Integrating Task and Data Allocation Using a Parallel File System

Hélio Crestana Guardia[1] and Liria Matsumoto Sato[2]

[1] Network and Distributed Systems Group,
Computer Science Department, Federal University of São Carlos
{helio@dc.ufscar.br}
[2] Computer Engineering and Digital Systems Department, University of São Paulo
{liria@pcs.usp.br}

Abstract—
Parallel files combine several disks to provide increased storage capacity and fast data transfer between primary and secondary memories. The use of parallel files by distributed applications however introduces new challenges to the parallel file system, which may inhibit the scalability of such applications. This work presents a study on the creation of a parallel file system that facilitates the development of distributed parallel applications with large amount of I/O operations. The proposed system was implemented and generated new tools that integrate file system operations with a mechanism for process creation and distribution. The evaluation of the system shows the viability of its utilization allowing the scalability of a distributed parallel application.

Keywords— **parallel i/o, distributed systems**

I. INTRODUCTION

Input/output (I/O) is a major bottleneck for several distributed applications that manipulate large amounts of data [OLD 98]. The process of storing and retrieving data from secondary memory greatly reduces the performance of these applications. In these cases, even the evolution in the technology used for the construction of hard disks, like the disk-arrays [KAT 89, CRO 89], brings only limited contribution to increasing the performance of the applications, as the use of a centralized file system still causes the requests to be handled serially. Distributed parallel file systems may combine the aggregated bandwidth and storage capacity of the disk-arrays with the decentralization of accesses required for parallel applications [STO 98]. The existence of independent disk controllers allows several I/O operations to take place at the same time, while still combining several disks for a single large data transfer. The I/O requirements of a parallel application however can affect its scalability and efficiency, even when a parallel file system is used. For distributed parallel applications, a major problem concerns the allocation and management of resources to minimize the response times and maximize the throughput [MAC 99, ARP 99, ROS 98, AVA 94].

Network contention problems and the wrong spatial layout of jobs, for instance, can lead to poor performance and inhibit the scalability of parallel applications.

In this paper we present a methodology for integrated task and data distribution using a parallel file system. First, we introduce the *Network Parallel File System* (NPFS) [GUA 99a, 99b], a parallel file system intended as a test bed for mechanisms to improve the performance of I/O in distributed applications. The performance of this file system is presented and is used to discuss some of the limitations of the use of a parallel file system by parallel applications. We then present an NPFS mechanism that allows the creation of parallel applications and automatic distribution of tasks into the cluster. To illustrate how the mechanism operates, a hypothetical parallel application was developed using the NPFS primitives. The results obtained show this mechanism allowed the scalability of a parallel application using a parallel file system for I/O operations.

II. NETWORK PARALLEL FILE SYSTEM

Network Parallel File System (NPFS) (**Fig. 1**) is a parallel file system aimed at a distributed computational system, namely a cluster of workstations. Reduced I/O latency and greater storage capacity are some of its goals, which are obtained by partitioning and distributing the data among several storage servers on the cluster. File fragments, called segments, are stored using each server's local file system in a client / server communication model.

Three types of processes are defined in the NPFS:

The **Master** process is responsible for starting the system, and also for the activation and deactivation of storage servers. A single instance of this process is used on the entire cluster. It is also requested in a few centralized operations, such as when opening and closing a parallel file.

The **Server** process, also known as **storage server** or **data server**, is used for storing and retrieving the fragments of parallel files. A copy of this process resides on each workstation. It can be implemented either as a single task or as a collection of cooperative processes.

Application processes developed by the users are called **Client** processes. Special NPFS service routines provide the necessary communication between the Clients and either the Master or the Server processes for the manipulation of parallel files.

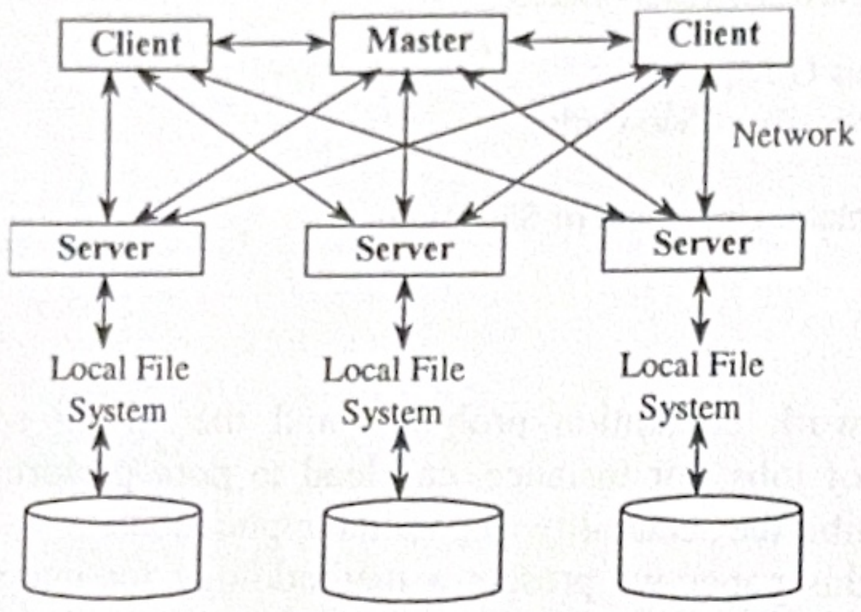

Fig. 1: NPFS architecture

Avoiding known bottlenecks and providing the independent operation of its modules were two of the main concerns in the project of the NPFS. Enhanced parallelism and the capacity to handle several requests at the same time were also considered. Good performance was sought by allowing the maximum parallelism of the hardware underneath. Fault tolerance against hardware failures, however, is not yet incorporated to the system.

