A Split-Phase Interface for Parallel File Systems

Sanjay Khanna
David Kotz
Department of Computer Science
Dartmouth College
6211 Sudikoff Laboratory
Hanover, NH 03755-3510
{kaun,dfk}@cs.dartmouth.edu

Technical Report PCS-TR97-312

March 20, 1997

Abstract

We describe the effects of a new user-level library for the Galley Parallel File System. This library allows some pre-existing sequential programs to make use of the Galley Parallel File System with minimal modification. It permits programs to efficiently use the parallel file system because the user-level library groups accesses together. We examine the performance of our library, and we show how code needs to be modified to use the library.

1 Introduction

Many parallel applications are limited by the performance of the I/O system, and the performance of many I/O systems is currently limited by the file system. The Galley Parallel File System [NK97] has demonstrated that it can provide parallel applications with high-throughput access to their data files, if they use new file-system interfaces. Unfortunately, it is sometimes inconvenient for programmers to rewrite their application code to fit the new interface. In this paper, we describe a new user-level library that runs on top of Galley, that provides programmers with an interface similar to the traditional interface, and with performance similar to Galley's interfaces.

2 Background

Many scientific programs access large data structures (e.g., matrices) stored in files. To obtain the necessary processing and I/O speed, parallel processes run the application on many processors, and spread the data files across many disks. The Galley Parallel File System [NK97] was written both

This research was funded by NSF under grant number CCR-9404919, by NASA Ames under agreement numbers NCC 2-849 and NAG 2-936, and by Sandia National Labs contract number AS-8500

to provide a parallel file system that programmers may use, and to provide programmers with the ability to choose how their files should be distributed across the disks. Nieuwejaar studied common workloads [NKP+96], and discovered that files were often accessed in a *strided* pattern. Strided patterns occur when file accesses (reads or writes) are of a fixed size, and successive accesses are separated by a fixed number of bytes. As a result, Galley provides an interface to read and write files in strided patterns. Unfortunately, it is not always easy to convert legacy applications to use the new interface, because the programmer must rewrite loops to build a Galley strided-access specification.

Our user-level library is built on top of the Galley Parallel File System [NK97], which is described in the next section. We attempt to enable programmers to adapt existing programs to the Galley Parallel File System, resulting in a programming style similar to that of Split-C [CDG+93]. In Split-C, programs communicate data between processors using split-phase *get* and *put* operations. The process makes a series of asynchronous *get* and *put* requests, then blocks waiting for all gets and puts to complete.

Our work may also be compared to the Vesta Parallel File System [CF96]. Vesta allows users to distribute files on multiple I/O nodes, similar to Galley. Galley requires the user to define the number of *subfiles* (one per I/O node) used for a file at the time of its creation. Vesta requires the user to define the *basic striping unit* (BSU) and the number of *cells* (sequential streams) of the file at the time of its creation. The BSU is the smallest unit (in bytes) of a Vesta file that can be accessed. One difference is that Vesta provides logical mappings to view BSUs striped across the cells in a variety of ways, whereas Galley does not provide any such views to the user. (A user-level library is required to provide anything other than the raw view of a Galley file; a Vesta interface library exists, for example.) When a Vesta file is opened, it must be opened in a particular view, which defines a two-dimensional stripe across the *cells* of the file. By defining the same view and selecting different stripes, a multi-process program is able to ensure that no two parts are accessing the same bytes of the file.

2.1 Galley Parallel File System

The Galley Parallel File System [NK97] is a parallel file system enabling processes to read and write files that are distributed across several disks. The method of distribution of the files is left to the user or to a user-level library on top of Galley. The files are stored on disks connected to I/O processors (IOPs), and the user programs run on compute processors (CPs). Each disk is connected to a separate IOP. A file is split into *subfiles*, each of which must reside on a single disk, and no more than one subfile for a particular file may reside on the same disk. The number of subfiles is determined when the file is created. Each subfile is split into named *forks*, which may be created on an ad-hoc basis. Multiple forks with the same name may exist in multiple subfiles of the same file. Each fork in a particular subfile is a sequential stream of data, and may be accessed similar to a Unix file. Galley provides a few different primitives for accessing the forks, and these are described in detail in the Galley paper [NK97]. Only the *gfs_listio*() primitive is used in the GFS-GROUP library and elsewhere throughout this paper. The *gfs_listio*() function allows the program to request a list of read or write transfers to a single fork, in one request.

