Design and Analysis of Parallel Disk Accesses in ViPIOS*

Kurt Stockinger†, Erich Schikuta, Thomas Fuerle, and Helmut Wanek
Institute for Applied Computer Science and Information Systems
Department of Data Engineering, University of Vienna,
Rathausstr. 19/4, A-1010 Vienna, Austria

Abstract

Due to the shift from CPU-bound to I/O bound problems the performance of the disk I/O accesses of parallel programs is a key factor for the success of solution approaches. The Vienna Parallel Input Output System (ViPIOS) [14] is a client-server based system to speed up the disk accesses for high performance programs by combining characteristics of parallel I/O runtime libraries and parallel file systems. In this paper we present the data access principle followed in ViPIOS to reach high I/O bandwidth. The underlying data layout policy and the disk access profile is explained. We justify our approach by an exhaustive performance and scalability analysis.

Keywords:

High performance computing, parallel I/O, runtime systems, scalability, performance analysis

1 Introduction

In the last few years the applications in high performance computing (Grand Challenges [2]) shifted from being CPU-bound to be I/O-bound. Performance can not be scaled up by increasing the number of CPUs any more, but by increasing the bandwidth of the I/O subsystem. This situation is commonly known as the I/O bottleneck in high performance computing ([9])

In designing parallel I/O support for high performance languages two issues have to be addressed, which we call *static fit property* and *dynamic fit property*.

The static fit property tries to adapt the disk access profile of the program accordingly to the necessities of the problem specification or the programming paradigm (as the Single Program Multiple Data SPMD approach). This can be realized by the programmer or the compiler. For example, a specific SPMD distribution of the application is resembled by a corresponding data layout on disks.

The dynamic fit property denotes the adaptability of the I/O runtime subsystem to application's and/or environment's characteristics which could not be foreseen at compile time. Basically three typical situations can be identified, namely *access profile mismatch*, *environment mismatch*, and *system mismatch*.

An access profile mismatch exists if during execution the application generates a disk access profile different from the one expected while compilation. We call it environment mismatch if the application is running on a different environment than being compiled for. This is the typical situation if system resources requested by the application (e.g. number or type of disks or CPUs) are not available. A third type of mismatch (system mismatch) can arise in an environment with multiple applications sharing the disk subsystem. Different disk access profiles of the respective application or of independently executing applications can generate bottlenecks on specific disks. This situation has to be addressed by a re-distribution of the data or the resources.

We see a solution to these problems in a dedicated, smart, concurrent executing runtime system, which gathers all available information of the application process both during the compilation process and the runtime execution. Thus, this system is able to to fulfill the needs for static and dynamic fit property. Initially it can provide the optimal fitting data access profile for the application (static fit) and may then react to the execution behavior dynamically (dynamic fit), allowing to reach optimal performance by exploiting maximum I/O bandwidth.

This approach led to the design and development of the Vienna Input Output System, ViPIOS ([14]).

The remainder of this paper is organized as follows. Section 2 introduces to the state of the art of parallel I/O. Section 3 describes the overall system architecture of ViPIOS. Section 4 gives a justification of our approach by an exhaustive runtime analysis of its performance showing ViPIOS's scalability behavior. Conclusions and prospects for future work are given in section 5.

*This work was carried out as part of the research project "Language, Compiler, and Advanced Data Structure Support for Parallel I/O Operations" supported by the Austrian Science Foundation (FWF Grant P11006-MAT)

†Now at IT Division, CERN, CH-1211 Geneva 23, Switzerland. Email: Kurt.Stockinger@cern.ch

2 State of the Art

Over the last couple of years several approaches for overcoming the so-called I/O bottleneck evolved which can be categorized into file systems, I/O libraries and other independent systems [17].

As for file systems the operating system is enhanced by features that deal directly with I/O at the level of files. The advantage of this approach is that the application programmer need not care for the basic problems of file I/O since this is accomplished by the operating system, i.e. the file system. Examples of parallel file systems are Galley [10], HFS [11] , HiDIOS [19], ParFiSys[4], PIOUS [12] (based on PVM []) and Vesta [8].