Some of the optimizations adopted in the project of the NPFS protocol include the possibility to have synchronous and asynchronous operations. Asynchronous operations do not require the Clients to block until they are complete.

The NPFS protocol guarantees data consistency. Acknowledge messages and retransmissions inhibit the effects of lost requests and the faulty behavior of the NPFS modules.

The NPFS operation is initiated with the activation of the Master process. It is the Master's responsibility to initiate the execution of a Server process on each of the machines specified; servers can also be activated and stopped dynamically. Once the Master is up and running, Client processes can start using the system by requesting the opening of the desired files. Clients read from and write to an open parallel file by sending requests directly to the required storage Servers. The Master needs to be contacted again for closing a parallel file, as well as for deleting or renaming a closed file.

Once an opening request arrives at the Master, the existence of the file is checked against a local database of parallel files. In case the information of a file is found, the required Servers are prompted for opening the fragments before a response is sent to the Client. When a new parallel file is being created, Clients can choose between either specifying which Servers should be used, or using the ones that are already active. The number of fragments can also be specified. Data distribution is done in a *round-robin* fashion over the fragments.

Data manipulation is provided to Client processes through a set of primitives, which include:

p_open: opens a parallel file
p_close: closes a parallel file
p_read: reads data from a parallel file
p_write: stores data on a parallel file
p_lseek: sets the current offset position for a file
p_rename: changes the name of a file
p_unlink: removes a file

A subset of the SIO Low-level API [COR 96] is also implemented.

Most of the file manipulation routines are accomplished by sending requests from the Clients to the Master, which handles the communication with the Servers. In *p_read* and *p_write*, requests are sent from the Clients directly to the required Servers. *p_close*, *p_rename*, and *p_unlink* can be realized synchronously or asynchronously.

III. Implementation of the NPFS

A complete version of the NPFS was implemented using the C language. Inter-process communication was achieved through message passing using the *socket* related system calls. Performance optimizations made UDP as the transport level protocol of choice due to its reduced communication overheads. Acknowledgement messages and a timeout mechanism were used to provide data reliability.

The primitives for file manipulation were made available to the user programs as a set of library routines, which can be inserted into the application code (Client modules). All the defined primitives were tested in the manipulation of real data using a cluster of PCs connected via a standard *Fast-Ethernet* network. All machines were running the Linux operating system.

In order to better understand the behavior of a parallel file system, the performance of the NPFS was verified for read and write requests over consecutive file offsets. The transfer rates obtained correspond to the relation between the amount of data transferred and the total elapsed time for data access and the required message passing. Different numbers of servers were used, ranging from a single local server to several servers spread across the cluster. Request sizes ranged from 0.5 Kbytes to 1 Mbytes.

The comparison with a regular file system and with a standard remote file system (NFS) used for the same type of test helped in the analysis of the NPFS performance.

Figures **Fig. 2** and **Fig. 3** present the results obtained.

