SFIO a striped file I/O library for MPI

Emin Gabrielyan, Roger D. Hersch
École Polytechnique Fédérale de Lausanne, Switzerland
{Emin.Gabrielyan,RD.Hersch}@epfl.ch

Abstract

This paper presents the design and evaluation of a Striped File I/O (SFIO) library for parallel I/O within 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 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 may represent 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 [1].

Parallel I/O systems should offer highly concurrent access capabilities to the common data files by all parallel application processes. They should exhibit linear increase in performance when increasing both the number of I/O nodes and the number of application's processing nodes. Parallelism for input/output operations can be achieved by striping the data across multiple disks so that read and write operations occur in parallel (see Fig. 1). A number of parallel file systems were designed ([2], [3], [4], [5], [6], [7], [8]), which rely on the parallel file striping paradigm.

MPI is a widely used standard framework for creating parallel applications running on various types of parallel computers [9]. A well known implementation of MPI, called MPICH, has been developed by Argone National Laboratory [10]. MPICH is used on different platforms and incorporates MPI-1.2 operations [11] as

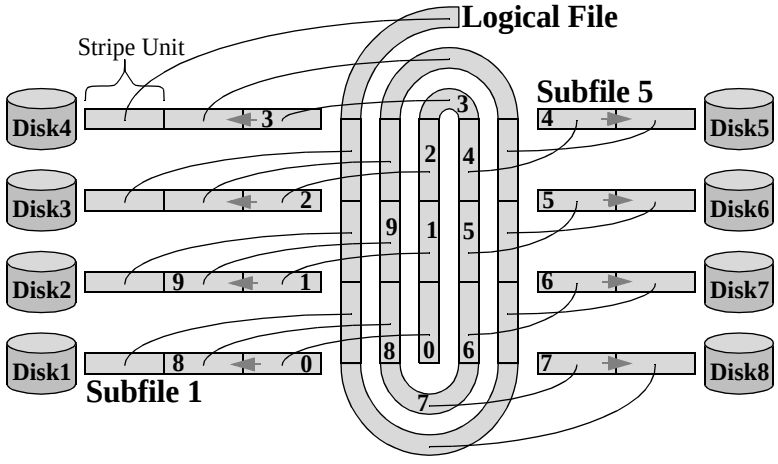

Fig. 1. File Striping

well as the MPI-I/O subset of MPI-II ([12], [13], [14]). 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 sequential and is based on NFS [10], [15].

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.

Another factor reducing peak performance is the read-modify-write operation useful for writing fragmented data to the target file. Read-modify-write requires reading the full contiguous extension of data covering the data fragment to be written, sending it 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.

Our project aims at offering scalable I/O throughput. 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. At the present time, the SFIO library interface does not provide non-blocking 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 optimizations are presented in Section 3. First performance results are given for various configurations of the Swiss-Tx supercomputer [16]. The performance test of SFIO on top of MPICH is given in section 4. In the same section we present the topology of the Swiss-T1 machine and the SFIO benchmarks on top of the native FCI communication system. Section 5 presents the conclusions and future work.

2. Global Architecture

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 the Unix and the Windows NT versions.

The SFIO parallel file striping library offers a simple Unix like interface. We intend to provide in the future an MPI-I/O interface on top of SFIO. The intermediate level of MPICH's MPI-I/O implementation is ADIO [15]. We successfully modified the ADIO layer of MPICH to route calls to the SFIO interface (Fig.2).

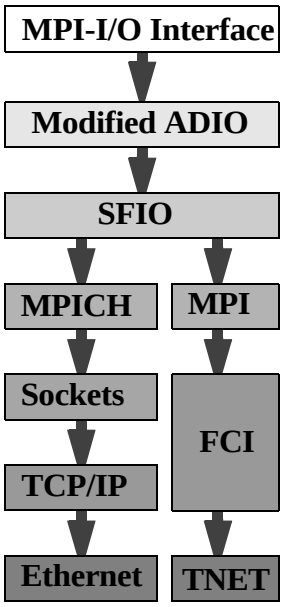

Fig. 2. SFIO integration into MPI-I/O

On the Swiss-T1 machine, SFIO can run on top of MPICH as well as on top of MPI/FCI, an MPI implementation making use of the low latency and high throughput TNET network [17].