Looking at one level above the file system, we come to the second category, namely the I/O libraries. In short, this method enhances existing programming languages by features dealing with file I/O. Thus, the application programmers can use conventional programming languages such as C, C++ or FORTRAN and also perform efficient I/O by mean of additional I/O functions rather than having to learn new concepts. Again we give some examples of this category, which are ADIO [18], Jovian [3], Panda [15], PASSION [6], MPI-IO [7] and ViPIOS [14] .

The last category comprises all approaches which do not fall into the first two ones. These approaches cannot be put into any common schema because they use very different techniques for tackling the I/O bottleneck problem. Experimenting testbeds and simulation software for products that can be used to classify existing file systems and I/O libraries [17]. Mostly these testbeds are used for studying and comparing the performance and usability of different systems.

3 The ViPIOS Approach

The Vienna Parallel Input Output System (ViPIOS) is an I/O runtime system, which provides efficient access to persistent files, by optimizing the data layout on the disks and allowing parallel read/write operations. ViPIOS is targeted as a supporting I/O module for high performance languages (e.g. HPF).

3.1 System Architecture

The ViPIOS system architecture is built upon a set of cooperating server processes, which accomplish the requests of the application client processes. Each application process AP is linked by the ViPIOS interface VI to the ViPIOS servers VS (see Figure 1).

The server processes run independently on all or a number of dedicated nodes on the underlying MPP. Generally speaking the ViPIOS server processes are executed on the I/O nodes of the underlying system. It is also possible that

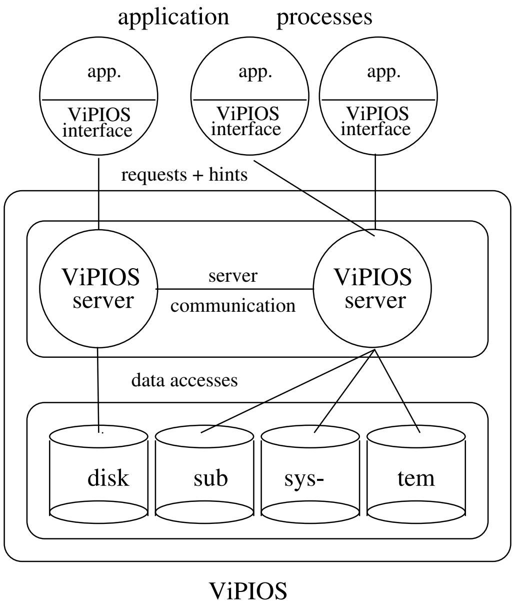

Figure 1. ViPIOS system architecture

an application client and a server share the same node if the system does not support dedicated I/O facilities.

In most cases each application process is assigned exactly one ViPIOS server, but one ViPIOS server can serve a number of application processes, i.e. there exists many-to-one relationship between the applications and the server.

3.1.1 Data Locality

Data locality is the main design principle of the ViPIOS to achieve high data access performance. This means that the data requested by an application process should be read/written from/to the best-suited disk.

In this context we distinguish between logical and physical data locality.

Logical data locality denotes the ability to choose the best suited ViPIOS server for an application process. This server is defined by the topological distance and/or the process characteristics. In general the access time is proportional to the topological distance of the application process to the ViPIOS server in the system network. It is also possible that special process characteristics can influence the ViPIOS server performance, like available memory, disk priority list of the underlying node (see the following), etc. Therefore it is also possible that a remote ViPIOS server could provide better performance than a closer one. However, only one specific ViPIOS server is chosen for each application process, which handles the respective requests. This process is called the *buddy server*, whereas all other servers are called *foe servers*. The buddy and foe relation is illustrated

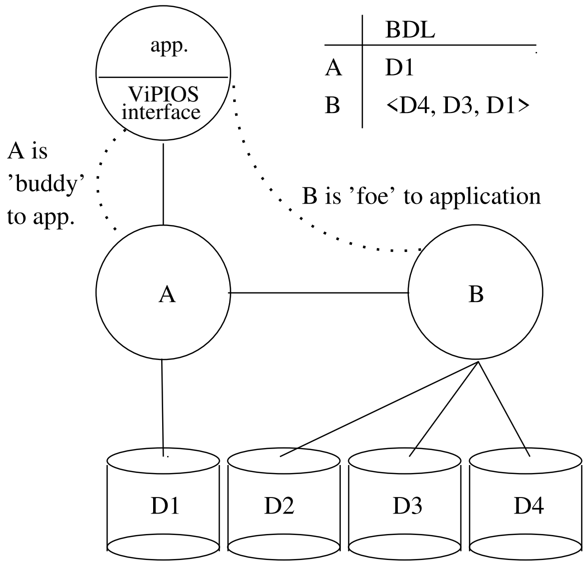

Figure 2. "Buddy" and "Foe" relation between application processes and ViPIOS Servers