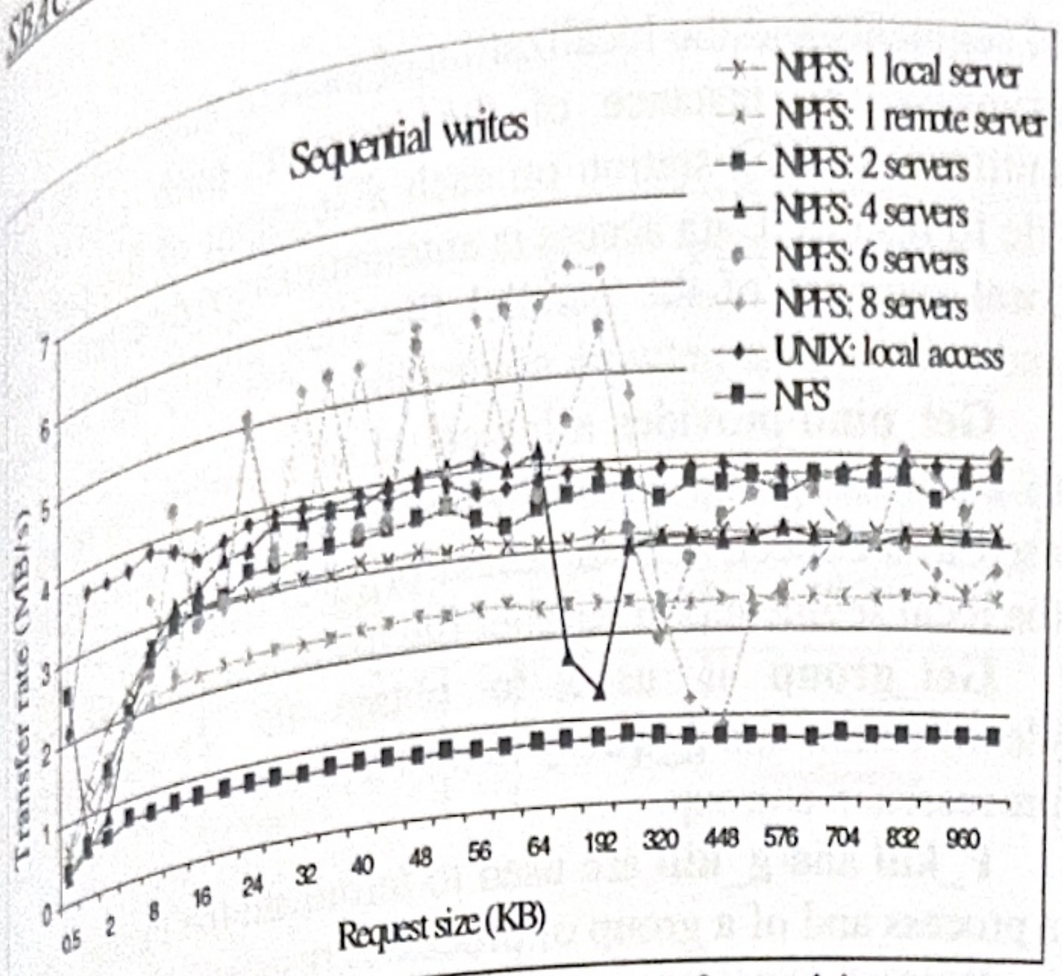

Fig. 2: Performance of the NPFS for writing

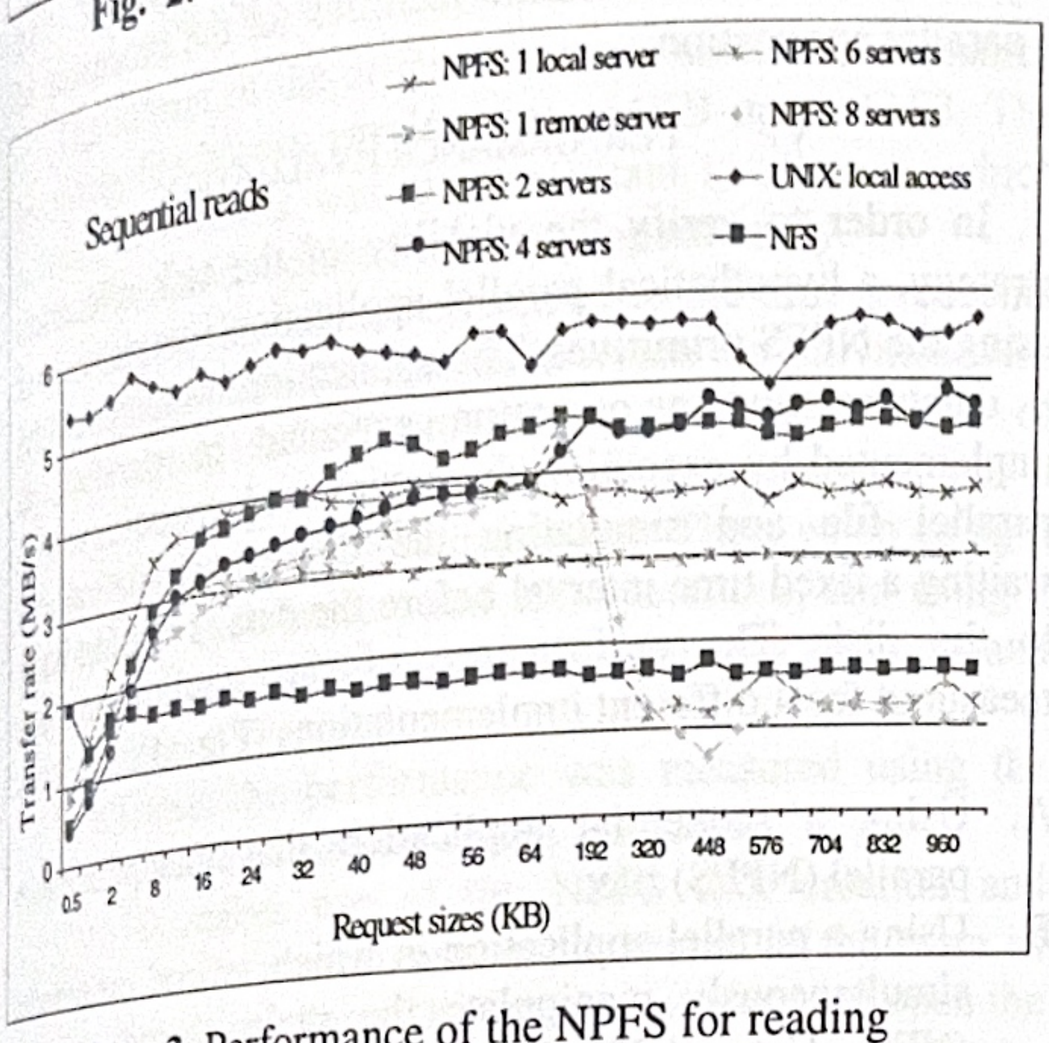

Fig. 3: Performance of the NPFS for reading

The figures above present comparative data about the behavior of NPFS, NFS and a local Unix file system. As it is intended only for providing a better understanding of the behavior of a parallel file system, no information is presented about the processors and hard disks used in the tests.

As would be expected, the performance of the parallel file system increases as more servers are used. In the write operations, the NPFS performance exceeds the results obtained with a local disc. In the read operations, however, the sequential access pattern used in the tests favored the performance of the standard local file system, which uses an aggressive prefetching policy. Different random access tests are still to be realized.

