at once in a noncontiguous request.

Collective calls and noncontiguous access are main features of MPI-IO. A collective call synchronizes all participating processes in one call and thus places the library in a position to apply optimizations to this call. With noncontiguous access we make a single request to a complex data set with holes in it. Optimizations can be applied by reading more data than necessary and throwing away superfluous parts of it (i.e. the holes). A combination of both finally maximizes the optimizations that can be applied by the MIP-IO implementation. From level 0 to level 3 we observe a decrease in number of calls to the actual parallel system and an increase of data being transferred with each call. Note that each element of the matrix could by itself again be a complex data structure. For a further analysis of how to use MPI-IO and of how to achieve good performance refer to e.g. [12]. We will now discuss the question how the single file is partitioned and distributed over a set of disks by the parallel file system.

4. Parallel File Systems

In this section we want to shed some light onto important approaches in the domain of high performance and parallel file systems. The goal here is to characterize the current systems from the user´s point of view. Following sections will go into detail with research results.

File systems for high performance computing is a research field where only a few companies and groups are active. We saw several systems being developed in the mid-90s. Parallel computer vendors produced proprietary systems and universities several research prototypes. Details are beyond the scope of this paper. Descriptions of approaches can easily be found in literature (Refer e.g. to [24]).

Three approaches will be introduced here. The selection is based on their importance though the latter refers to different characteristics:

- PVFS/PVFS2 is currently the only powerful open source parallel file system that is available to the public and is under constant development.
- Lustre is a new file system for clusters where all the experiences from the past will be joint with new concepts in order to design and implement the high performance file system of the future.
- GPFS is a commercial system which is used on several of the most powerful parallel computers in the world.

With respect to their characterstics we can say that they are the most prominent representatives of their respective classes. That´s why they shall be introduced here.

The Parallel Virtual File System (PVFS) was started as a project at Clemson University by Rob Ross, Walt Ligon and others and is now a project between this university and Rob Ross´s group at the Argonne National Laboratories [18, 3]. The project started in the mid-90s. Their latest product is PVFS2 which is a complete rewrite of PVFS and is freely available as a beta version since November 2003. We will concentrate on PVFS2 here, as it serves as a platform for our own research. PVFS2 (as well as PVFS) is a parallel file system and offers various APIs. It has an own parallel file access semantics that is usually used from MPI-IO via ROMIO and also offers a POSIX semantics. It allows to freely select nodes with disks to serve as I/O nodes. They may as well serve as compute nodes, if the user configures it that way. PVFS2 is a major improvement over PVFS as we find a very modular implementation with many internal interfaces. There is e.g. a layer that handles communication between PVFS2 components and which can be adapted to different available networking technologies. Another interface gives access to routines that control the distribution of data over the disks. Currently a striping mechanism is used, which gives PVFS2 a RAID-0 characteristics. New functions can be plugged-in easily. As with the predecessor PVFS we can expect to see PVFS2 being installed on many clusters of varying sizes. Research issues with PVFS(2) will be discussed later on.

Lustre is a new system being developed by Cluster File Systems Inc. It is a broad research and development project where the product itself will be open source and freely available [21]. Lustre aims at scalability and availability and thus deploys an innovative distributed locking mechanism and extensive concepts for fault tolerance. In summer 2003 three of eight top supercomputers ran Lustre as their high performance I/O-system and future plans include e.g. SNL/ASCI Red Storm [2] with more than 8,000 nodes as a new installation. A comprehensive description of research issues and concepts can be found in [1].

GPFS is an older approach from IBM and belongs to a category called shared storage architectures where we do not find dedicated I/O servers with own intelligence [9]. Thus, GPFS does not hide the device layer from the user and higher abstraction software layers must take care of this. GPFS is still frequently used and in particular also on several top supercomputers. As it is not freely available and also does not incorporate the latest I/O concepts its importance will decrease in the future.

Let us now analyse which research issues exist in the field of high performance parallel I/O.

5. Research Directions with Parallel I/O

Resarch in high performance parallel I/O falls into four categories:

1. Usability
2. Increase of performance
3. Improved availability
4. Evaluation of performance and availability

Item one refers to differing semantics at the user´s API and the questions: What is appropriate for which application or class of applications? How to hide data management and how to provide more abstract though performant I/O concepts? We will not go into details with this question but concentrate here on the more technical aspects of parallel I/O.

