SFIO, Système de fichiers distribués pour MPI-I/O

Emin Gabrielyan

EPFL, Département d'informatique
Laboratoire de Systèmes Périphériques
Emin.Gabrielyan@epfl.ch

Résumé

Cet article présente l'architecture d'un système de fichiers distribué (SFIO) pour la gestion des entrées/sorties parallèles dans un environment MPI. Différentes techniques d'optimisation des communications et d'accès aux disques sont présentées. A l'aide de types dérivés MPI, on peut transmettre sur le réseau des données fragmentées à écrire sur disque à l'aide d'une seule commande MPI. Nous présentons les performances d'entrée/sorties du système de fichiers distribué sur le superordinateur Swiss-Tx formé de noeuds de calcul et E/S de type DEC Alpha.

SFIO, Parallel File Striping for MPI-I/O

Emin Gabrielyan

EPFL, Computer Science Dept.
Peripheral Systems Lab.
Emin.Gabrielyan@epfl.ch

Abstract

This paper presents the design and evaluation of a Striped File I/O (SFIO) library for parallel I/O in an MPI environment. We present techniques for optimizing communications and disk accesses for small striping factors. Using MPI derived datatype capabilities, we transmit fragmented data over the network by single MPI transfers. We present first results regarding the I/O performance of the SFIO library on DEC Alpha clusters, both for the Fast Ethernet and for the TNet Communication networks.

1. Motivation/Introduction

For I/O bound parallel applications, parallel file striping is an alternative to Storage Area Networks (SAN). In particular, parallel file striping offers high throughput I/O capabilities at a much cheaper price, since it does not require a special network for accessing the mass storage sub-system [6].

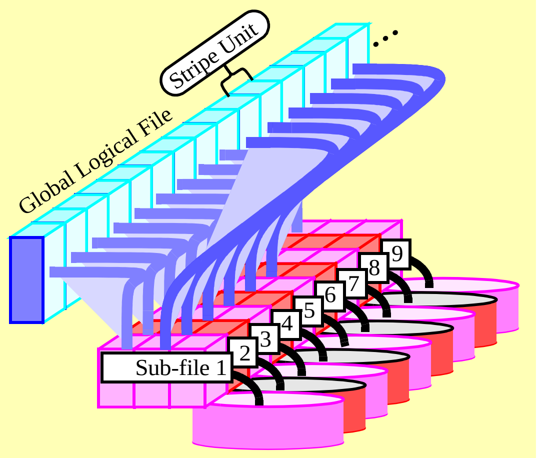

Fig.1. File striping

Important aspects of parallel I/O systems are highly concurrent access capabilities to the common datafiles by all parallel application processes and linear increase in performance when increasing the number of I/O nodes and processors. Parallelism for input/output operations can be achieved by striping the data accross multiple disks so that read and write operations occur in parallel (see Fig. 1). A number of parallel file systems where designed ([1], [2], [3], [5]), which make use of the parallel file striping paradigm.

MPI is currently the most used standard framework for creating parallel applications running on various types of parallel computers. A well known implementation of MPI [9], called MPICH, has been developed by Argone National Laboratory. MPICH is used on different platforms and incorporates MPI-1.2 operations [10] as well as the MPI-I/O subset of MPI-II ([11], [12], [13]). MPICH is most popular for cluster architecture supercomputers, based on Fast or Gigabit Ethernet networks. MPICH's MPI-I/O underlying I/O implementation is completely sequential and is based on NFS ([4], [14]).

Due to the locking mechanisms needed to avoid simultaneous multiple accesses to the shared NFS file, MPICH MPI-I/O write operations can be carried out only at a very slow throughput[1].

Other factor reducing peak performance is the read-modify-write operations useful for writing fragmented data to the target file. Read-modify-write requires sending the full data covering the written data fragment over the network, modifying it and transmitting it back. In the case of high data fragmentation, i.e. small chunks of data spread over a large dataspace in the file, network access overhead may become dominant.