Network contention is the main bottleneck for the performance of a distributed parallel file system. It can be observed that the performance for both read and write operations drop for requests bigger than 64KB when using more than 4 storage servers. The *socket* maximum request size causes the performance of the NPFS to drop in these cases, as data has to be fragmented before transmission.

Our experience with the project and experimentation of a distributed parallel file system allows us to affirm that:

- The parallelism of the storage server is necessary to allow network and disk operations to overlap. Using this technique we were able to reduce the response time on the read and write operations. Further studies are still to be realized to determine the most appropriate number of tasks per storage servers and local disks;
- The implementation of the data servers, if not stateless, should be independent of the data distribution mechanism; this facilitates the introduction of new data distribution polices in the system;
- The determination of the file segments and data servers involved in each read and write operations should be made in the application code. In the NPFS, this is embedded in the user primitives and occurs transparently to the applications. This type of implementation provided better performance than when the Master process was used to determine the servers for each operation. Making every data Server to analyze all the requests (sent in broadcast) did not produce good performance either;
- The performance of an application with extensive use of I/O operations is a result of choosing appropriate values for the number of data segments and disks, buffer sizes, and striping units; the automatic determination of these characteristics for each application should provide significant performance gains, as does the AutoPilot project [SIM 99];
- The asynchronous operations of the storage servers, i.e. the ability to handle several requests at the same time without having to wait until a previous request is complete, produced reduced response times. On the other hand, this characteristic made the consistency protocol much more complex and unpredictable in the sense that variations in the operation of any disk or in the network response times caused unneeded retransmissions;
- Prefetching with the use of buffers and caches on the application and on the data servers have showed this mechanism to be extremely fruitful in reducing the response time of the read and write requests;
- Faster processors reduce the processing time embedded in the I/O operations; scalar distributed parallel file systems however will not be possible until the network bottleneck is eliminated.

IV. INTEGRATED TASK AND DATA ALLOCATION

Transfer rate and response times are commonly used to measure the performance of a file system; most of the NPFS project choices were made to optimize these

parameters. As a result, when a parallel file is used by a sequential application, the transfer rate and the response time should improve as more servers are used for the storage and retrieval of the data.

As we analyze the performance of the NPFS, however, we can observe the system does not scale well for more than 4 data servers. When several processes of a same application simultaneously access a shared parallel file, the network contention becomes more evident, due to new interference between the distributed file system and the inter-process communication mechanism.

The I/O properties of many parallel programs result in an execution behavior that can be partitioned into disjoint intervals, each consisting of a single burst of I/O activity followed by a single burst of computation [ROS 98]. In this programming model, data can usually be partitioned among several processes that can be executed in parallel. Inter-process communication usually occurs between each data manipulation phase.

In this way, by devising a process allocation strategy that is sensitive to parallel I/O traffic and network contention, we can tune the layout of these processes and improve the performance of parallel applications.

Thus, special NPFS primitives were developed to allow the creation and automatic distribution of parallel applications. This is achieved by exploring the data locality of tasks as they select a parallel file to work on. Using this strategy the network traffic is reduced as well as the interference between the application and the distributed parallel file system.

Besides allowing the creation and allocation of parallel application, NPFS provides primitives for inter-process communication. These primitives include:

int **spawn**(*char* *path, *char* **param, *int* n_proc, *int* fd);
int **get_pind**(*pid_t* pid);
int **get_group**(*pid_t* pid);
int ***g_query**(*int* gid, *int* *n_proc);

int **p_kill**(*pid_t* pid);
int **g_kill**(*int* gid);
int **g_free**(*int* gid);

int **p_send**(*int* dest, *char* *msg, *int* len, *int* ack);
int **p_receive**(*int* source, *char* *msg, *int* len, *int* t_out);

Spawn is used to start the execution of a parallel application. **Path** contains the name of the program to be executed, while **param** lists the command line parameters to be passed during the initialization of this program. **N_proc** contains the number of processes to be used in the application; it is usually equal to the number of segments of the parallel file to be manipulated. **Fd** is the descriptor of an open parallel file. It is used to find out about the number of segments and the localization of each of them. Using this primitive, an instance of the program **path** is usually initiated on each station on each a segment of the parallel file **fd** resides. Data access is automatically redirected to the local segment of the parallel file thus improving locality and reducing the network contention.

Get_pind provides a logical id of a process in relation to a process group created with the **spawn** primitive. It is used in a collective data access primitive or when opening the local segment of a parallel file using a segmented view.

Get_group is used to obtain the process group identification and **g_query** allows a process to know all the processes in a group.

P_kill and **g_kill** are used to terminate the execution of a process and of a group of processes respectively.

P_send and **p_receive** provide synchronous and asynchronous communication facilities for the processes of a parallel application.

V. Performance Evaluation

In order to verify the viability of the task allocation strategy, a hypothetical parallel application was developed using the NPFS primitives. The application, which could be a database update operation executed in parallel, is implemented by executing the actual read operations on a parallel file and simulating the data manipulation by waiting a fixed time interval before the data is written back to the disk. The performance of this application was measured for 4 different implementations (**Fig. 4**):

A. Using a sequential application that manipulates a parallel (NPFS) file;
B. Using a parallel application in which several processes simultaneously manipulate the same shared remote (NFS – Network File System) file;
C. Using a parallel application in which several processes simultaneously manipulate the same shared parallel (NPFS) file;
D. Using a parallel application created with the process allocation mechanism of NPFS (spawn).