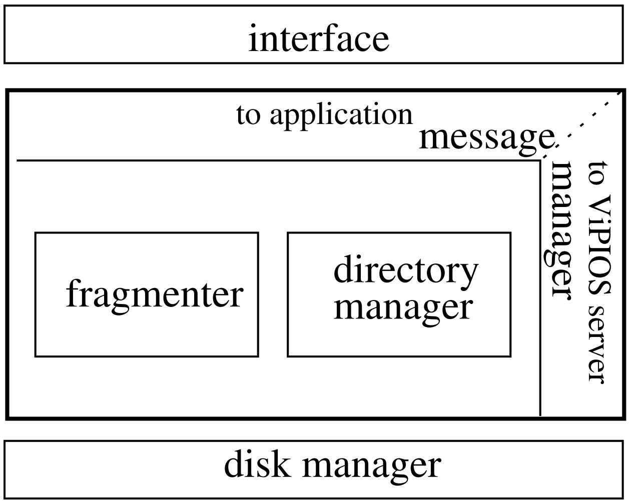

Figure 3. ViPIOS server architecture

by Figure 2.

The goal of the principle of *physical data locality* is to determine the disk set which provides the best (mostly the fastest) data access. For each node an ordered sequence of accessible disks of the system is defined (the *best disk list, BDL*), which in turn orders the disk according to their access behavior. Disks with good access characteristics precede disks with bad access characteristics in this list (Figure 2).

3.1.2 ViPIOS Server

A ViPIOS server process consists of several functional units as depicted by Figure 3, namely:

- The **Interface** provides the connection to the "outside world"(i.e. applications, programmers, compilers, etc.). Different interfaces are supported by *interface modules* to allow flexibility and extendibility. Up to now we implemented an HPF interface module (suitable for the VFC, the HPF derivative of Vienna FORTRAN [5]) a (basic) MPI-IO interface module, and the specific ViPIOS interface, which is in turn the interface for the specialized modules.

- The **Message manager** is responsible for the external (to the applications) and the internal (to other ViPIOS servers) communication.

- The **Fragmenter** can be regarded as "ViPIOS's brain". It represents a smart data administration tool, which models different distribution strategies and makes decisions on the effective data layout, administration, and ViPIOS actions.

- The **Directory Manager** stores the meta information of the data. In particular, we designed 3 different modes of operation, which are centralized (one dedicated ViPIOS directory server), replicated (all servers store the whole directory information), and localized (each server knows the directory information of its local data) management. At present merely localized management is implemented.

- The **Disk Manager** provides the access to supported disk sub-systems. What is more, this layer is modularized in order to allow extendibility and to simplify the porting of the system. Currently the Disk Manager supports modules for ADIO [18], MPI-IO, and Unix style file systems.

3.1.3 Requests and Messages

Requests are issued by an application via a call to one of the functions of the ViPIOS interface, which in turn translates this call into a request message. Finally, this request message is sent to the buddy server.

The *local directory* of the buddy server holds all the information necessary to map a client's request to the physical files on the disks. The *fragmenter* uses this information to decompose (*fragment*) a request into eihter sub-requests which can be resolved locally or sub-requests which have to be communicated to other ViPIOS-servers (foe servers). The *I/O subsystem* actually performs the necessary disk accesses and the transmission of data to/from the AP.

3.2 Data Access Modes

One of the main design principles of the ViPIOS is the parallel execution of disk accesses. In the following section we present the disk access types which are supported by our system.

According to the SPMD programming paradigms parallelism is expressed by the data distribution scheme of the HPF language in the application program. Basically, ViPIOS has to direct the application process' data access requests to independent ViPIOS servers in order to provide parallel disk accesses. However, a single SPMD process, which performs its data accesses sequentially, sends its requests to just one server. Depending on the location of the requested data on the disks in the ViPIOS system, we differentiate between two access modes (see Figure 4):

- Local data access,
- Remote data access.