To be able to provide the highest level of parallelization of access requests as well as a good load balance, small striping units are required. However low stripe unit size increases the communication and disk access cost. Our SFIO parallel file striping implementation integrates the relevant optimizations by merging sets of network messages and disk accesses into single messages and single disk access requests. The merging operation makes use of MPI derived datatypes.

The SFIO library interface does not provide nonblocking operations, but internally, accesses to the network and disks are made asynchronously.

Section 2 presents the overall architecture of the SFIO implementation as well as the software layers in order to provide an MPI-I/O interface on top of SFIO. The SFIO interface description, small examples as well as the details of the system design, caching techniques and other optimisations are presented in Section 3. First performance

[1] When 7 compute nodes access one shared NFS file in an interleaved maner, write throughput performance on MPICH MPI-IO is 35 KB/s per node.

results are given for various configurations of the Swiss-Tx supercomputer [7]. Section 5 presents the conclusions and future work.

2. Global Architecture

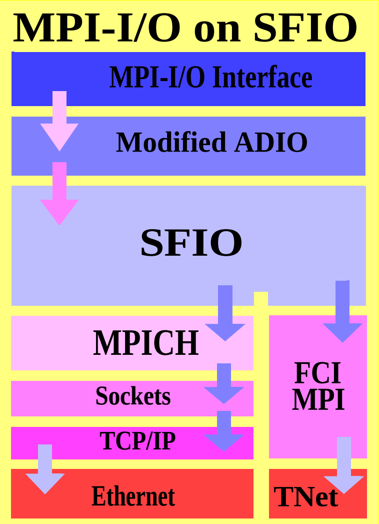

Fig. 2. Integration of SFIO

The SFIO library is implemented using MPI-1.2 message passing calls. It is therefore as portable as MPI-1.2. The local disk access calls, which depend on the underlying operating system are non-portable. However, they are separately integrated into the source for Unix and Windows-NT versions.

The SFIO parallel file striping library offers a simple Unix like interface. We also intend to provide an MPI-I/O interface on top of SFIO. The intermediate level of MPICH's MPI-I/O implementation is ADIO [14]. We successfully modified the ADIO layer of MPICH to route calls to the SFIO interface.

On the Swiss-T1 machine, SFIO can run on top of MPICH as well as on top of FCI-MPI using the low latency and high throughput network Tnet [8].

3. Unix like interface for parallel striped file I/O

Interface

Two functions, *mopen* and *mclose* are provided to open and close a striped file. Note that a file should be opened by all compute nodes irrespectively of whether that node uses the file or not. This restriction is placed in order to ensure correct behavior of future collective parallel I/O functions. Additionally, the operation of opening as well as of closing a file, implies a global synchronization point in the program. The generic functions to read and write to a file are respectively *mreadc* and *mwritec*.

The multiple I/O request specification interface allows an application program to specify multiple I/O requests within one call. This permits optimizations which otherwise would not be possible. The multiple I/O request operations are *mreadb* and *mwriteb*.

The following source gives a simple SFIO example. The striped file with a stripe unit size of 5 bytes consists of two sub-files. A single compute node accesses the striped file. It is assumed that the program is launched with one compute node MPI process.

```
#include <mpi.h>
#include "mio.h"
int _main(int argc, char *argv[])
{
        MFILE *f;
        f=mopen
        (
                "t0-p1,/tmp/a1.dat;"
                "t0-p2,/tmp/a2.dat;"
                ,5
        );
        mwritec(f,0,"Hello World",11);
        mclose(f);
}
```

Below is an example of multiple compute nodes accessing a striped file. Again the striped file with a stripe unit size of 5 bytes consists of two subfiles. It is accessed by three compute nodes. Each of them writes at different positions simultaneously.

```
#include <mpi.h>
#include "../mpi/sfio/mio.h"
int _main(int argc, char *argv[])
{
        MFILE *f;
        f=mopen
        (
                "t0-p1,/tmp/a1.dat;"
                "t0-p2,/tmp/a2.dat;"
                ,5
        );
        if(rank()==0)
        {
                mwritec(f,0,"Hello*World,*",13);
        }
        else if(rank()==1)
        {
                mwritec(f,13,"I*am*a*program*",15);
        }
        else if(rank()==2)
        {
                mwritec(f,28,"written*with*SFIO.",18);
        }
        mclose(f);
}
```