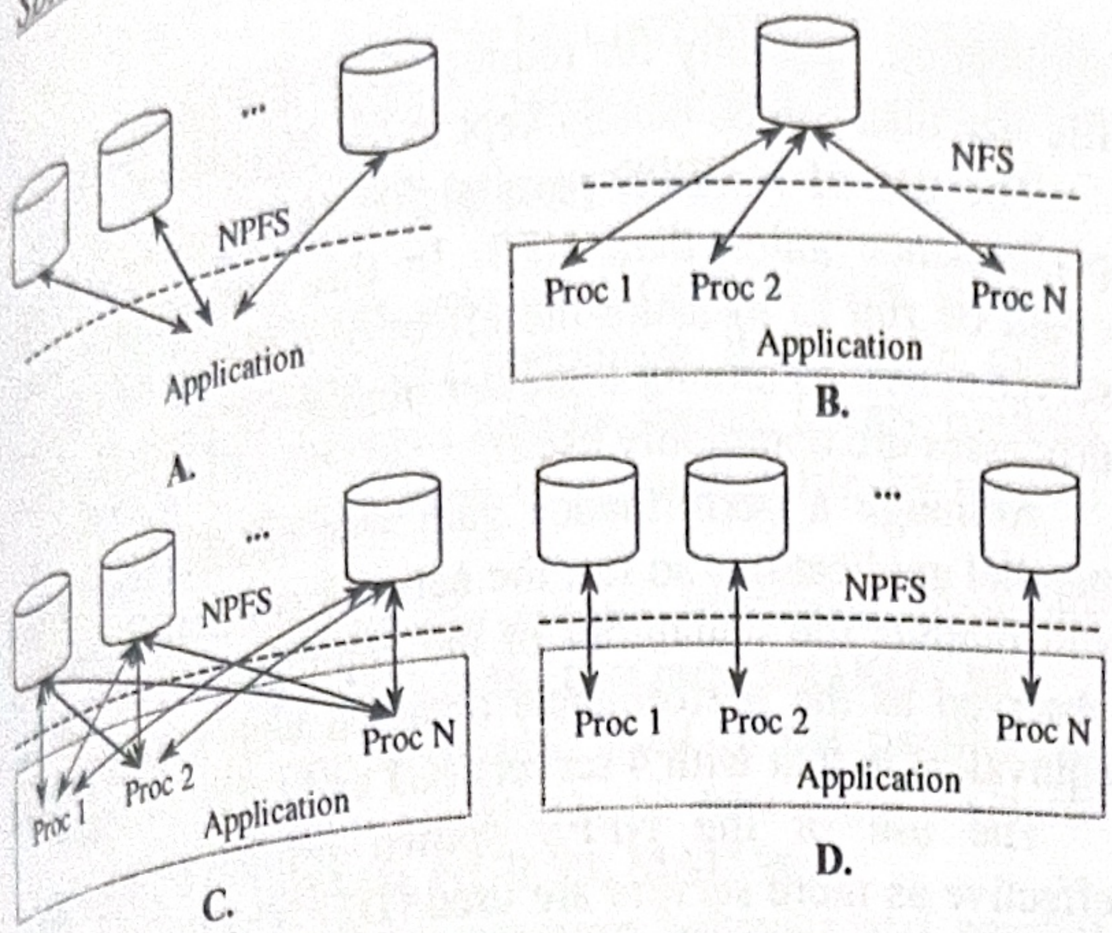

Fig. 4: Execution models used for the analysis of the NPFS task allocation mechanism

The impact of the different request sizes was measured by using file blocks varying from 64KB up to 1MB. The number of requests was made constant in all tests, which caused the amount of data to vary proportionally.

In case **A**, the results correspond to the time to execute 1000 requests. For **B**, **C**, and **D**, in which the activities were divided among several processors, the total number of requests was kept constant, but several processes ran a smaller number of requests in parallel. Thus, when using 4 data servers, each process ran 250 operations, and using 8 servers, each process was responsible for only 125 read and write requests.

In all tests the performance was measured using the elapsed time. Thus, smaller values are better.

Case **D** makes use of the NPFS task creation and allocation mechanism. It was implemented in the form of the programs that follow. The first one was used to open the parallel file and to create the application responsible for the manipulation of the data. Its source code is presented bellow:

```
// Opens a parallel file using its global view (single file)
fd=p_open(f_name,O_RDWR,0,V_GLOBAL,
        n_seg,str_size,NULL);

// Obtains the parallel file data
// int f_query(int fd, s_pfile *pfile);
status=f_query(fd, &pfile);

// Starts the execution of the parallel application
group=spawn(PROG,param,pfile.n_seg,fd))<0);

// Queries the processes in the group
proc=g_query(group,&n_proc);

// Waits until all the initiated processes are complete
for(i=0; i < n_proc; i++)
   status=p_receive(ANY,msg,MSG_SIZE,timeout);

// Frees the group information
status=g_free(group);

// Closes the parallel file
p_close(fd,1);
```

The manipulation process was implemented as follows:

```
// Obtains the logical identification in the group; it will
// determine the file segment to use
n_seg=get_pind(pid);

// Opens a parallel file in its segmented view
fd=p_open(f_name,O_CREAT|O_RDWR,0,
        V_SEGMENTED,n_seg,str_size,NULL);

for(offs=0, i=0; i < n_req; i++) {

  // reads the data from the local segment
  // p_read(int fd, char *buf, size_t count);
  n_bytes=p_read(fd,buf,req_size);

  // Manipulates the data

  // Sets the current file offset
  p_lseek(fd,offs,SEEK_SET);

  // Writes the modified data back to the disk
  n_bytes=p_write(fd,buf,req_size);

  // Updates the current file offset
  offs+=req_size;
}
// Closes the parallel file
p_close(fd,1);

// Informs the end of operation to the control process (the
// one that ran the spawn command)
p_send(dest,msg,strlen(msg),0);
```

The results obtained in cases A, B, C and D are presented in figure **Fig. 5**.