Local data access describes the case where the buddy server can resolve the requests from the applications on its own disks (the disks of its BDL). We call it also *buddy access*.

Remote data access denotes the access scheme where the buddy server can not resolve the request on its own disks but has to broadcast the request to the other ViPIOS servers to find the owner of the data. The respective server (foe server) accesses the requested data and sends it directly to the application via the network. We call this access also *foe access*.

Intuitively a remote access should be slower than a local access because of the additional broadcast message. As a consequence we would assume that the local data access should be increased whereas the remote data access should be decreased in order to gain optimal performance. However, the performance analysis will falsify this assumption.

Based on these access types 3 three disk access modes can be distinguished in ViPIOS, namely:

- Sequential,
- Parallel, and
- Coordinated mode.

Sequential Mode. The sequential mode of an operation allows a single application process to send a sequential read/write operation, which is processed by a single VIPIOS server in sequential manner. The read/write operation consists commonly of processing a number of data blocks, which are placed on one or a number of disks controlled by the server itself (disks belonging to the best-disk-list of the server).

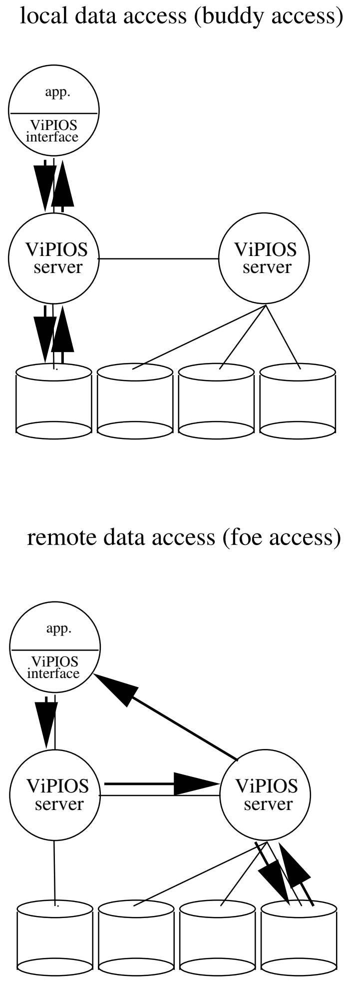

Figure 4. Local versus remote data access

Parallel Mode. In the parallel mode the application process requests a single read/write operation. The ViPIOS processes the sequential process in parallel by splitting the operation in independent sub-operations and distributing them onto available ViPIOS server processes.

Thus, independent servers can either access contiguous blocks of memory in parallel or they can distribute the file onto a number of disks which in turn are controlled by these servers.

Coordinated Mode. The coordinated mode is realized according to the SPMD approach. In particular, collective operations are used where read/write operations are requested by a number of application processes collectively. In fact, each application process is requesting a single sub-operation of the original collective operation. These sub-operations are processed by ViPIOS servers sequentially, which automatically results in a parallel execution mode.

It is also possible that the sequential server operations are processed in parallel by parallelization by the server itself. This can be accomplished by distributing the data onto other servers.

4 The Performance of ViPIOS

A prototype implementation of ViPIOS is completed. In particular, ViPIOS was implemented on a workstation cluster environment of 8 LINUX Pentium-133 PCs (nodes) connected by a 100 Mbit Ethernet network. We used MPI as communication standard and accessed the disks of the respective workstations via a Unix file system. Each PC had 32 MB main memory and a hard disk of 1 GByte. Compared to currently available super computers an environment like this offers the following advantages for testing purposes:

- low hard- and software costs
- simple administration
- availability (the system can be dedicated to the test program so results are not influenced by changing workloads etc.)

Furthermore the Beowulf projects [1] [13] [16] have shown that due to some enhancements in the network topology such PC-networks can provide peak performance in excess of 1 GOPS and also high disk bandwidth.

Even this early ViPIOS implementation shows promising performance results. In order to test ViPIOS, we carried out several experiments as follows and measured the access times for the different supported file access modes presented in this paper. The times shown are mean values for samples of at least 20 test runs, which were repeated until the standard deviation fell under a certain threshold to filter ”outliers”.