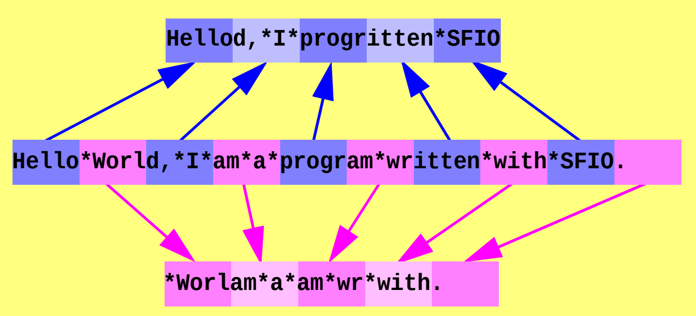
Hellod,*I*progritten*SFIO
Hello*World,*I*am*a*program*written*with*SFIO.
*Worlam*a*am*wr*with.

Fig. 3. Distribution of striped file accross sub-files

We assume that the program is launched with three compute and two I/O MPI processes. The end the global file contains the text combined from the fragments written by the first, second and third compute nodes, i. e. "**Hello*World,*I*am*a*program*written*with*SFIO.**" The text is distributed accross the two sub-files. The first sub-file contains "**Hellod,*I*progritten*SFIO**" and the second "***Worlam*a*am*wr*with.**" (Fig. 3)

Function Calls

In this sub-section we present the SFIO library application programmer interface.

File management operations are *mopen*, *mclose*, *mchsize*, *mdelete* and *mcreate.*

```
MFILE* mopen(char *name, int chunk);
void mclose(MFILE *f);
void mchsize(MFILE *f, long size);
void mdelete(char *name);
void mcreate(char *name);
```

All the presented file management operations are collective. Operation *mopen* returns to the compute node a pointer to the logical striped file descriptor. The striped file name, required for the *mopen*, *mdelete*, *mcreate* commands is a string containing the full specification of the number, sequence, locations and paths of sub-files representing the global striped file. The format of the name is a sequence of sub-files, spearated by “;”: “`<host>,<path>;<host>,<path>;<host>,<path>`...”. For example “`t0-p1,/tmp/a1.dat;t0-p2,/tmp/a2.dat;`”

There are single block and multi-block data access requests.

```
void mread(MFILE *f, long offset, char *buffer, unsigned size);
void mwrite(MFILE *f, long offset, char *buffer, unsigned size);
void mreadc(MFILE *f, long offset, char *buffer, unsigned size);
void mwritec(MFILE *f, long offset, char *buffer, unsigned size);
void mreadb(MFILE *f, unsigned blknum,
     long offsets[], char *buffers[], unsigned sizes[]);
void mwriteb(MFILE *f, unsigned blknum,
     long offsets[], char *buffers[], unsigned sizes[]);
```

The data access requests are blocking and non-collective. *mreadc* and *mwritec* functions are the optimized versions of the *mread* and *mwrite* functions.

Error management functions are given by *merror* and its collective counterprt *merrora*.

```
void merrora(unsigned long *ioerr);
void merror(unsigned long *ioerr);
void prioerrora();
```

merror and *merrora* return an array of error statistic accumulated on all the I/O nodes. At the same time, they reset the error counters on all the I/O nodes. Statistics are accumulated for operating system I/O calls and listed according to open, close, creat, unlink, ftruncate, lseek, write and read functions. *prioerrora* is a collective operation which prints the error statistic to the standart output of the application.

Implementation details

In our programming model, we assume a set of compute nodes and an I/O subsystem. The I/O subsystem is represented as set of I/O nodes running I/O listener processes. Both compute nodes and I/O listeners are MPI processes within a single MPI program. This allows the I/O subsystem to optimize the data transfers between compute nodes and I/O nodes using MPI derived datatypes. The user is allowed to directly use MPI operations only across the compute nodes for computation purposes. The I/O nodes are available to the user only through the SFIO interface.