Unlike the majority of file access sub-systems SFIO is not a block-oriented library [18],[5],[19],[20],[21]. Independance from block orientation provides a number of advantages. There is no need to send entire blocks over the network or to access them on the disk. The stripe units do not form blocks; neither network transfers nor disk accesses are rounded to the size of the stripe unit size. The amount of data accessed on the disk and transferred over the network is the size specified at the application level.

The functional architecture of the SFIO library is shown in Fig. 3. Only the structure of the access functions is described. On top of the graph we have the application's

interface to data access operations and at the bottom, we have the I/O node operations.

The *mread* and *mwrite* operations are the non-optimized single block access functions and the *mreadc* and *mwritec* operations are their optimized counterparts. The *mreadb* and *mwriteb* operations are multi-block access functions. All the *mread, mwrite, mreadc, mwritec, mreadb, mwriteb* interface functions are operating at the level of the logical file. For example the SFIO write access operation *mwritec(f,0,buffer,size)* writes data to the beginning of the logical file *f*. Access interface functions are unaware of the fact that the logical file is striped across subfiles. In the SFIO library, all the interface access functions are routed to the *mrw* cyclic distribution module. This module is responsible for data striping. Contiguous requests (or a set of contiguous requests for *mwriteb* and *mreadb*) are split into small fragments according to the striping factor. The small requests generated by the *mrw* module specify the selected subfile, and the node on which the subfile is located. Global pointers are translated to subfile pointers. Subfile access requests contain enough information to execute and complete the I/O operation. For the non-optimized *mread* and *mwrite* operations, the library routes the requests to the *sfp_read* and *sfp_write* modules that are responsible to send appropriate single subrequests to the I/O nodes using MPI as the transport layer.

The network communication and disk access optimization is demonstrated by the remaining part of the graph. For the optimized interface functions *mreadc, mwritec, mreadb, mwriteb*, the *mrw* module routes the requests to