3 GFS-GROUP library

GFS-GROUP is a user-level library to aid the conversion of existing sequential *C* programs using regular Unix-like I/O to use the Galley Parallel File System [NK97] with minor modifications. The library that we provide accumulates the user's requests and then submits them asynchronously to Galley as a collection of requests, thereby reducing the overhead for each request. It keeps track of handles for the requests. Because the requests are sent asynchronously, the user must ensure that the request is complete before using the information from a read or reusing the buffer for a write. Of course, if the original program was not written in this manner, then some significant additions to the code may be necessary to implement a buffer. Many programs are already written to use a large buffer, and are simply using numerous small I/O requests to access non-contiguous file data. The location of the reads or writes need not be changed, and whenever a previously read value is used or a write buffer needs to be overwritten, *gfs_group_waitio*() must be called to ensure that the buffer is ready to be used. The main purpose of the library is to group the user's requests together before submitting them to Galley, thereby reducing the total number of Galley requests. Since each Galley request becomes a separate message to the IOP, they can become quite expensive.

Because the GFS-GROUP library is a user-level library, and is meant to be portable, it cannot gain control of the scheduler. Therefore, it is able to submit requests to the Galley Parallel File system only when it has been called by a user program. The descriptions of the library functions below also describe when the library submits a request to Galley.

3.1 The GFS-GROUP library functions

There are five function calls provided by the GFS-GROUP library, namely

int *gfs_group_read*(**int** kid, **int** offset, **char** *buf, **int** size)
int *gfs_group_write*(**int** kid, **int** offset, **char** *buf, **int** size)
void *gfs_group_doneio*()
int *gfs_group_testio*() and
void *gfs_group_waitio*()

These functions are described in the following sections.

3.1.1 int *gfs_group_read*(int kid, int offset, char *buf, int size)

This function submits a read request to the GFS-GROUP library. The arguments are as below:

int *kid* is the id of the fork to read from or write to. Similar to a file descriptor for regular Unix files, a fork must be opened before it can be read or written.

int *offset* is the offset, from the beginning of the fork, where data should be read. Please note that there are no seeks or accesses relative to the current position, because there is no notion of "current position" or "file pointer."

char **buf* is the buffer that will receive what is read from disk.

int *size* is the number of bytes to read.

The return value is 0 if there are no errors, and -1 if there was an error. An error code is stored in *gfs_errno* if there was an error. If this request is to access a fork different from the last fork accessed, all the previous requests that have not yet been submitted to Galley are now submitted to Galley. This effect results from our implementation, which gathers requests into lists for *gfs_listio*(), and a *gfs_listio*() request cannot access two different forks. This function submits all the previous requests in addition to the current request to Galley if the number of requests not submitted is greater than a threshold (currently 1024), or the total size of all requests not submitted is greater than a size threshold (currently 16MB). In addition, depending on a compile-time option, if it is not waiting for Galley to complete any previous requests, it will also submit all the previous requests including the current request to Galley (the intent is to keep Galley busy).

3.1.2 int *gfs_group_write*(int kid, int offset, char *buf, int size)

This function submits a write request to the GFS-GROUP library. The arguments are the same as for a read request. Please note that once a read (write) request has been submitted, no write (read) requests can be submitted until *gfs_group_doneio*() has been called.

3.1.3 void *gfs_group_doneio*()

This function tells the GFS-GROUP library that one group of I/O requests has been completed, forcing any remaining I/O requests to be submitted to Galley. Please note that reads and writes may not be combined in the same group.