When a compute node invokes an I/O operation, the SFIO library takes control of that compute node. The library routes the requests to the corresponding I/O listener proxy on the compute node, caches the routed requests and does an optimisation of requests queued for each I/O node in order to minimize the cost of disk accesses and network communications. After actual transmission of the messages, the I/O listener(s) prepares a reply which is sent back to the compute node.

Optimisation

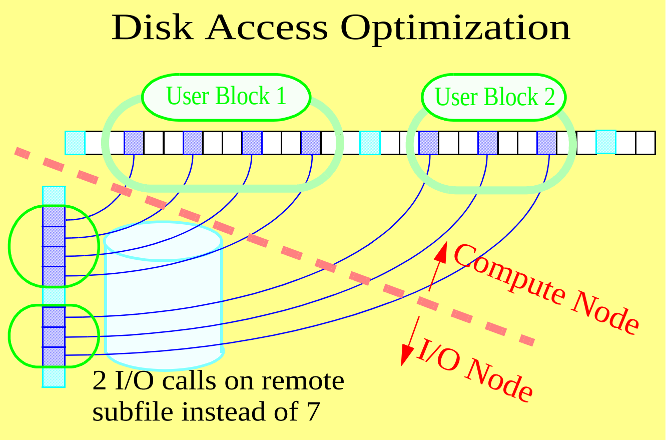

Fig. 4. Disk Optimisation

In order to optimize the disk accesses on the remote I/O node, the algorithm implemented on the compute node tries to combine all overlapping or consecutive I/O requests collected in the cache (Fig. 4). Requests queued for each I/O node are sorted according of their offsets on the remote disk subfile.

Queued I/O node access requests cached on the compute node are launched either at the end of the function call or when the buffer size reserved on the remote I/O listener for data reception may become full. Memory is not a problem on the compute node, since data always stays in user memory and is not buffered. When launching I/O requests, the SFIO library performs a single data transmission to each of the I/O nodes. It creates dynamically a derived datatype which points to the set of pieces in user space memory related to the given I/O node and transmits the data in a single stream without additional copy. The I/O listener at the same time receives the data as a contiguous block.

4. Performance results

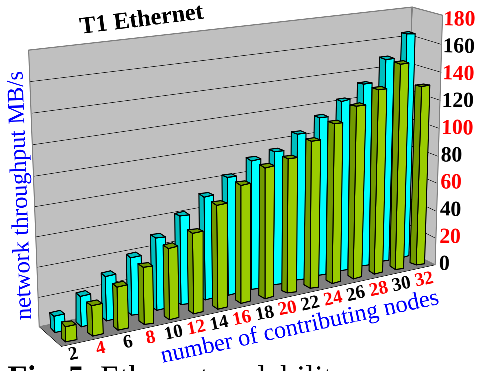

Fig. 5. Ethernet scalability

Let us explore the scalability of our parallel I/O implementation (SFIO) as a function of the number of contributing I/O nodes. Performance results have been measured on the Swiss-T1 machine [7]. Swiss-T1 consists of 64 Alpha processors grouped in 32 nodes. Two types of networks are used, Tnet and Fast Ethernet. To have an idea about the network capabilities, throughput as a function of number of nodes is measured by a simple MPI program for both networks. The nodes are equally divided into transmitting and receiving nodes and maximal all-to-all traffic is created.

Fig. 5 demonstrates cluster throughput scalability with a Fast Ethernet Network and Fig. 6. with TNet. With Fast Ethernet, each node is connected to a Fast Ethernet crossbar switch. The underlying topology of TNet consists of eight 12-port full crossbar switches. The blue graphs show the peak performances and the green graphs the average performances.