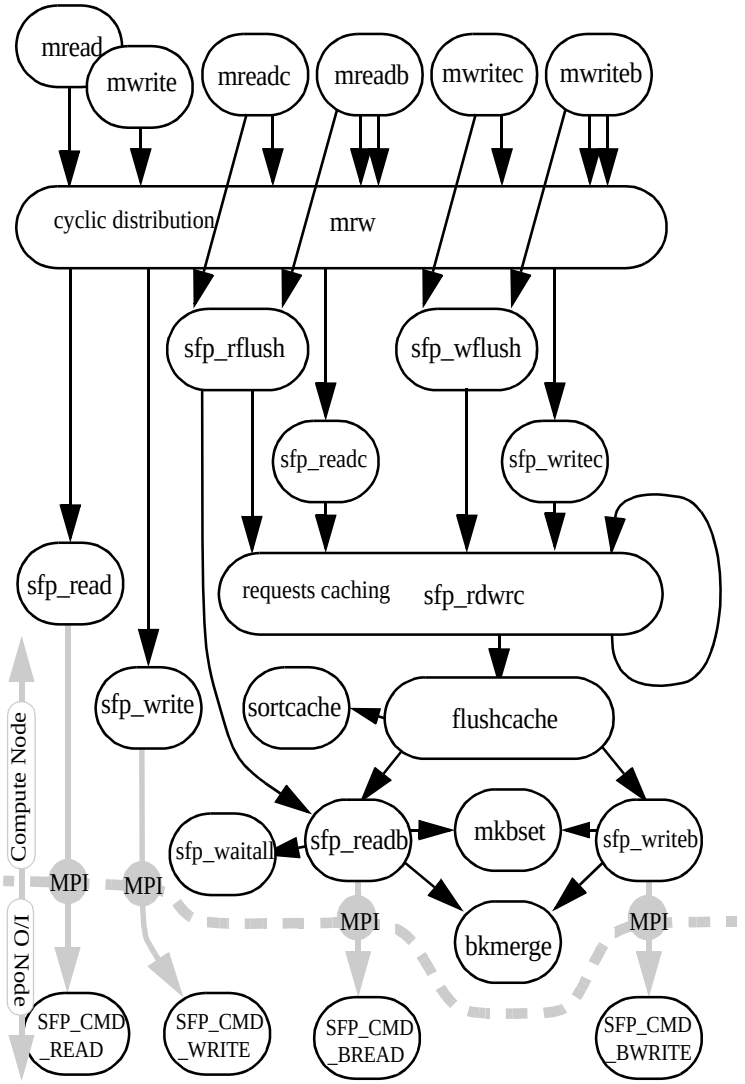

Fig. 3. SFIO functional architecture

the *sfp_readc* and *sfp_writec* functions. These functions access the *sfp_rdwrc* module which stores the sub-requests into a two-dimensional cache. The 2D cache structure comprises as one axis the I/O nodes and as a second axis the set of subfiles each I/O node is dealing with. In the general case, on each I/O node there may be one subfile per global file.

Each entry of the cache can be flushed. Flushing happens either because the user operation terminates, i.e. when a signal is communicated down through the *sfp_rflush* and *sfp_wflush* functions; or it can happen if the *sfp_rdwrc* module predicts a possible overflow of reception buffers on the remote I/O nodes. The *sfp_rdwrc* module makes sure that all generated requests fit within the buffers of the compute and of the remote I/O nodes. The entry to be flushed is passed to the *flushcache* operation that also frees the relative resources within the cache.

As soon as a large list of the sub-requests needs to be processed, the library can carry out an effective optimizations in order to save network communications and disk accesses. Note that the data itself is never cached, and always stays in user space. Three optimization procedures are carried out, before an actual transmission takes place. The requests are sorted by their offsets in the remote subfiles. This operation is carried out by the *sortcache* module. Overlapping and consecutive requests are merged whenever possible into single requests by the *bkmerge* module. This merging operation reduces the number of disk access calls on the remote I/O nodes.

The *mkbset* module creates a derived MPI datatype pointing to the fragmented pieces of user data in the user's memory. This allows to efficiently transmit the data associated to many requests over the network as one contiguous stream. The data can be transmitted or received without any memory copy at the application or library level.

The actual data transmission to the I/O nodes is carried out by the *sfp_readb* and *sfp_writeb* functions.

3. The Unix like SFIO interface

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 the correct behaviour 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 C code shows a simple SFIO example. The striped file with a stripe unit size of 5 bytes consists of two subfiles. 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 "/usr/local/sfio/mio.h"
int _main(int argc, char *argv[])
{
    MFILE *f;
    f=mopen //Collective Operation
    (
        "t0-p1,/tmp/a1.dat;"
        "t0-p2,/tmp/a2.dat;"
        ,5
    );
    if(rank()==0)
    {
        //writes at location 0 in the
        //global file 11 characters
        mwritec(f,0,"Hello World",11);
    }
    mclose(f);  //Collective Operation
}
```

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 "/usr/local/sfio/mio.h"
int _main(int argc, char *argv[])
{
    MFILE *f;
    char bu[]=”Hello*World!*”;
    int r=rank();
    f=mopen
    (
        "t0-p1,/tmp/a.dat;"
        "t0-p2,/tmp/a.dat;"
        ,5
    );
    //each process writes at its own
    //position 13 characters
    mwritec(f,13*r,bu,13);
    mclose(f);
}
```

We assume that the program is launched with three compute and two I/O MPI processes. After the parallel writing operation, the global file contains the text com-