4.1 Local versus Remote Access

To analyze the performance behavior of local versus remote access we implemented the following test profile. A data set of 400 MB (resp. 800 MB) was read from the disks in 2 chunks. The 2 read operations were synchronized, i.e. the second one was started after the first one was completed. We distinguished between 3 situations,

- first both chunks were read locally,
- then the first one locally and the second one in a remote remote, and
- finally both chunks were read in a remote way.

If the remote accesses are slower than the local accesses, as suspected in the section 3.2, obviously increasing execution times should be measured. The times for the test are shown in Figure 5. The contribution of

- the pure read time from disk,
- the time for sending data messages from the server to the application, and
- the time for administration messages (non data messages) for the broadcast requests between the servers and directory accesses

to the whole access time is given. Interestingly, remote accesses prove to be of similar performance as local accesses. A closer look at the contribution of each factor makes this situation clear. The times for disk accesses and the times for sending data from the server processes to the applications is the same for local and remote accesses. A remote access adds the time for the broadcast message from the buddy to foe server. However, these times for broadcasting messages are very small in comparison to the data administration times. Even more it proves that remote accesses are a fraction faster than local accesses, because administration messages between buddy and client can be done in parallel with the data accesses of the foe. This result is also yielded by the ”Two-Phase-Access” principle [6], which states that it is preferable to access data where it is stored and transfer it via the network to the requesting process rather than re-distributing the data to the local disk and access it there.

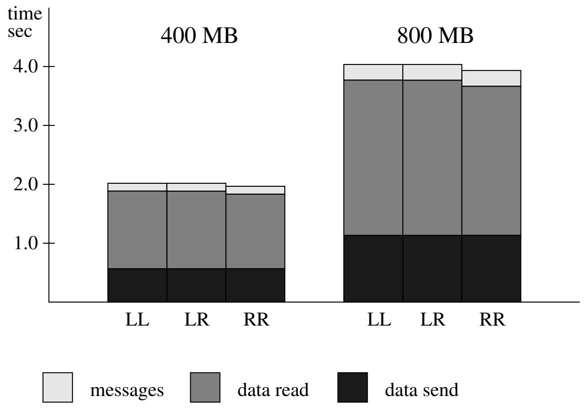

Figure 5. Local versus remote data access behavior

4.2 Parallel Mode

For the analysis of the parallel mode we used a similar test set-up as in section 4.1. The difference is that the accesses to the 2 chunks of data are not synchronized but requested at once. This allows the local-remote accesses to be performed in parallel (one chunk on the buddy, the other on the foe server). This situation is depicted by Figure 6. The local-remote accesses show performance gain over the local-local and remote-remote accesses. This can be explained by the fact that local-local as well as remote-remote data access can only be performed in a sequential way when one server has to accomplish this task. We learn that we have to distribute the data of a client among the available servers and thus being able to access the data according to the parallel mode.

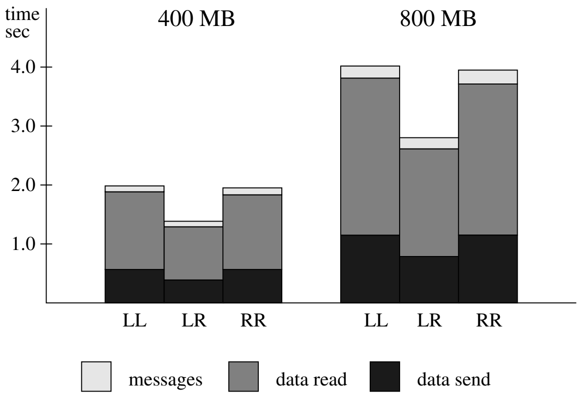

Figure 6. Parallel mode behavior

4.3 Coordinated Mode

First, we ran some scalability tests where we changed the number of server processes whereas the number of client processes were kept constant (see Figure 7). Moreover, we used dedicated I/O nodes which means that each process (either server or client) ran on a different node. The motivation for these experiments was to find out whether an increased number of servers which handle the disk I/O yields a better performance.