3.1.4 int *gfs_group_testio*()

This function checks whether all the I/O submitted to the GFS-GROUP library so far has been completed. It returns TRUE if it has, FALSE if it has not. This function also submits a request to Galley if it is not waiting for Galley to complete previous requests. Thus, if the GFS-GROUP library had submitted a large request to Galley, and then the user submitted a few small requests to Galley and then called *gfs_group_testio*(), this function will not submit the remaining requests to Galley if it is still waiting for Galley to complete the large request.

3.1.5 void *gfs_group_waitio*()

This function will wait for all outstanding I/O to complete, after submitting any unsubmitted requests to Galley. Please note that I/O may be completed in any order. If one group with writes was followed by *gfs_group_doneio*(), and then one group with reads, the writes may not have been completed when the reads were performed unless *gfs_group_waitio*() was also called between the two groups.

3.2 Using the GFS-GROUP library

The GFS-GROUP library uses the *gfs_listio* function call provided by the Galley Parallel File System [NK97]. This restricts each set of requests submitted to the Galley Parallel File System to access the same fork in the same subfile, and also each set of requests must be either all reads or all writes. Figure 1 is an example sequential program that accesses a regular Unix file. Figure 2 is the program in Figure 1 converted to use the GFS-GROUP library. Figure 3 is the same program converted to use Galley without the GFS-GROUP library.

4 Experiments and Results

We ran several experiments to evaluate the benefit of using the GFS-GROUP library. We compared the times to read from and write to disk. All the programs used four IOPs and one CP. All the machines were IBM RS6000s running AIX 4.1.3. The processors communicated via a 100 Mbps FDDI network. Each program created one fork on each IOP, and wrote a matrix striped across the IOPs by writing the first column to the first subfile, the next column to the next subfile, and so on. The writes were verified by another program to ensure that they were correct. The time recorded includes only the time taken to read or write the files. The timer was stopped after flushing the Galley disk caches, but before closing the files. Each program wrote a large matrix to the file, and then another program read it back. The synchronous program used a buffer large enough to store

```
main()
{
  int i, j;
  int matrix[10][10];
  FILE *f;
  f = fopen("myfil", "w");

  for (i=0; i < 10; i++) {
     for (j=0; j < 10; j++) {
        matrix[i][j] = i * 100 + j;
     }
     fwrite(matrix[i], sizeof(int), 10, f);
  }

  fclose(f);
}
```

Figure 1: Example Sequential Program

```
main()
{
  int i, j;
  int matrix[10][10];
  int fid, kid;

  gfs_init(NULL); /* initialize Galley */
  fid = gfs_open_file("myfil");
  kid = gfs_open_fork(fid, 1, "matrix");

  for (i=0; i < 10; i++) {
     for (j=0; j < 10; j++) {
        matrix[i][j] = i * 100 + j;
     }

        /* write row i of matrix */
     gfs_group_write(kid, i*10*sizeof(int), matrix[i], 10*sizeof(int));
  }

  gfs_group_doneio();
  gfs_close_fork(kid);
  gfs_close_file(fid);
}
```

Figure 2: Example Sequential Program converted to use GFS-GROUP

```
main()
{
  int i, j;
  int matrix[10][10];
  gfs_handle my_handles[10];
  int fid, kid;

  gfs_init(NULL); /* initialize Galley */
  fid = gfs_open_file("myfil");
  kid = gfs_open_fork(fid, 1, "matrix");

  for (i=0; i < 10; i++) {
     for (j=0; j < 10; j++) {
        matrix[i][j] = i * 100 + j;
     }

        /* write row i of matrix */
     my_handles[i] = gfs_new_handle();
     gfs_nb_write(my_handles[i], kid, i*10*sizeof(int), matrix[i], 10*sizeof(int));
  }

  for (i=0; i < 10; i++) {
     gfs_wait(my_handles[i]); /* wait for I/O to complete */
  }

  gfs_close_fork(kid);
  gfs_close_file(fid);
}
```

Figure 3: Example Sequential Program converted to use Galley without GFS-GROUP

one column. The other programs used a buffer large enough to store four columns because we used four IOPs. Each program was run with two sets of arguments to compare the effectiveness of using the GFS-GROUP library when writing small records as well as when writing large records. For the first set of experiments, each program read or wrote a 4096x4096 matrix where each entry was 16 bytes. For the second set of experiments, each program read or wrote a 128x128 matrix where each entry was 16384 bytes. Thus, all the matrices were 256 MB in size, or 64 MB per IOP. Each program was run ten times with each set of arguments; we report the mean execution times.