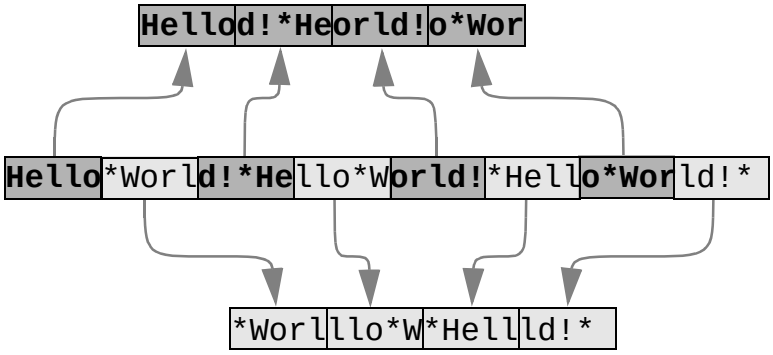

Fig. 4. Distribution of striped file across subfiles

bined from the fragments written by the first, second and third compute nodes, i. e.
"**Hello***Worl**d!*He**llo*W**orld!***Hell**o*Wor**ld!*"
The text is distributed across the two subfiles. The first subfile contains "**Hellod!*Heorld!o*Wor**" and the second "*Worlllo*W*Hellld!*" (Fig. 4)

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 stripeUnitSz);
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 specification of the I/O nodes together with the paths of subfiles representing the global striped file. The format of the name is a sequence of subfiles, separated by semicolon:
"<host>,<path>;<host>,<path>...".
For example
"tonep0,/tmp/a.dat;tonep1,/tmp/a.dat;".
The *mchsize* operation change the size of the logical file. If the specified size is smaller than the current, the operation truncate logical file to the new size.

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 numberOfBlocks,
    long offsets[],
    char *buffers[],
    unsigned sizes[]);
void mwriteb(MFILE *f,
    unsigned numberOfBlocks,
    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. The multiple block data access operations *mreadb* and *mwriteb* are optimized. The *numberOfBlocks* argument in *mreadb* and *mwriteb* operations specifies the number of blocks to be accessed by the single operation in the logical file. The information about each block has to be provided by three arrays *offsets*, *buffers* and *sizes* each having a number of elements given by the variable *numberOfBlocks*. The *offsets* array contains the positions of each block in the logical file. The *buffers* array contains the addresses of each block in the user memory and the *sizes* aray stores the size of each block in bytes.

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

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

merror and *merrora* return an array of error statistics 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 statistics to the standard output of the application.

Implementation

In our programming model, we assume a set of compute nodes and an I/O subsystem. The I/O subsystem comprises a set of I/O nodes running I/O listener processes. Both compute processes 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 optimization 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, each I/O listener prepares a reply, which is sent back to the compute node.

Optimization

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. 5). Requests queued for each I/O node are sorted according to 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 chunk. Upon reception of a data write or read request, the I/O node immediately launches the corresponding disk access request.

4. SFIO performance

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 [16]. The Swiss-T1 supercomputer is based on Compaq AlphaServer DS20 machines and consists of 64 Alpha processors grouped in 32 nodes. Two types of networks are interconnecting the processors, the 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 all-to-all traffic is generated. Fig. 6 demonstrates the

