Acknowledgements

Summary

Résumé

Table of Contents

Chapter 1. Introduction

Section 1.1. Parallel communication challenges

We do not know if parallel communications were first used for bandwidth enhancement or for fault-tolerance. Laying the first transatlantic cable took entrepreneur Cyrus Field twelve years and four failed expeditions. Cables were constantly snapping and could not be recovered from the ocean floor. On 5 August 1858 a cable started to operate, but for a very short time. It stopped operating on September 18. Eight years later, on 13 July 1866, the Great Eastern, by far the largest ship, began laying a cable, this time made of a single piece, 2730 nautical miles long, insulated with a new resin from the gutta-percha tree growing in Malay Archipelago. When two weeks later, on 27th of July 1866, the cable began operating, the mission for Cyrus Field was not yet accomplished. He immediately sent back the Great Eastern to sea for landing the second parallel cable. By 17 September 1866, not one, but two parallel circuits were sending messages across the Atlantic.

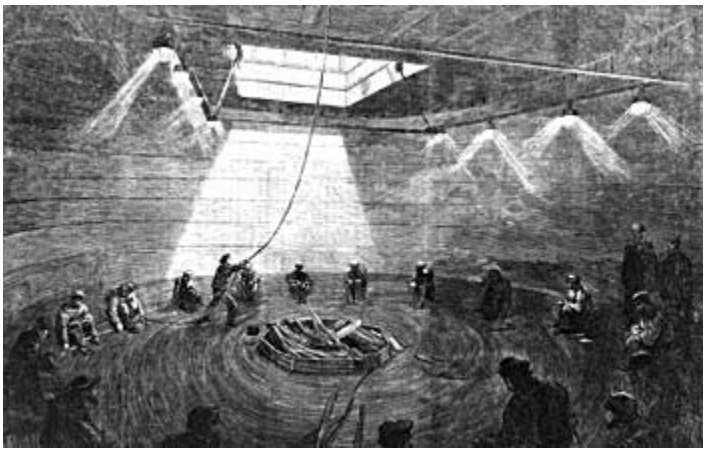

Figure 1. Loading the transatlantic cable into the 'Great Eastern' in 1865

This transatlantic cable station was transmitting messages nearly 100 years. It was still in operation when in March 1964, in the middle of the cold war, Paul Baran wrote an article "On Distributed Communications Networks". At that time Paul Baran was working on a communication method which could withstand a nuclear attack and enable transmissions of vital information across the country [Baran64], [Baran65]. Paul Baran concluded that extremely survivable networks can be built if structured with parallel redundant paths. He has shown that even moderated redundancy permits withstanding extremely heavy weapon attacks. In 1965, the Air Force approved testing of Baran's theory. Four years later, on 1st October 1969, the progenitor of the global Internet, the Advanced Research Projects Agency Network (ARPANET) of the U.S. Department of Defense, was born.

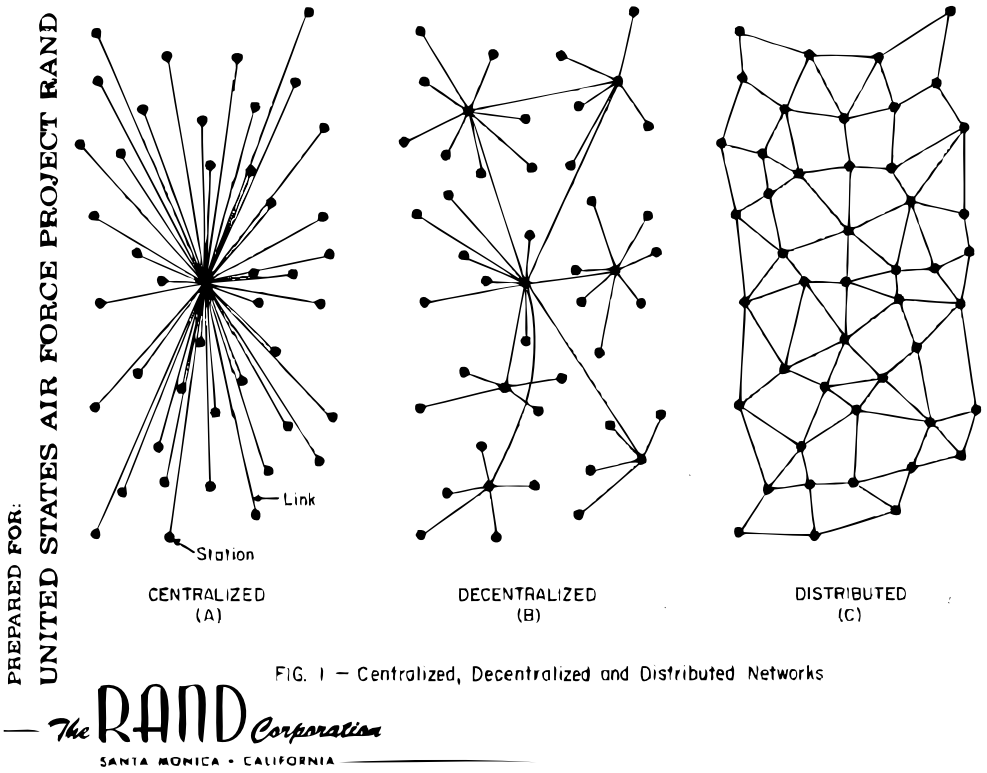

Figure 2. Diagrams from the 51-page report of Paul Baran to the U.S. Air Force, 1964