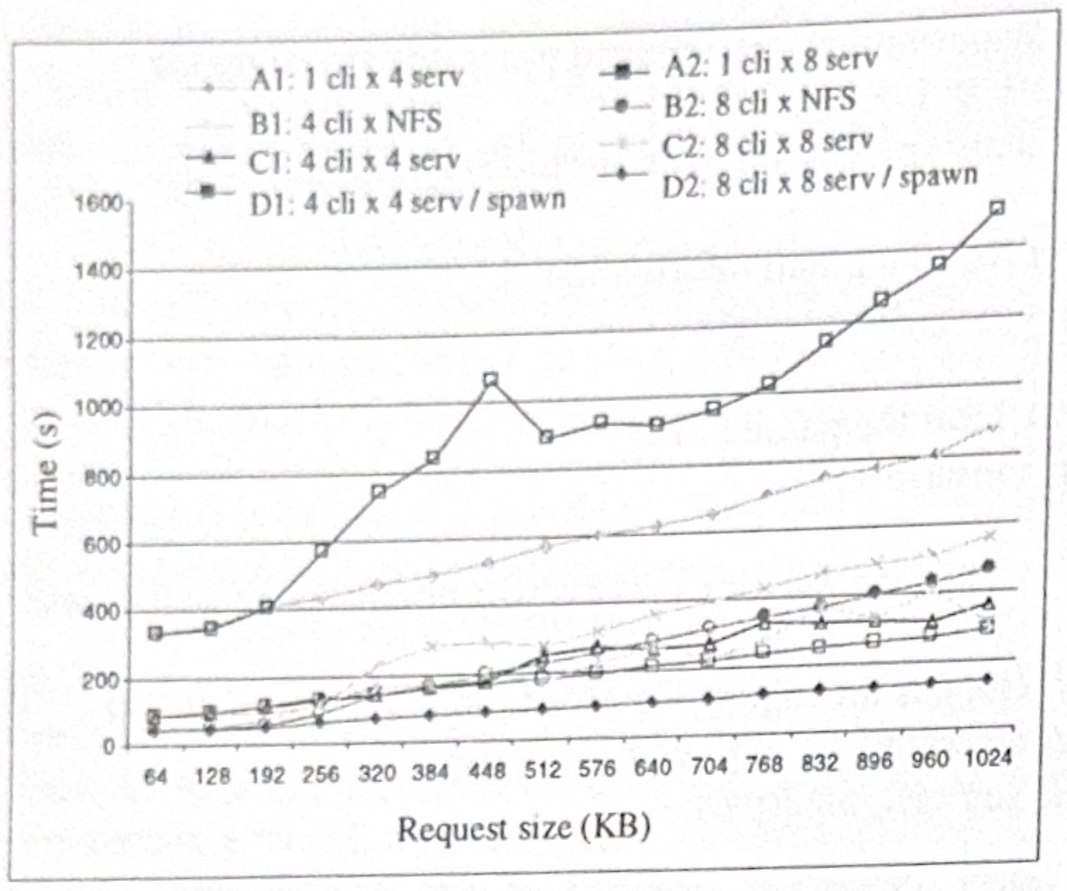

Fig. 5: Performance of the hypothetical sort application

For each request size, the same test was run for cases **A**, **C**, and **D**, using either **4** or **8** storage servers. The same tests were run for case **B** using a single NFS server, as would be the case for a standard network application with no parallel I/O support. The performance comparison between the NPFS and NFS can be found in figures **Fig. 2** and **Fig. 3**.

When 4 storage servers are used with a sequential application, 1000 operations (read, processing, and write) are realized (curve **A1**). When this application is split into 4 tasks that manipulate the same NFS file (**B1**), there is a significant performance gain, as each task runs simultaneously a smaller number of operations.

The benefits of a parallel file system can be seen in case **C1**. In this case, there is a significant performance gain compared to **B1** for most of the request sizes tested. The use of a single file in case **B1** made it possible for the NFS server to fit the entire file in RAM thus improving the performance for request sizes smaller than 256KB. Case a different file were used for each task, the influence of the Unix file buffer cache would be less effective.

The NPFS **spawn** routine for task allocation (**D1**) provided the best results for the parallel application when using 4 data servers. The network traffic reduction obtained with the locality of the data accesses improved the stability of the parallel application. The *speedup* was also increased in this case.

The performance of the sequential application using a parallel file with 8 data servers (**A2**) is worse than the one with 4 servers (**A1**). As expected, the network contention is notorious for request sizes greater than 64KB as the number of servers is increased.

Once again, there is a significant performance gain as the total number of operations is accomplished in parallel by the several tasks. The use of a shared NFS file (**B2**) also provides a performance gain compared to the sequential application, specially for request sizes that produce a total file size than can be mostly kept in the cache.

The use of a NPFS parallel file (**C2**) produces better performance gains than NFS. In this case, further tests could be run to measure the effect of dividing the storage server activities among a greater number of processes (two processes are commonly used).