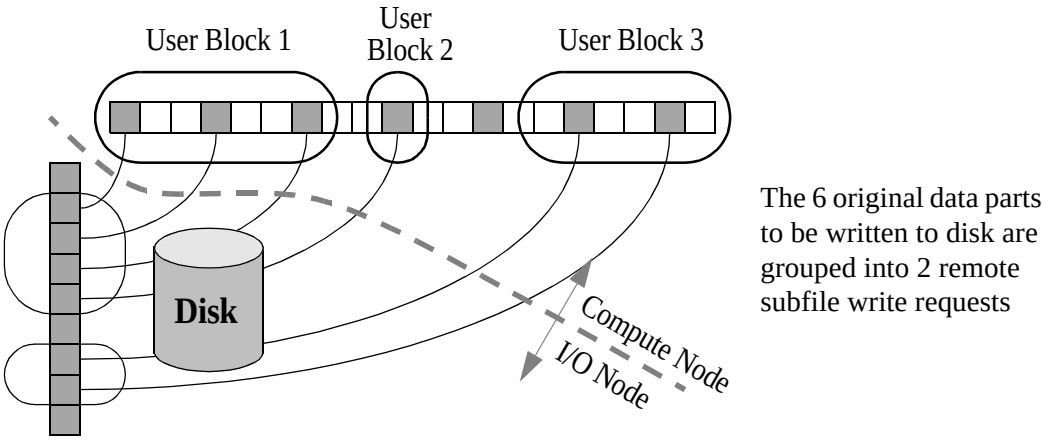

Fig. 5. Disk Access Optimisation

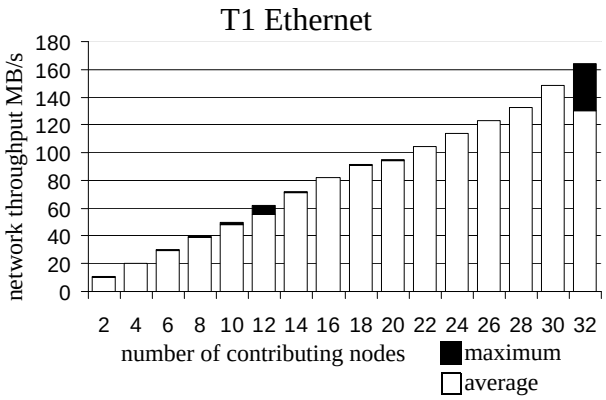

Fig. 6. Aggregate throughput of Fast Ethernet as a function of the number of contributing nodes

cluster's communication throughput scalability over Fast Ethernet. The Fast Ethernet network of T1 consists of a full crossbar switch.

Let us now analyze the performances of the SFIO library on the Swiss-T1 machine on top of MPICH using Fast Ethernet. We 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 concurrent 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.