Second, we experimented with non dedicated I/O nodes which means that on every node one server and one client process were running concurrently. The results forced us to differentiate between the following two cases:

- The client process ran on the same node as its associated server process as depicted in Figure 8.
- Each client process was connected to a server process on a different node as depicted in Figure 9.

4.4 Dedicated I/O nodes

The workload for our first experiment was 400 MB. Since we ran an SPMD program with four clients, the task of each

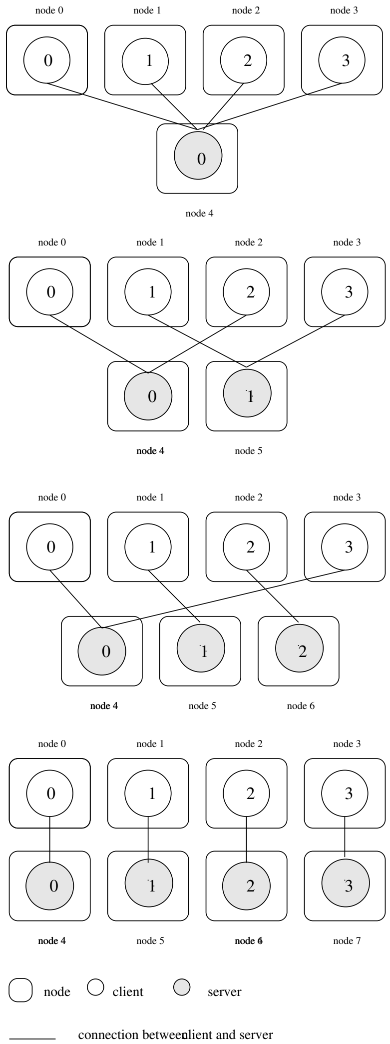

Figure 7. Topologies with different number of servers

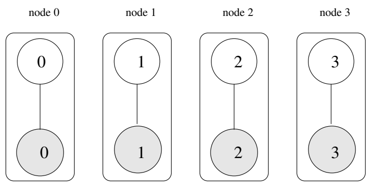

Figure 8. Servers and clients reside on the same node

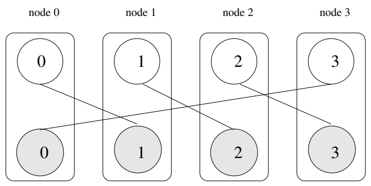

Figure 9. Servers and clients reside on different nodes

Table 1. Scalability results for dedicated I/O nodes

clients	servers	max	min	mean	variance
4	1	147.98	136.02	143.47	9.1245
4	2	86.67	76.87	81.73	5.7647
4	3	86.52	76.48	80.87	6.3144
4	4	48.00	42.17	44.18	2.6367

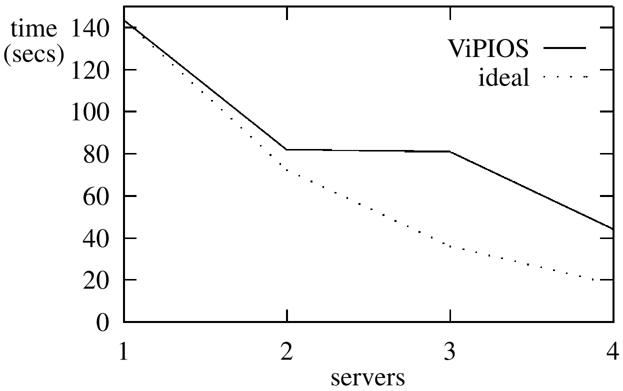

Figure 10. Scalability results

client was to read 100 MB without overlap. In other words, client 0 read the first quarter of the file, client 1 read the second quarter and so on. In order to measure the overall time for reading the whole file, the concurrently running processes were synchronized before and after the read request. By imposing these two barriers we guaranteed that the time to read the whole file corresponded to the reading time of the slowest process.

The whole process was iterated 50 times. In order to suppress any caching effects, data was not only read from one file but from 10 different files in a round robin fashion. Thus, during the first iteration file1 was read, during the second iteration file2 etc. After 10 iterations file1 was read again. It is important to state that each server read its data locally from the node it was running on.

In addition to the mean we measured the maximum, minimum and variance of the time required for reading the whole file. The topology of the server and client processes corresponded to the last topology in Figure 7. The number of client processes was set to 4 whereas the number of server processes was increased from 1 to 4. The results are given in Table 1.