Increase of performance is of course the main goal of any parallel I/O. Therefore we find here most of the current research questions. We see the following categories:

- Access pattern analysis and prediction is applied in order to learn how the program accesses I/O objects and functions.
- The mapping between logical and physical file layout should be targeted on an increased locality with disk access and a reduction of network traffic.
- Noncontiguous access, which is frequent in parallel programs, should be mapped onto chunk oriented disk access.
- Collective operations at user level allow to bundle many formerly independent request into one single request.
- With metadata performance we tackle the problem of distributed knowledge and how to efficiently maintain consistent information about our system.

We will go into details with these issues below.

Availability is also vital for I/O systems. As the parallel I/O system consists also of many components it suffers from the same availability problems as the clusters themselves. With respect to time we distinguish between:

- short-term availability and
- long-term availability.

Short-term availability refers mainly to nodes and disks crashing during individual program runs. The need for fault tolerance mechansims depends on the semantics of the data on disk and the overlap of partitions (i.e. sets of nodes) used for computation and for I/O. If compute and I/O nodes are identical then a crash of the system (by node failure or network failure) might result in a loss of all data. As the program also crashes this produces no additional negative effects. However, if disk data constitutes a checkpoint for program restart then measures must be taken to protect the data. Solutions usually employ RAID concepts like e.g. mirroring. In case of separate partitions we will see independent crashes in either system. A

crash of the compute partition is unproblematic as the program crashes too anyway. Data will continue to be available for resuming execution. With a crash of the I/O partition we see two aspects: Without checkpoint data being stored the final situation is just that the overall availability for the program is reduced because of the number of components being used is increased. We can presumably live with that effect. If again we store checkpoint data then the crash in the I/O partition might be allowed to also crash the program. However it is inacceptable that data gets lost. Again fault tolerance mechansims like e.g. mirroring must be applied.

With long term storage loss of data is inacceptable in any case. Thus it is mandatory to have fault tolerance mechanisms in place. As a standard scenario we will see here separate compute and I/O partitions. Fault tolerance can therefore be different in either part or be neglected in the compute partitions at all. A promising approach here is presented by CEFT-PVFS, which is a fault-tolerant add-on onto PVFS that uses mirroring. It greatly enhances data availability and thus is applicable to both, short-term storage of checkpoint data and long-term storage of any data of parallel programs. For details refer to [39].

Let us come back to the performance issue and have a closer look onto the categories of research. Access pattern analysis and prediction is an issue that has been studied since the beginning of the 90s already. The question here is how to categorize the access patterns with respect to time and space and to describe when an application access which data. Results can be used to optimize performance by joining misleadingly assumed to be independent requests and by applying intelligent caching mechanisms. Much research here was conducted in Daniel Reeds group. See e.g. [23] for more details.

Related to the above mentioned is the question of how to map logical parts of a file onto physical parts. Most systems support only striping as a low level partitioning concept. The question reduces to how to find an optimal striping factor. A few other research prototypes like e.g. Clusterfile [8] support any partitioning concept. The correct tuning of the mapping is then much more versatile but it remains the problem of how to tune it optimally. The actual mapping function chosen here has a crucial impact on the overall performance. PVFS2 with its increased modularization also enables users to plug in their own mapping functions.

With parallel applications we find that processes access data which is stored noncontiguously in the physical file. For example if the process accesses one column of a matrix this might be translated into several request of just a single row element in case that the physical storage of the matrix is oriented row-wise. Instead of sending these requests to disks independently we could also read a bigger chunk of data which comprises all elements needed and throw away the unnecessary part of it. As transferring and processing data is usually much faster than the latency part of disk access this will result in improved overall performance. Support for noncontiguous access is built into ROMIO and thus is available for any MPI-based program [36]. The concept here is called data sieving. We find other researchers being active here, too [38]. It could be an interesting issue of making sieving more dynamic and base the sieving parameters onto runtime performance metrics.

Collective operations is a problem that relates to sieving. Whenever different processes do logically the same thing, e.g. read a matrix column, it will be advantageous to group them together and add means to improve performance. One such mechanism is integrated into ROMIO and called two-phase protocol. It handles collective operations as a combination of internal message passing and accessing of disks. A collective read is decomposed into a phase where the data is read as a big chunk from the physical disks followed by phase where it is

sent to its respective process. With collective writes we group data together via internal messages and subsequently write it to disk. Obviously in ROMIO both mechanisms for collective calls and noncontiguous data access work closely together. Both are subject to further research and tuning.