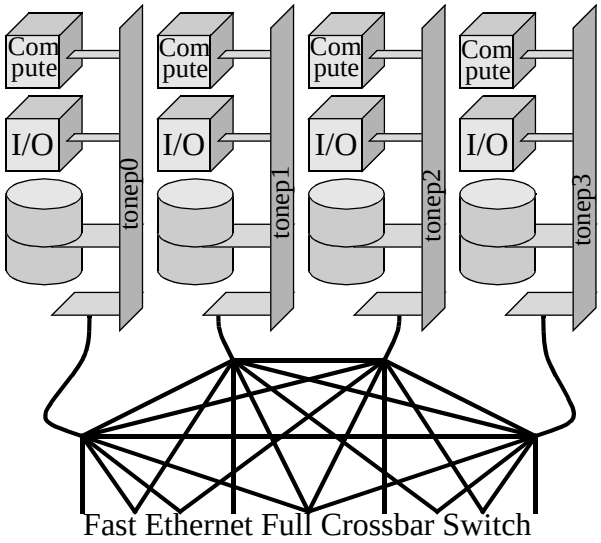

Fig. 7. SFIO architecture on Swiss-T1

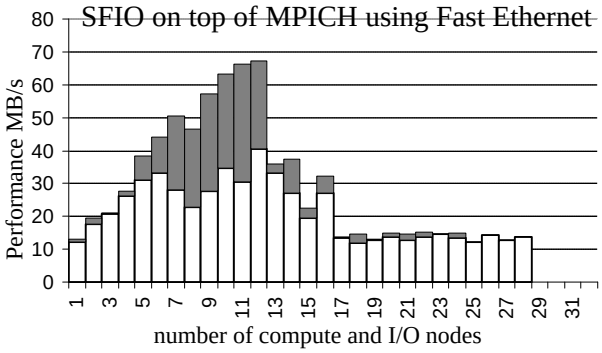

Fig. 8. SFIO/MPICH all-to-all I/O performance

The size of the striped file is 2Gbyte and the striped unit size is 200 bytes only. The MPICH application's I/O performance as a function of the number of compute and I/O nodes is measured for the Fast Ethernet network. It is presented in Fig. 8. The white graph represents the average throughput and the gray graph the peak performance. These results are surprising and need further investigation. We suppose that the fall of the performance may be possibly due to a non-efficient implementation of data intensive collective operations in the current version of MPICH.

Let us analyze the capacities of the TNET network of the Swiss-T1 machine. TNET is a high throughput and low latency network (less than 20ms MPI latency and more than 50MB/s bandwidth) [17]. A high performance MPI implementation called MPI/FCI is available for communication through TNET [17].

The TNET throughput as a function of the number of nodes is measured by a simple MPI program. The contrib-

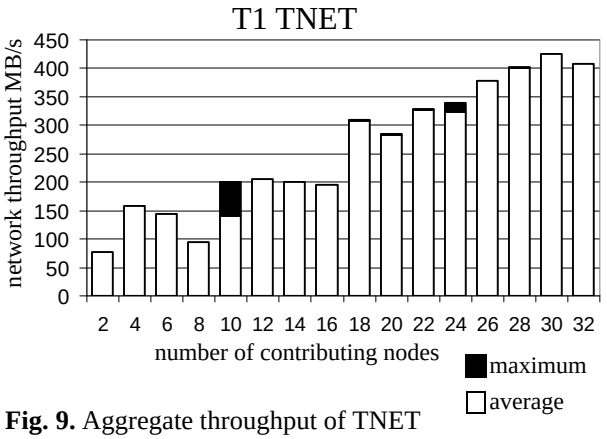

Fig. 9. Aggregate throughput of TNET as a function of the number of contributing nodes

uting nodes are equally divided into transmitting and receiving nodes (Fig. 9). Due to TNET's specific network topology (Fig. 10), communication throughput does not increase smoothly. A significant increase in throughput occurs when the number of nodes increases from 8 to 10, 16 to 18 and 24 to 26 nodes.

The Swiss-T1's TNET network [22] consists of eight 12-port full crossbar switches (Fig. 10). Routing between switches that do not have direct connectivity is static [16]. The topology together with the routing information defines the network's peak throughput over the subset of processors assigned to a given application.

Let us now analyze the performances of the SFIO library on the Swiss-T1 machine on top of MPI/FCI using the proprietary TNET network. As before, the first processor of each compute node is assigned to a compute process and the second processor to an I/O listener process. Therefore, each node acts both as a compute node and an I/O node.

As in SFIO/MPICH, the performance of SFIO over MPI/FCI is measured for concurrent write accesses from all compute nodes to all I/O nodes, the striped file being distributed over all I/O nodes.

In order to limit operating system caching effects, the total size of the striped file linearly increases with the number of I/O nodes. With a global file size proportional to the number of contributing I/O nodes, we keep the size of subfiles per I/O node fixed at 1GB/subfile.

The stripe unit size is 200 bytes. The MPI/FCI application's I/O performance is measured as a function of the number of compute and I/O nodes (Fig. 11). For each configuration, 53 measurements are carried out. At job launch