While the inspiration for structuring the early Internet with parallel paths came from the challenge to achieve a high tolerance to failures, almost a decade later IBM, at a much smaller scale, invented a parallel communication port for achieving faster communications. Since then, many other research directions relying on parallel and distributed communications have developed. Thanks to parallel communications uniform battery power consumption maximizing the network lifetime can be achieved in sensor and ad-hoc networks (energy efficiency and power-aware routing) [Ping06], [Luo06], [Kim06]. Parallelizing the communications across independent networks aims at offering additional security and protection of information, e.g. in voice over IP networks. Redundant parallel transmissions can be required for precision purposes, e.g. in GPS.

Section 1.2. Capacity enhancement and fault-tolerance

The focus of the research in parallel communications is maximizing the capacity and the fault-tolerance. Bandwidth is enhanced by using several parallel circuits between a source and a destination [Hoang06]. Yet a greater level of parallelism can be achieved by distributing the sources and destinations across the network. For example distributing storage resources in parallel I/O systems parallelizes both the I/O access and the communications.

Regarding fault-tolerance, the nature has created many systems relying on parallel structures. When developing its distributed network models (the seeds of the Internet), Paul Baran

was inspired himself from discussions with neurophysiologist Warren Sturgis McCulloch about the capability of the brain to recover lost functions by bypassing a dysfunctional region thanks to parallel structures. The living multi-cellular organisms from insects to vertebrates demonstrate numerous other examples of duplicated organs that are functioning in parallel. The evolution of life on earth made reduplicated organs nearly a universal property of living bodies [Gregory35].

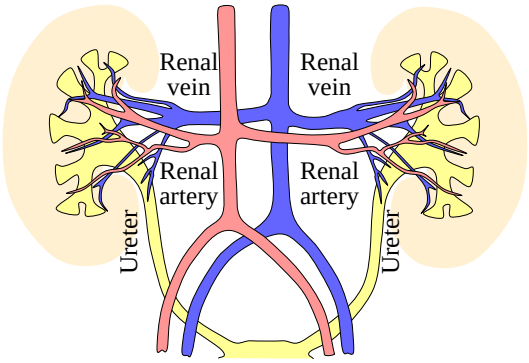

Figure 3. Kidney blood filtering in the human organism

Very often, the primary purpose of duplication of organs is the tolerance to failures while the capacity enhancement is of a secondary importance. The ideas of achieving extremely high levels of fault-tolerance in bio-inspired electronic systems of the future (e.g. by reproducing and healing) have always intrigued engineers and stimulated their imaginations [Bradley00].

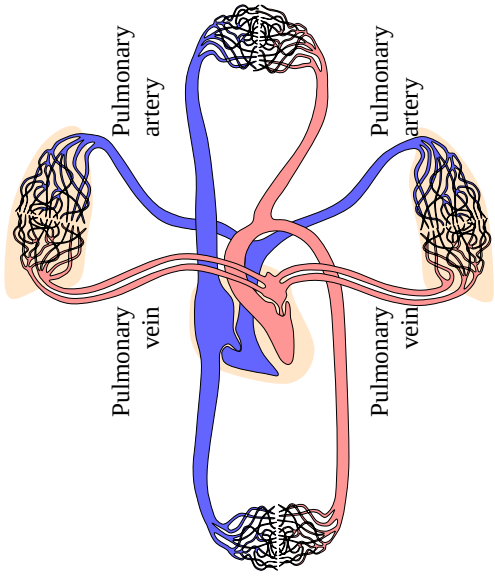

Figure 4. Pulmonary circuit of the human organism

Maintaining an idle parallel resource has been already used in many mission-critical man-made systems. In networking the communication can switch (often automatically) to a backup

path in case of failures of primary links. An appealing approach is however to use the parallel resources simultaneously, similarly to biological organisms (see Figure 3 and Figure 4). This is possible thanks to packetized communications where the communication can be routed simultaneously over several parallel paths. Individual failures should cause only minimal damages to the communication flow.

Section 1.3. Fine-grained and coarse-grained network paradigms

1.3.1. Packet switching or hot potato routing

Store and forward routing was simultaneously and independently invented by Donald Davies and Paul Baran. The term "packet switching" comes from Donald Davies. Paul Baran called this technique "hot potato routing" [Boehm64], [Davies72], [Baran02]. Today's internet relies on a store-and-forward policy: each switch or router waits for the full packet to arrive before sending it to the next switch. The first store and forward routers of ARPANET were called Interface Message Processors (see Figure 1).