-

-

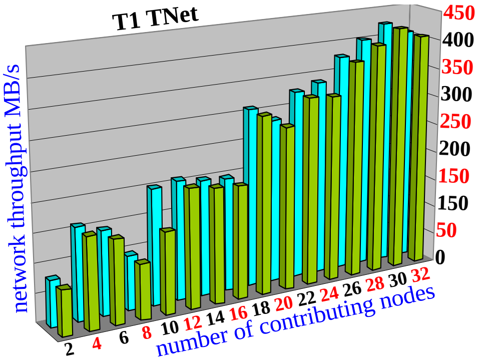

Fig. 6. TNet scalability

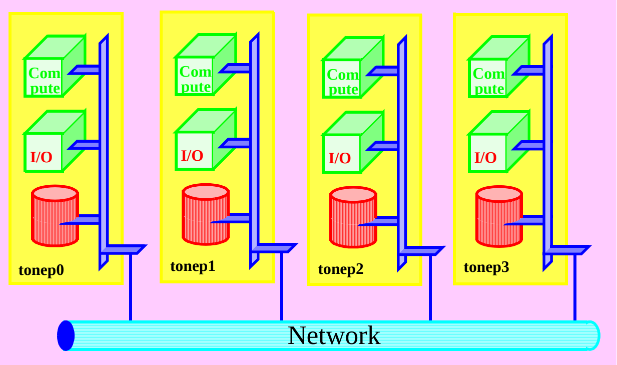

Fig. 7. SFIO Architecture on Swiss-T1

Let us now analyze the performances of the SFIO library on the Swiss-T1 machine for MPICH on Fast Ethernet and FCI-MPI on TNet. Let us assign the first processor of each compute node to a compute process and the second processor to an I/O listener (Fig. 7).

SFIO performance is measured for concurent write access from all compute nodes to all I/O nodes, the striped file being distributed over all I/O nodes. The number of I/O nodes is equal to the number of compute nodes.

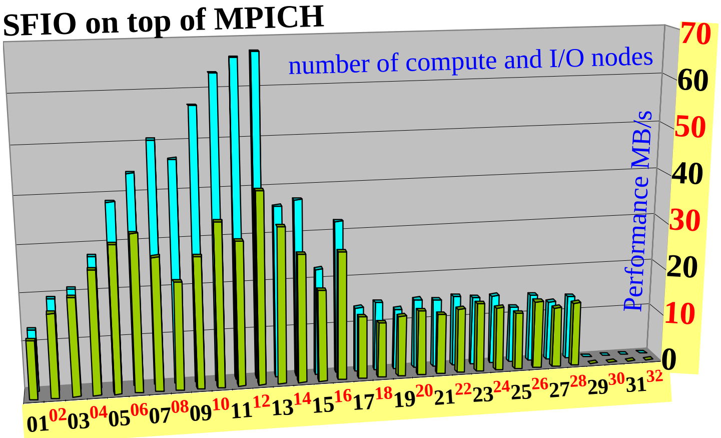

Fig. 8. SFIO all-to-all I/O performance on Fast Ethernet

The size of the striped file is 2Gbyte and the striped unit size is 200 bytes only. The application's I/O performance as a function of the number of compute and I/O nodes is measured on both Fast Ethernet and TNet and presented in Fig. 8 and Fig. 9. The blue graphs show the peak performances and the green graphs the average performances. We are very surprised with the performance results of SFIO on top of MPICH. This result needs further investigation.

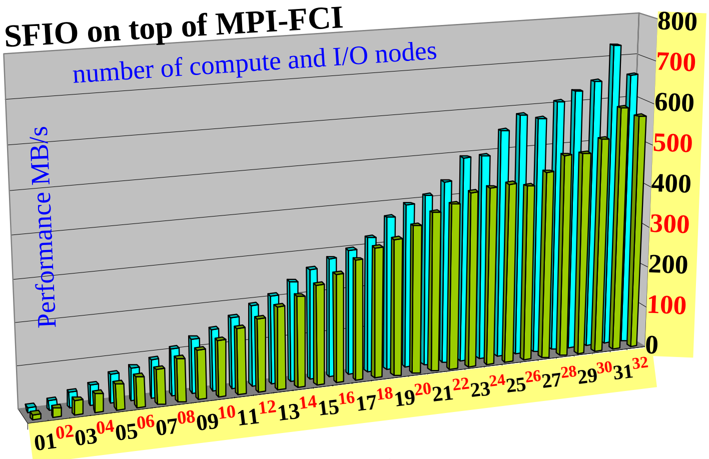

Fig. 9. SFIO all-to-all I/O performance on TNet

With MPI-FCI the situation is much better. It is highly scalable. When more than 23 nodes participate in the I/O operations, the speed-up may slightly vary due to TNet's particular communication topology. The effect of topology on the I/O performance will be further studied.