The graph in Figure 10 depicts the mean time in seconds to read the file in relation to the number of server processes. There seems to be no significant performance gain by increasing the number of servers from two to three. But remember that clients are synchronized in order to measure the time it takes to read the 400 MB file completely. In the case with three servers there is one server with two client processes attached. This server has to read and transfer the same amount of data (200 MB) as any server in the two servers case. So it takes approximately the same time to accomplish the operation. The other two server-client pairs are idle almost half of the time. In fact, there is a performance gain by introducing a third server but this will only show up if clients are allowed to work asynchronously.

A comparison with the ideal linear scaling (dotted line in Figure 10) shows that ViPIOS scales well with an increasing number of servers.

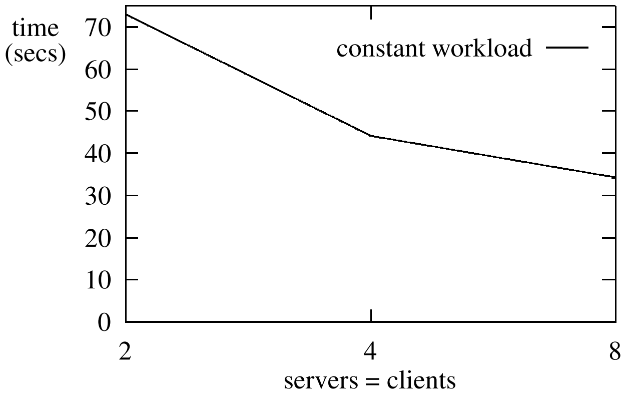

Figure 11. Non dedicated I/O nodes

4.5 Non dedicated I/O nodes

Next, we studied the results when a server process and the corresponding client process were running on the same node (see Figure 8). In these experiments the number of the client processes was always kept equal to the number of server processes in order to ensure that the work done on every processing node is the same. The results were found to be strongly dependent on the topology of the ViPIOS. If the client and the server connected to it were executing on the same node (like shown in Figure 8) then the performance was very poor. If every client was connected to a server which ran on a different node (Figure 9) the performance was boosted by a factor greater than 10. The reason for this strange behavior was found to be the message passing via MPI. The performance measurements for the second case are shown in Table 2 for a constant workload of 400 MB. That means that with 2 clients every client read 200 MB, with 4 clients every client read 100 MB and so on.

Table 2. Constant workload

clients	servers	max	min	mean	variance
2	2	87.92	62.20	72.98	0.0734
4	4	48.00	42.17	44.18	2.6367
8	8	46.53	27.18	34.29	0.0413

As can be clearly seen in Figure 11 the ViPIOS still scales for a constant workload, even if the I/O nodes are not dedicated. This can easily be explained by the increasing network contention. By using a more elaborate network topology these problems may well be overridden. What is more, the interconnection between the computing nodes will yield a better bandwidth in a real super-computing environment.

5 Conclusions and future work

We presented the ViPIOS system and gave a performance analysis of the different access modes which are supported.

We learn from these performance tests some guidelines, which have to be followed by the administration of the data by the fragmenter modules of the ViPIOS servers.

These guidelines are:

- Local and remote accesses show similar performance results. It is not necessary to prefer the one to the other. Even more, remote accesses are necessary in order to exploit the performance of the parallel mode.
- Parallel accesses allow an increase in I/O bandwidth. In addition, they speed-up the performance of the system. The data of a single application has to be distributed among available servers and accesses have to be split up into local and remote ones.
- The scalability analysis showed that the usage of a higher number of servers can increase the I/O bandwidth of an application. This result can be extended to disks as well, i.e. more disks allow higher bandwidth.

These results are important for the design of the ViPIOS's fragmenter. This module directs the disk layout according to information given by the applications.

In the near future we will test ViPIOS on real supercomputers and compare it to other parallel I/O systems (PASSION, PIOFS, ...). We will also study the effects of runtime optimizations (like data redistribution etc) and the behavior of ViPIOS when confronted with real life applications.