A hard problem for every parallel file system is the fact that data is distributed over nodes whereas the information what the data means and what belongs together must be stored in a central location. This component is usually called a metadata server. For parallel files it holds information on access rights and time stamps, partitioning concepts used, locations of physical file segments and size. Having one metadata server we run into the following problem: With every access to the logical file by a process the metadata server has to be contacted. As there usually is only one such server we see network contention, server overloading, and thus a dramatic decrease in performance due to the serialization of requests in this component. (Consider also the situation where 512 processes open a file; without support for collective operations this would result in 512 independent requests to the metadata server). So what could be an efficient solution to this problem? We don´t know yet. Currently, there are only proposals but no implementation has proven to be scalable with respect to metadata server operations. PFVS2 includes a concept where this server exists in several instances, each serving different logical files. The selection of the server is based on a hashing over the file name. This alleviates performance problems and better balances the load when working with many files. For an individual file where a high number of client processes perform concurrent access, this is however no solution. New concepts have to be developed that deploy intelligent caching mechanisms for metadata of files.

Last but not least how will all this be evaluated? There is not yet a standardized benchmark suite by which we could compare different high performance parallel I/O systems. There is an initiative by the PVFS developers group but there is no real progress yet [28]. Just recently there seems to be some activity where I/O benchmarking could also be included [25]. Without powerful benchmarks it is not possible to compare performance impacts of concepts deployed for above listed problems in a sophisticated way. In the meantime we use e.g. b_eff_io, a benchmark developed by Rolf Rabenseifner [4, 30].

Summing up we can say that there are many open research problems in the field of parallel high performance I/O in cluster environments. In many cases they are well described already such that it should only be a matter of time when we will see further powerful solutions.

6. Load Management in Parallel I/O Systems

The author´s group conducts research in three connected areas:

- Enabling parallel I/O technologies in selected scientific applications
- Management of load in parallel file systems
- Evaluation and improvement of metadata performance in parallel file systems

We will present here the work in the field of load management. As we just started to intensify our investigations there are no publishable results available by now. Details will be given at the conference.

Load management in parallel file systems starts from the consideration that the mapping of logical file segments onto their physical counterparts is of crucial importance for the overall performance. Furthermore it seems advisable that the physical spread of a file can be

controlled during runtime of the accessing program. In such a way the number of nodes utilized can be adapted to needs of the programmer and/or needs of the system administrator.

In order to bring dynamics to the physical file layout we need to develop and implement two mechanisms. One relates to the mapping function of the actual file which gives us details on where to find physical parts of it. The second allows us to migrate file parts from source nodes to target nodes. Both mechanisms are connected in a way that when a migration is triggered both of them will have to react in a coordinated way and during remapping and migration any file access must be deferred. The mechanisms applied here are closely related to those used for preemptive process migration [19] where running processes are moved from overloaded nodes to underloaded nodes. Algorithms applied must be restricted in runtime and will have to deal with a variety of potential deadlock situations.

For controlling file migration we need a measurement and decision unit. In combination with migration they constitute a control loop, similar to the one for load balancing by process or data migration (see figure 4).

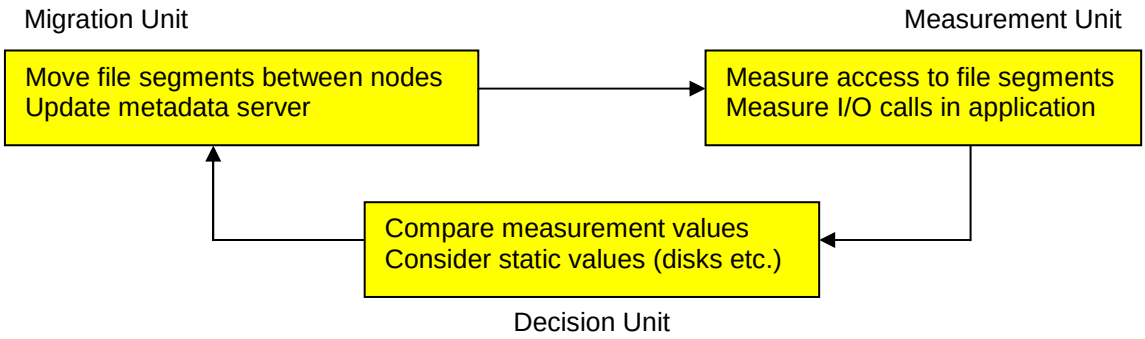

Figure 4: Control loop with measurement, decision, and migration unit.