4.1 Programs used to evaluate GFS-GROUP

We used four programs to evaluate the effectiveness of GFS-GROUP.

4.1.1 group-send

This program used the GFS-GROUP library. The program submitted one request to the GFS-GROUP library for each matrix entry, and after every four columns it waited for the requests to complete so that the buffer could be filled with data for the next four columns. The GFS-GROUP library submitted a request to Galley whenever it was called, *if* it was not waiting for a previous request to complete. It also submitted a request whenever it received data for a new column because it could not submit data for different columns (forks) in the same *gfs_listio*() request.

4.1.2 group-nosend

This was the same program as **group-send** except that the GFS-GROUP library was compiled differently. This made the GFS-GROUP library wait until a new column (fork) was used before submitting a request to Galley.

4.1.3 asynchronous

This program was written to measure the overhead of the GFS-GROUP library. It submits requests similar to the **group-nosend** program, except that it does not use the GFS-GROUP library. Rather, the overhead necessary for manipulating asynchronous reads and writes is included in the same program. Any additional time required by **group-nosend** was thus the overhead of the GFS-GROUP library.

4.1.4 synchronous

This program submits synchronous *gfs_listio*() requests to the Galley Parallel File System. Each request contains one column, and there is no overlap of writing to the separate disks because it

Write Timings (256 MB)

	4096x4096 matrix		128x128 matrix	
Program	mean	std dev	mean	std dev
synchronous	148.9	3.79	103.4	6.14
asynchronous	139.5	3.36	66.9	7.97
group-nosend	144.2	2.47	65.3	9.19
group-send	171.3	4.39	63.0	6.98

Read Timings (256 MB)

	4096x4096 matrix		128x128 matrix	
Program	mean	std dev	mean	std dev
synchronous	204.7	3.71	126.5	20.73
asynchronous	175.2	4.32	70.3	6.88
group-nosend	192.9	8.31	78.5	9.71
group-send	206.2	7.26	77.2	4.38

Table 1: Timings of Experiments. Each IOP has a file system using 16K striping across two 1 GB disks.

waits for each request to complete before submitting the next one.

The timings of the experiments are shown in Table 1. We used an unpaired-observations t-test to decide whether the differences are significant at the 95% confidence level. We show the results in Table 2. The approximate speedups of the programs are given in Table 3. We regard programs to have the same speed if there was no significant difference between the timings of the programs.

Looking at the write timings for the large (4096x4096) matrix with small elements, we see that the **group-nosend** and **synchronous** programs perform about the same, the **asynchronous** program is faster, and the **group-send** program is slower. Clearly **asynchronous** is faster than **synchronous** because it can overlap I/O time on all four IOPs, and overlap I/O with computing the next column to be written. The overhead of the **GFS-GROUP** library makes **group-nosend** about the same time as the **synchronous** program. Allowing **GFS-GROUP** to submit requests whenever it is not waiting for Galley (**group-send**) was not worthwile because the requests were so small that it was always worth waiting so multiple requests could be grouped.

Reading was slower because reads cannot complete until the physical I/O is complete, while writes can complete once the data reaches Galley's IOP cache (we include the time to flush the cache at the end but meanwhile there is some extra concurrency available at the CPs). As a result, the **synchronous** program slows to about the same as **group-send**.

On the other hand, from the write timings for the small (128x128) matrix with large elements, we see that all but the **synchronous** programs were equivalent and much faster than the **synchronous** program, because **synchronous** had no overlap between I/O on one IOP and another.