5. Conclusion and future work

SFIO is a cheap alternative to Storage Area Networks. It is a light-weight portable parallel I/O system available for MPI programmers. Integrated into standard MPI-I/O, SFIO may become a high performance portable MPI-I/O solution for the MPI community.

We plan to realize SFIO benchmarking and check scalability for larger numbers of processors on large supercomputers, e.g. at Sandia National Laboratory.

We intend to implement nonblocking parallel I/O function calls. Disk access optimisations may also be further improved.

Finally we are planning to implement the collective operations as follows: collective operations assume that all compute nodes issue an I/O request at the same logical step in the program. The compute nodes, under control of SFIO library, consult each other to arrive at a common I/O strategy. The I/O nodes are informed about the strategy by the compute nodes and SFIO creates the optimized data flow.

References

[1] Sachin More, Alok Choudhray, Ian Foster, Ming Q. Xu. MTIO a multi-threaded parallel I/O system, Proceedings of the 11th International Parallel Processing Symposium (IPPS '97), pages 368-373

[2] Ron Oldfield and David Kotz. The Armada Parallel File System, Dartmouth College Dpt. of Compute Science, November 22, 1998, pages 1-14, http://www.cs.dartmouth.edu/~dfk/armada/design.html

[3] Benoit A. Gennart, Emin Gabrielyan, Roger D. Hersch, Parallel File Striping on the Swiss-Tx Architecture, EPFL Supercomputing Review, Nov. 99, pp. 15-22, http://sawww.epfl.ch/SIC/SA/publications/SCR99/scr11-page15.html

[4] Rajeev Thakur, William Gropp, Ewing Lusk, On Implementing MPI-IO Portably and with High Performance, Sixth Workshop on I/O in Parallel and Distributed Systems, ACM, May 1999, pp. 23-32.

[5] V. Messerli, O. Figueiredo, B. Gennart, R.D. Hersch, Parallelizing I/O intensive Image Access and Processing Applications, IEEE Concurrency, Vol. 7, No. 2, April-June 1999, pp. 28-37

[6] Martha Bancroft, Nick Bear, Jim Finlayson, Robert Hill, Richard Isicoff and Hoot Thompson, Functionality and Performance Evaluation of File Systems for Storage Area Networks (SAN), 17-th IEEE Symp. on Mass storage systems, University of Maryland, March 2000, http://esdis-it.gsfc.nasa.gov/msst/conf2000/PAPERS/A05PA.PDF

[7] Pierre Kuonen, Ralf Gruber, Parallel computer architectures for commodity computing and the Swiss-T1 machine. EPFL Supercomputing Review, Nov 99, pp. 3-11.

[8] Stephan Brauss, Communication Libraries for the Swiss-Tx Machines. EPFL Supercomputing Review, Nov 99, pp. 12-15.

[9] Peter S. Pacheco, Parallel Programming with MPI, by Morgan Kaufmann Publishers, pages 137-178, 1997

[10] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, Jack Dongarra, MPI - The Complete Reference, Volume 1, The MPI Core, MIT Press, pages 123-189, 1996

[11] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, Marc Snir, MPI - The Complete Reference, Volume 2, The MPI Extensions, MIT Press, pages 185-274, 1998

[12] William Gropp, Ewing Lusk, Rajeev Thakur, Using MPI-2 Advanced Features of the Message-Passing Interface, MIT Press, pages 51-118, 1999

[13] Message Passing Interface Forum, MPI-2 Extentions to the Message-Passing Interface, University of Tennessee, pages 209-300, 1997

[14] Rajeev Thakur, William Gropp, Ewing Lusk “A Case for Using MPI’s Derived Datatypes to Improve I/O Performance” http://www.supercomp.org/sc98/TechPapers/sc98_FullAbstracts/Thakur447/, pages 1-9, 1998