Basically we identify two levels on which measurements must be performed: the logical file level, defined by the user, and the physical file level. Consequently, we need to instrument both, the MPI library and the PVFS library. Measurements on these levels must be related such that we can evaluate the influence of certain I/O-calls in the program to disk activities on the I/O-nodes. As for the low level measurements we have to distinguish between non-shared and shared resources, i.e. nodes that are used by one user only or by several users. With the latter we must also take into consideration aspects of fairness when making load balancing decisions.

Measurements are fed into a decision component that also has knowledge about static properties of the underlying system, e.g. number of nodes, sizes of disks, etc. A comparison of selected measures with threshold values will trigger file migrations. In addition to the automatic control loop we support application triggers where the program can control its file layout by using the MPI “Info” mechanism which passes information from the application to the libraries.

For the evaluation of the load management quality and behavior we adapt a visualization component. Trace-based and on-line oriented concepts will equally be applied and well-known monitoring and instrumentation techniques will be deployed [20]. A functional prototype of the load management environment will be available by end of this year.

Conclusion and Future Work

Parallel high performance I/O is an issue that holds many challenging and interesting research questions. There are not too many groups world-wide being active in this field although it is of increasing practical relevance. While active research in this field being neglected we see already new areas appearing, in particular with Grid computing. In addition to research issues in clusters we find here also Grid-related problems like heterogeneity of environments, security, and others. First concepts are already available and make their way to discussions in the Global Grid Forum. While the more hidden technical aspects are still under heavy investigation it seems that there is some consolidation at the lower abstraction user level: MPI-IO is widely used on clusters as well as on Grids. Which approach at the higher abstraction level, e.g. parallel netCDF, HDF5, or something else, will win recognition is still to be seen. Future work might provide application programers with powerful though easy to use APIs. Finally, what we would like to see is a powerful benchmark suite that allows a ranking of supercomputers taking into account the I/O performance.

References

[1] Peter Braam: The Lustre Storage Architecture. http://www.lustre.org/docs/lustre.pdf

[2] William Camp, James Tomkins: Thor´s Hammer/RedStorm. http://www.lanl.gov/orgs/ccn/salishan2003/pdf/camp.pdf

[3] Phil Carns, Walt Ligon III, Rob Ross, Rajeev Thakur: PVFS – A Parallel File System For Linux Clusters. In Proceedings of the 4th Annual Linux Showcase and Conference, Atlanta, GA, October 2000, pp. 317-327.(Best Paper Award).

[4] Effective I/O Bandwith Benchmark. http://www.hlrs.de/organization/par/services/models/mpi/b_eff_io/

[5] European Conference on Parallel Computing Euro-Par 2003, Munich, Germany. http://wwwbode.cs.tum.edu/archiv/Euro-Par2000/

[6] Conference on File and Storage Technologies FAST2002. http://www.usenix.org/events/fast02/

[7] Florin Isaila: An Overview of File System Architectures. In “Algorithms for Memory Hierarchies", Lecture Notes on Computer Science, Volume 2625, Springer Verlag, 2003, pages 273-289.

[8] Florin Isaila, Walter F. Tichy: Clusterfile: A Flexible Physical Layout Parallel File System. In “Concurrency & Computation: Practice & Experience”, John Wiley & Sons Ltd, Volume 15, Issue 7-8 (June - July 2003), pp. 653-679.

[9] General Parallel File System for Linux (GPFS). http://www.ibm.com/servers/eserver/clusters/software/gpfs.html

[10] Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung: The Google File System. In “Proceedings of the 19th ACM Symposium on Operating System Principles”, 2003. http://www.cs.rochester.edu/sosp2003/papers/p125-ghemawat.pdf

[11] William Gropp, Steven Huss-Lederman et al.: MPI – The Complete Reference: Volume 2, The MPI-2 Extensions. MIT Press, Cambridge, MA, 1998.

[12] William Gropp, Ewing Lusk, Rajeev Thakur: Using MPI-2: Advanced Features of the Message-Passing Interface. MIT Press 1999.

[13] HDF5 – A new generation of HDF. http://hdf.ncsa.uiuc.edu/HDF5/

[14] High Performance Computing, Networking, and Storage Conference SC2004, Pittsburgh, USA. http://www.sc-conference.org/sc2004/

[15] IBM Blue Gene Project. http://www.research.ibm.com/bluegene/

[16] LHC – The Large Hadron Collider. lhc-new-homepage.web.cern.ch/lhc-new-homepage/

[17] Jianwei Li et al.: Parallel netCDF – A high performance scientific I/O interface. In proceedings of the 2003 Supercomputing Conference, Phoenix, USA. http://www.sc-conference.org/sc2003/paperpdfs/pap258.pdf

[18] Walt Ligon III, Rob Ross: PVFS – Parallel Virtual File System. In: Beowulf Cluster Computing with Linux, Thomas Sterling, editor, pages 391-430, MIT Press, November, 2001.