Although a performance gain was obtained in all the parallel applications so far, the network contention problem still inhibits the scalability of the application. This can be observed as the performance obtained with 8 servers was equivalent to that with 4 servers (**C1** and **C2**).

The use of the NPFS **spawn** primitive was most effective as more servers are used (**D2**). The locality of the data accesses caused a significant reduction on the network traffic and provided the best performance for the application.

The speedups obtained with the several forms of parallel execution tested are presented in figure **Fig. 6**.

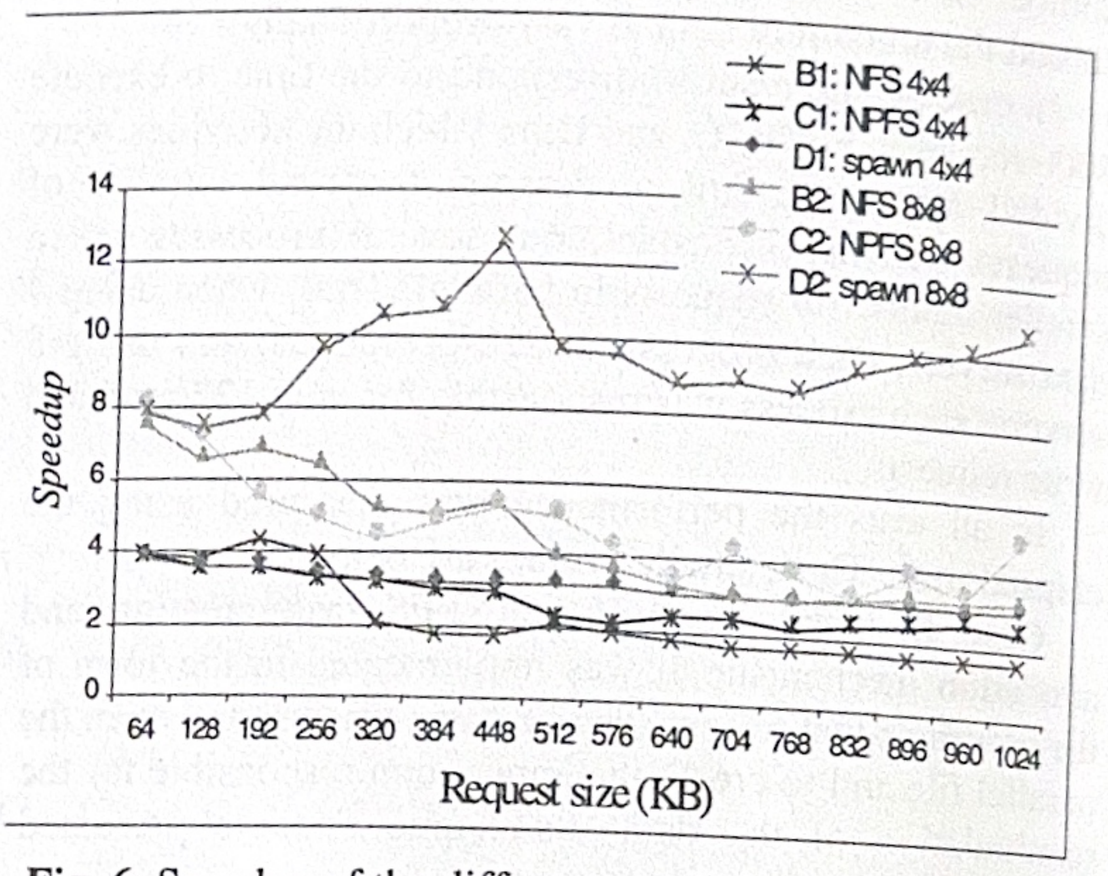

Fig. 6: Speedup of the different parallel application tested

The speedup obtained with a parallel application using a shared remote NFS file was reduced by the centralization of the accesses. This inherent serialization of the requests in this form of parallel execution reduces the performance gains (efficiency) when more processes are used (**B1** and **B2**).

When NPFS parallel files are used (**C1** and **C2**) the performance of the application is improved. However, the existence of a shared network still limits the efficiency of the application, as increasing the number of servers does not significantly improve the performance of the application.

The scalability of the application is enabled with the use of the NPFS **spawn** primitive, as can be observed with the comparison of **D1** and **D2** in figure Fig. 6. The integrated data and task allocation mechanism not only facilitates the creation of parallel applications with large amounts of I/O

transfers but also provides the best performance among the common mechanisms used for the creation of parallel tasks.

VI. Related Work

Parallel file systems provide greater storage capacity and high performance I/O operations than single disks. Some parallel I/O packages include special library routines to facilitate the manipulation of a parallel file by parallel applications.

Higher-level parallel file system interfaces allow the manipulation of data using collective requests [KOT 94].

Adaptive file striping is also used in the parallel file system to dynamically configure the best parameters for each application [SIM 99, HUB 95, MAD 96].

The impact of I/O on program behavior and parallel scheduling has also been addressed [ROS 98] for the coordinated allocation of processor and disk resources in large-scale parallel computer systems.

High-performance I/O in clusters of workstations has been addressed in the form of a mechanism for data redundancy and processing load balancing based on a centralized queuing system [ARP 99].

The impact of the spatial layout of jobs in parallel machines is studied in [MAC 99] with respect to the number of I/O and compute nodes, the network bandwidth, the layout of jobs and the read or write demands of applications.