Write Results (256 MB each)

	4096x4096 matrix				128x128 matrix			
Program	synch	asynch	nosend	send	synch	asynch	nosend	send
synchronous	—	yes	yes	yes	—	yes	yes	yes
asynchronous	yes	—	yes	yes	yes	—	no	no
group-nosend	yes	yes	—	yes	yes	no	—	no
group-send	yes	yes	yes	—	yes	no	no	—

Read Results (256 MB each)

	4096x4096 matrix				128x128 matrix			
Program	synch	asynch	nosend	send	synch	asynch	nosend	send
synchronous	—	yes	yes	no	—	yes	yes	yes
asynchronous	yes	—	yes	yes	yes	—	yes	yes
group-nosend	yes	yes	—	yes	yes	yes	—	no
group-send	no	yes	yes	—	yes	yes	no	—

Table 2: Results of 95% significance t-tests: "yes" means that the different performance was significantly different at the 95% confidence level. Each IOP has a file system using 16K striping across two 1 GB disks.

When reading the small (128x128) matrix with large elements, the **asynchronous** program was fastest, followed by the two (equivalently fast) group programs, and **synchronous** was again slowest. The two group programs were slower than **asynchronous** due to library overhead, which appears to be about 9-10%. **Group-send** was not slower than **group-nosend** because there was rarely an opportunity to submit a request when Galley was not busy. It was not faster probably because there was little delay in our programs between the GFS-GROUP calls, so Galley was rarely idle.

5 Conclusions

The GFS-GROUP library provides an easier way to convert some sequential I/O loops into parallel than to use pure Galley routines. It appears to have acceptable overhead for large requests, but somewhat disappointing overhead for tiny requests. We found it unhelpful to use an aggressive (**group-send**) approach, at least in our experiments.

6 Future Work

When using the GFS-GROUP library, it would be nice to be able to select at runtime whether to submit requests to Galley only if there are too many requests on hand (as in **group-nosend**) or to submit them immediately if the library is not waiting for Galley to complete a previous request

Write Results (256 MB each)

	4096x4096 matrix				128x128 matrix			
Base Program →	synch	asynch	nosend	send	synch	asynch	nosend	send
synchronous	*1.00*	0.94	0.97	1.15	*1.00*	0.65	0.63	0.61
asynchronous	1.07	*1.00*	1.03	1.23	1.55	*1.00*	*0.98*	*0.94*
group-nosend	1.03	0.97	*1.00*	1.19	1.58	*1.02*	*1.00*	*0.96*
group-send	0.87	0.81	0.84	*1.00*	1.64	*1.06*	*1.04*	*1.00*

Read Results (256 MB each)

	4096x4096 matrix				128x128 matrix			
Base Program →	synch	asynch	nosend	send	synch	asynch	nosend	send
synchronous	*1.00*	0.86	0.94	*1.01*	*1.00*	0.56	0.62	0.61
asynchronous	1.17	*1.00*	1.10	1.18	1.80	*1.00*	1.12	1.10
group-nosend	1.06	0.91	*1.00*	1.07	1.61	0.90	*1.00*	*0.98*
group-send	*0.99*	0.85	0.94	*1.00*	1.64	0.91	*1.02*	*1.00*

Table 3: Approximate Speedups of programs. If the number is in *italics*, then there is no significant difference between the base program and the program being compared.

(as in **group-send**). Currently, the library must be compiled with the appropriate option.

We could also perform more experiments to be able to better analyze the effects of using the GFS-GROUP library on different kinds of file system accesses. The current experiments have focused only on simple file reading and writing, but other possibilities include writing data to existing files, and accessing files in a non-sequential order.

References

[CDG+93] David E. Culler, Andrea Drusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick. Parallel programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–283, Portland, OR, 1993. IEEE Computer Society Press.

[CF96] Peter F. Corbett and Dror G. Feitelson. The Vesta parallel file system. *ACM Transactions on Computer Systems*, 14(3):225–264, August 1996.

[NK97] Nils Nieuwejaar and David Kotz. The Galley parallel file system. *Parallel Computing*, 23(4), 1997. To appear.

[NKP+96] Nils Nieuwejaar, David Kotz, Apratim Purakayastha, Carla Schlatter Ellis, and Michael Best. File-access characteristics of parallel scientific workloads. *IEEE Transactions on Parallel and Distributed Systems*, 7(10):1075–1089, October 1996.