[19] Thomas Ludwig: Load Management for Process Objects. In Proceedings of the Workshop on Dynamic Object Placement and Load Balancing in Parallel and Distributed Systems, Utrecht, the Netherlands, pp. 3-7, Katholieke Universiteit, Leuven, Belgium, 1992.

[20] Roland Wismüller, Thomas Ludwig, Wolgang Karl and Arndt Bode: Monitoring Concepts for Parallel Systems – An Evolution towards Interoperable Tool Environments. In Parallel and Distributed Computing Practices, volume 4, number 3., 2002.
[21] Lustre. http://www.lustre.org
[22] Peter Lyman, Hal R. Varian: "How Much Information? 2003", Berkeley University, 2003. http://www.sims.berkeley.edu/research/projects/how-much-info-2003/
[23] Tara Madhyastha, Daniel Reed: Learning to Classify Parallel Input/Output Access Patterns. IEEE Trans. Parallel Distrib. Syst. 13(8): 802-813.
[24] John May: Parallel I/O for High Performance Computing. Morgan Kaufmann Publishers, San Francisco, USA, 2001.
[25] Rick Merritt: Darpa to overhaul supercomputing benchmarks by 2006. http://www.eetimes.com/sys/news/OEG20031114S0035
[26] MPICH – A Portable Implementation of MPI. http://www-unix.mcs.anl.gov/mpi/mpich/
[27] NetCDF (network Common Data Form). http://my.unidata.ucar.edu/content/software/netcdf/
[28] Parallel I/O Benchmarking Consortium. http://www-unix.mcs.anl.gov/~rross/pio-benchmark/
[29] Parallel Virtual File System (PVFS). http://www.pfvs.org
[30] Rolf Rabenseifner, Alice Koniges: Effective File-{I/O} Bandwidth Benchmark, In Proceedings of the European Conference on Parallel Processing, pp. 1273-1283, 2000.
[31] Daniel Reed (editor): Scalable Input/Output – Achieving System Balance. MIT Press, 2004.
[32] The SC2004 StorCloud-Initiative. http://www.sc-conference.org/sc2004/storcloud.html
[33] Evgenia Smirni, Daniel A. Reed: Workload characterization of input/output intensive parallel applications. In "Proceedings of the Conference on Modelling Techniques and Tools for Computer Performance Evaluation", Volume 1245, Springer Verlag, 1997, pages 169-180.
[34] Rajeev Thakur, Ewing Lusk, William Gropp: I/O in Parallel Applications: The Weakest Link. In the International Journal of High Performance Computing Applications, Volume 12, Number 4, 1998, pages 389-395.
[35] Rajeev Thakur, Ewing Lusk, and William Gropp: Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation. Technical Memorandum ANL/MCS-TM-234, Mathematics and Computer Science Division, Argonne National Laboratory, Revised July 1998.
[36] Rajeev Thakur, William Gropp, Ewing Lusk: Data Sieving and Collective I/O in ROMIO. In Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation, February 1999, pp. 182-189.
[37] Mustafa Uysal, Anurag Acharya, Joel Saltz: Requirements of I/O Systems for Parallel Machines – An Application-Driven Study. Technical Report CS-TR-3802, Dept. of Computer Science, University of Maryland, College Park, MD, May 1997.
[38] Joachim Worringen, Jesper Larsson Träff, Hubert Ritzdorf: Improving Generic Non-Contiguous File Access for MPI-IO. In Recent Advances in Parallel Virtual Machine and Message Passing Interface. 10th European PVM/MPI Users' Group Meeting, volume 2840 of Lecture Notes in Computer Science, pages 309-318, 2003.
[39] Y. Zhu, H. Jiang, X. Qin, D. Feng and D. Swanson: Design, Implementation, and Performance Evaluation of a Cost-Effective Fault-Tolerant Parallel Virtual File System. In Proceeding of International Workshop on Storage Network Architecture and Parallel I/Os, in conjunctions with 12th International Conference on Parallel Architectures and Compilation Techniques, New Orleans, LA, Sept. 27 - Oct. 1, 2003.