As of our knowledge so far, no system has presented an explicit mechanism for the creation of parallel applications and integrated task and data allocation.

VII. Conclusions

This paper presented a parallel file system, called NPFS, which provides a mechanism for the creation of efficient parallel application with large amounts of I/O.

The basic NPFS primitives for file manipulation were presented as well as the performance results of the utilization of the system. The comparison of NPFS with a standard local file, NFS, and with another parallel file system [GUA 99a] shows NPFS can provide the applications with easy to use enhanced storage capacity and higher I/O transfer rates.

As could be observed, however, the NPFS does not scale well for more than a few servers, mainly due to a bottleneck found in the network contention. Distributed parallel applications tend only to aggravate this problem.

In order to improve the benefits of parallel I/O in clusters of workstations NPFS introduces a mechanism for automatic task and data allocation. The use of this mechanism enables the simplified creation of parallel application and the automatic allocation of tasks on a cluster of workstations.

In general, the mechanism developed aims to combine the benefits of a distributed parallel file system with the locality of the data accesses to improve the response time and transfer rate of parallel distributed applications.

The potential of the integrated task and data allocation mechanism was evaluated for a hypothetical application. The results obtained show this mechanism enabled the scalability of the application tested.

Even though one may think a parallel file is not helpful if local access to data is provided to each task of a parallel application, the programming ease provided for the manipulation of large amounts of data without bothering with the creation of several separate files is a strong plus.

New mechanisms for task and data allocation are also being developed for the cases in which there is not a clear segmentation of the data, which provides the complete locality of the accesses. Different levels of granularity shall also be considered.

As the number of stations in a cluster grows, the developed task allocation mechanism becomes more important. In these cases, processing load sharing can also be obtained allowing the NPFS to automatically choose the best servers to store the data. The determination of how to allocate each task of a parallel application that uses a parallel file could then be naturally obtained with the **spawn** primitive.

As a result, we expect to improve the adequacy of a cluster of workstations for the execution of parallel applications with large amounts of I/O operations.

VIII. Future Work

Processing, disk, and network heterogeneity are in the current focus of this work. This includes automatically adjusting the NPFS protocol timeout intervals to best cope with different equipment present on a cluster of workstations.

References

[ARP 99] ARPACI-DUSSEAU, Remzi H.; et al.. Cluster I/O with River: Making the fast case common. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 10-22, Atlanta, GA, May 1999. ACM Press.

[AVA 94] AVALANI, Bhavan; CHOUDHARY, Alok; FOSTER, Ian, and KIRSHNAIYER, Rakesh. Integrating task and data parallelism using parallel I/O techniques. In *Proceedings of the International Workshop on Parallel Processing*, Bangalore, India, December 1994.

[COR 96] CORBETT, Peter F. et al. *Proposal for a common parallel file system programming interface.* http://www.cs.arizona.edu/sio/api1.0.ps, September 1996. Version 1.0.

[CRO 89] CROCKETT, Thomas W. File Concepts for Parallel I/O. In *Proceedings of Supercomputing'89*, p.574-9, 1989.

[GUA 99a] GUARDIA, Hélio C. and SATO, Liria M. NPFS: Um Sistema de Arquivos Paralelos em Rede, In *Proceedings of the 17th Brazilian Symposium on Computer Network* (SBRC), 1999.

[GUA 99b] GUARDIA, Hélio C. *Considerações Sobre as Estratégias de um Sistema de Arquivos Paralelos Integrado ao Processamento Distribuído.* PhD Thesis, EPUSP 1999.

[HUB 95] HUBER, Jay et al. *PPFS: A high performance portable parallel file system.* Technical Report UIUCDCS-R-95-1903, University of Illinois at Urbana Champaign, January 1995.

[KAT 89] KATZ, Randy H.; GIBSON, Garth A. and PATTERSON, David A. Disk System Architectures for High Performance Computing. In *Proceedings of the IEEE*, v.77, n.12, p.1842-58, December 1989.

[KOT 94] KOTZ, David. *Disk-directed I/O for MIMD multiprocessors.* Technical Report PCS-TR94-226, Dept. of Computer Science, Dartmouth College, July 1994. Revised November 8, 1994.

[MAC 99] MACHE, Jens; et al.. The impact of spatial layout of jobs on parallel I/O performance. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 45-56, Atlanta, GA, May 1999. ACM Press.

[MAD 96] MADHYASTHA, Tara M., ELFORD, Christopher L. and REED, Daniel A. Optimizing input/output using adaptive file system policies. In *Proceedings of the Fifth NASA Goddard Conference on Mass Storage Systems*, pages II:493-514, September 1996.

[OLD 98] OLDFIELD, Ron and KOTZ, David. *Applications of parallel I/O.* Technical Report PCS-TR98-337, Dept. of Computer Science, Dartmouth College, August 1998.

[ROS 98] ROSTI, Emilia et al.. The impact of I/O on program behavior and parallel scheduling. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, pages 56-65. ACM Press, June 1998.

[SIM 99] SIMITCI, Huseyin and REED, Daniel A. Adaptive disk striping for parallel input/output. In *Proceedings of the Seventh NASA Goddard Conference on Mass Storage Systems*, San Diego, CA, March 1999. IEEE Computer Society Press. To appear

[STO 98] STOCKINGER, H. Classification of Parallel Input/Output Products. In *Proceedings of the PDPTA'98*, 1998.