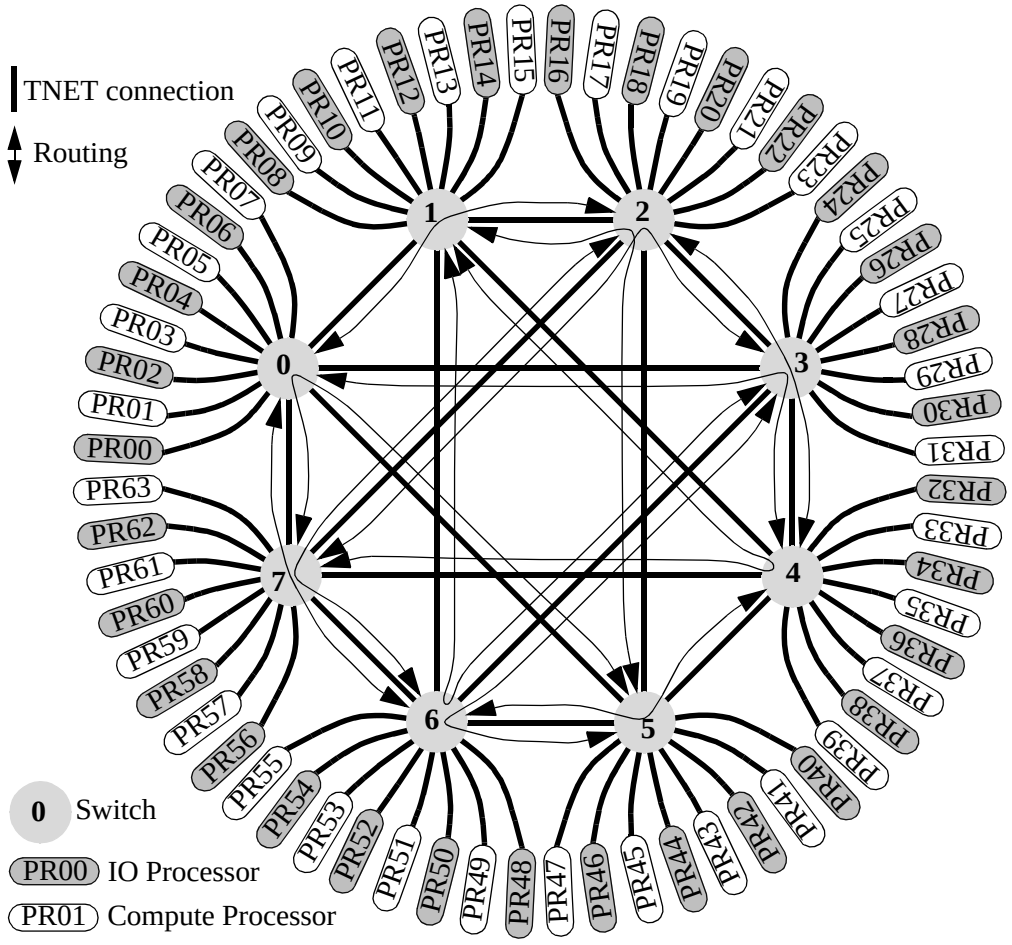

Fig. 10. The Swiss-T1 network interconnection topology

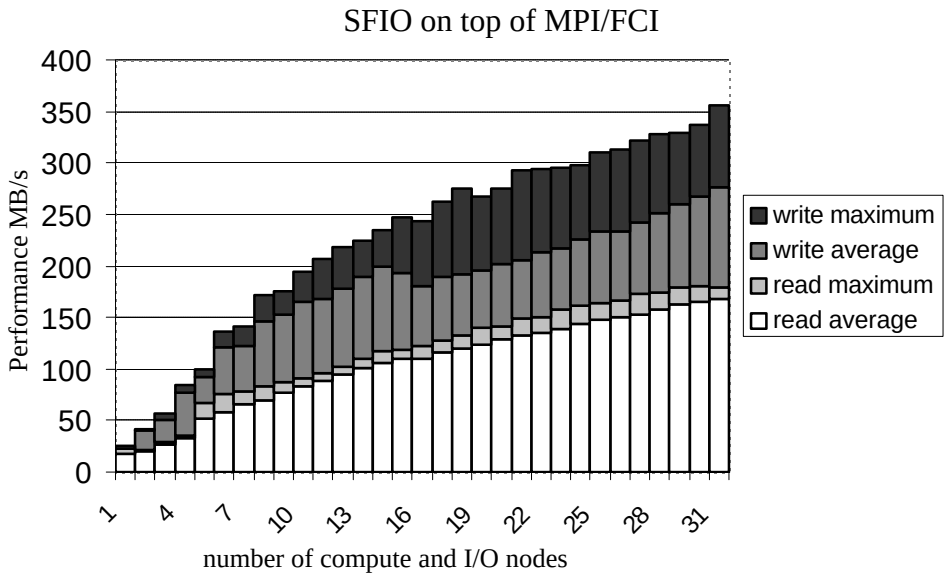

Fig. 11. SFIO all-to-all I/O performance on TNET

time, pairs of I/O and compute processes are assigned randomly to processing nodes.

The I/O throughput on MPI/FCI scales well when increasing the number of nodes. The speed-up may slightly vary due to TNET's particular communication topology (Fig. 10). The effect of topology on the I/O performance needs to be further studied.

4. Conclusion and future work

SFIO is a cheap alternative to Storage Area Networks. It is a light-weight portable parallel I/O system 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 check the scalability of SFIO for a larger number of processors on a large supercomputer at Sandia National Laboratory.

We intend to implement non-blocking parallel I/O function calls. Disk access optimizations may also be further improved. The library has to be further developed in order to support global files larger than 4GB.

Finally, we are planning to implement 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 the SFIO library, consult each other to arrive at a common I/O strategy. The I/O nodes are informed about the strategy of the compute nodes and an optimized data flow schedule is created.

References

[1] 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*

[2] Sachin More, Alok Choudhary, 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

[3] 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*

[4] 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

[5] Chandramohan A. Thekkath, Timothy Mann, Edward K. Lee, Frangipani: A Scalable Distributed File System. In Proceedings of the 16th ACM Symposium on Operating Systems Principles, pages 224-

237. ACM Press, October 1997, *ftp://ftp.digital.com/pub/DEC/SRC/publications/thekkath/frangipani-sosp.ps*

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