References

[1] *http://cesdis1.gsfc.nasa.gov/linux-web/beowulf.html.*

[2] A Report by the Committee on Physical, Math., and Eng. Sciences Federal Coordinating Council for Science, Eng. and Technology, *High-performance computing and communications, grand challenges 1993 report*, pp. 41 – 64, Committee on Physical, Math., and Eng. Sciences Federal Coordinating Council for Science, Eng. and Technology, Washington D.C., October 1993.

[3] Robert Bennett, Kelvin Bryant, Alan Sussman, Raja Das, and Joel Saltz, *Jovian: A framework for optimizing parallel I/O*, Proceedings of the Scalable Parallel Libraries Conference (Mississippi State, MS), IEEE Computer Society Press, October 1994, pp. 10–20.

[4] J. Carretero, F. Pérez, P. de Miguel, F. García, and L. Alonso, *I/O data mapping in* ParFiSys: *support for high-performance I/O in parallel and distributed systems*, Euro-Par '96, Lecture Notes in Computer Science, vol. 1123, Springer-Verlag, August 1996, pp. 522–526.

[5] B.M. Chapman et al., *Vienna FORTRAN compilation system*, User's Guide, Vienna, 1994.

[6] Alok Choudhary, Rajesh Bordawekar, Michael Harry, Rakesh Krishnaiyer, Ravi Ponnusamy, Tarvinder Singh, and Rajeev Thakur, *PASSION: parallel and scalable software for input-output*, Tech. Report SCCS-636, ECE Dept., NPAC and CASE Center, Syracuse University, September 1994.

[7] Peter Corbett, Dror Feitelson, Yarson Hsu, Jean-Pierre Prost, Marc Snir, Sam Fineberg, Bill Nitzberg, Bernard Traversat, and Parkson Wong, *MPI-IO: a parallel file I/O interface for MPI*, Tech. Report RC 19841 (87784), IBM T.J. Watson Research Center, November 1994, Version 0.2.

[8] Peter F. Corbett, Dror G. Feitelson, Jean-Pierre Prost, George S. Almasi, Sandra Johnson Baylor, Anthony S. Bolmarcich, Yarsun Hsu, Julian Satran, Marc Snir, Robert Colao, Brian Herr, Joseph Kavaky, Thomas R. Morgan, and Anthony Zlotek, *Parallel file systems for the IBM SP computers*, IBM Systems Journal **34** (1995), no. 2, 222–248.

[9] Juan Miguel del Rosario and Alok Choudhary, *High performance I/O for parallel computers: Problems and prospects*, IEEE Computer **27** (1994), no. 3, 59–68.

[10] Sanjay Khanna and David Kotz, *A split-phase interface for parallel file systems*, Tech. Report PCS-TR97-312, Dept. of Computer Science, Dartmouth College, March 1997.

[11] Orran Krieger and Michael Stumm, *HFS: a flexible file system for large-scale multiprocessors*, Proceedings of the 1993 DAGS/PC Symposium (Hanover, NH), Dartmouth Institute for Advanced Graduate Studies, June 1993, pp. 6–14.

[12] S. Moyer and V. S. Sunderam, *Parallel I/O as a parallel application*, International Journal of Supercomputer Applications **9** (1995), no. 2, 95–107.

[13] Chance Reschke, Thomas Sterling, and Daniel Ridge, *A design study of alternative network topologies for the beowulf parallel workstation*, Proceedings, High Performance and Distributed Computing, 1996, 1996.

[14] Erich Schikuta, Thomas Fuerle, and Helmut Wanek, *Vipios: The vienna parallel input/output system*, Euro-Par'98, September 1998.

[15] Kent E. Seamons and Marianne Winslett, *Multidimensional array I/O in Panda 1.0*, Journal of Supercomputing **10** (1996), no. 2, 191–211.

[16] Thomas Sterling, Donald J. Becker, Daniel Savarese, John E. Dorband, Udaya A. Ranawake, and Charles V. Packer, *Beowulf: A parallel workstation for scientific computation*, Proceedings, International Conference on Parallel Computing, 1995, vol. 1, August 1995, pp. 11–14.

[17] Heinz Stockinger, *Dictionary on parallel i/o*, Master's thesis, University of Vienna, February 1998.

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

[19] Andrew Tridgell and David Walsh, *The HiDIOS filesystem*, Parallel Computing Workshop (England), September 1995.