[7] Jay Huber, Christopher L. Elford, Daniel A. Reed, Andrew A. Chien, and David S. Blumenthal. PPFS: A High Performance Portable Parallel File System. In Proceeding of the 9th ACM International Conference on Supercomputing, pages 385-394. ACM Press, July 1995.

[8] David Kotz. Disk-directed I/O for MIMD Multiprocessors. ACM Transactions on Computer Systems, 15(1):41-74, February 1997.

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

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

[11] 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

[12] 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

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

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

[15] 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

[16] Pierre Kuonen, Ralf Gruber, Parallel computer architectures for commodity computing and the Swiss-T1 machine. EPFL Supercomputing Review, Nov 99, pp. 3-11, *http://sawww.epfl.ch/SIC/SA/publications/SCR99/scr11-page3.html*

[17] Stephan Brauss, Communication Libraries for the Swiss-Tx Machines. EPFL Supercomputing Review, Nov 99, pp. 12-15. *http://sawww.epfl.ch/SIC/SA/publications/SCR99/scr11-page12.html*

[18] 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*

[19] Edward K. Lee, Highly-Available, Scalable Network Storage, In Digest of Papers COMPCON 1995, pages 397-402. IEEE Computer Society Press, March 1995.

[20] Edward K. Lee and Chandramohan A. Thekkath, Petal: Distributed Virtual Disks, In Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS-VII, pages 84-92. ACM, October 1996. *ftp://ftp.digital.com/pub/DEC/SRC/publications/eklee/petal-paper.pdf*

[21] Edward K. Lee, Chandramohan A. Thekkath, Chris Whitaker, Jim Hogg, A Comparison of Two Distributed Disk Systems, SRC Research Report 155, April 30, 1998, *http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-155.html*

[22] Kuonen P. The K-Ring: a versatile model for the design of MIMD computer topology, Proceedings of the High-Performance Computing Conference (HPC'99), San Diego, USA, pp. 381-385; April (1999).