Figure 5. One of the first Interface Message Processor (IMP) of ARPANET connecting UCLA with SRI in August 1969

The router in packet switched networks maintains queues for processing, routing and transmitting through one of the outgoing interfaces. No circuit is reserved from a source to a destination. There is no bandwidth reservation policy. This may lead to contentions and congestions. To avoid congestions in the packet discarding method, if a packet arrives at a switch and no room is left in the buffer, the packet is simply discarded (e.g., UDP). The adjustable window method gives the original sender the right to send *N* packets before getting permission to send more (e.g., TCP).

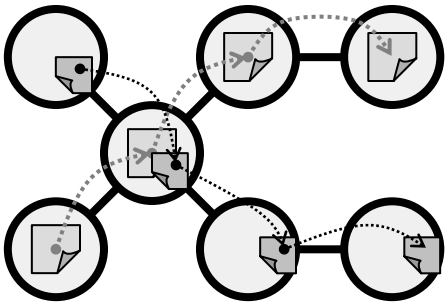

Figure 6. Packet switching network: packets are entirely stored at each intermediate switch and then only forwarded to the next switch

Since the packets are completely stored at each intermediate switch before being transmitted to the next hop, a communication delay propagates between the end nodes as the number of hops separating the nodes increases. The communication delay is a function of the number of intermediate switches multiplied by the size of the packet.

1.3.2. Wormhole routing

Wormhole or cut-through routing is used in multiprocessor and cluster computer networks aiming at high performance and low latency. Store and forward switching technology cannot meet the strict bounds on the communication latencies dictated by requirements of a computing cluster. Wormhole routing technology is solving the problem of the propagation of the delay across a multi-hop communication path in store-and-forward switching.

The short address is translated at an intermediate switch before the message itself arrives. Thus, as soon as the message starts arriving, the switch very quickly examines the header without waiting for the entire message, decides where to send the message, sets up an outgoing circuit to the next switch and then immediately starts directing the rest of the message that is being received to the outgoing interface. The switch transmits the message out, through an outgoing link, at the same time as the message arrives. By quickly setting up the routing at each intermediate switch and by directing the message content to the outgoing circuit without storing the message, the message traverses the network at once, simultaneously through all intermediate links of the path. The destination node, even if it is many hops away, starts receiving the message nearly as soon as the sending node starts its transmission. The message is simply “copied” from the source to the destination without being ever entirely stored anywhere in between (Figure 7).

This technique is implemented by breaking the packets into very small pieces called flits (flow units). The first flit sets up the routing behavior for all subsequent flits associated with the message. The messages rarely (if ever) have any delay as they travel though the network. The latency between two nodes, even if separated by many hops, becomes similar to the latency of directly connected nodes.

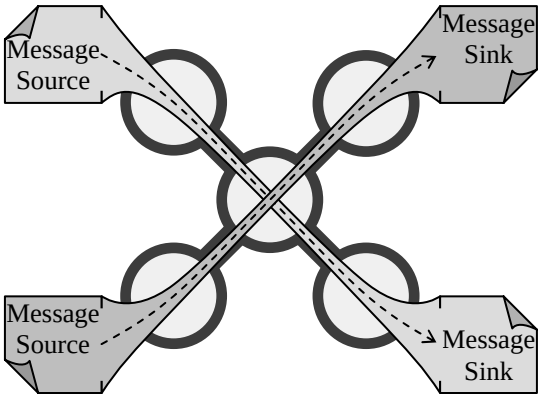

Figure 7. Wormhole or cut-through routing network: a packet is "copied" through the communication path from the source directly to the destination without being stored in any intermediate switch

MYRINET is an example of a wormhole routing network for cluster supercomputers. MPI is the most popular communication library for these networks.

Wormhole routing and store-and-forward packet switching are examples from two well known network paradigms. Packet switching belongs to the fine-grained network paradigm and wormhole routing is an example of the coarse-grained circuit switching paradigm. Nearly all coarse-grained networks are aiming at low latencies and use connection oriented transmission methods. ATM, frame relay, TDM, WDM or DWDM, all-optical switching, light-path on-demand switching, Optical Burst Switching (OBS), MYRINET, wormhole routing, cut-through and virtual cut-through routing are all broadband or local area network examples of the coarse-grained switching paradigm [Worster97], [Qiao99].

Section 1.4. Three topics in parallel communications

It is hard to imagine a single study consistently covering all areas of parallel and distributed communications. In this dissertation we are focusing on three anchor topics. The first topic is parallel I/O in computer cluster networks. The second topic addresses the problems in high-speed low-latency networks arising from simultaneous parallel transmissions, e.g. those of parallel I/O requests. The third topic addresses fault-tolerance in fine-grained packetized networks.

These three topics are the main bold sides of the domains covered by parallel communications. While all these three topics rely on parallel communications, they are pursuing three orthogonal goals. For achieving the desired results we rely on techniques derived from different disciplines, such as graph theory or erasure resilient coding.

1.4.1. Problems and the objectives

Parallel I/O relies on distributed storage. The main objectives pursued in parallel I/O are a good load balance, the scalability as the number of I/O nodes grows and the throughput efficiency when multiple computing nodes are accessing concurrently a shared parallel file. Parallel I/O is

used in computer clusters, interconnected with high performance coarse-grained network (such as MYRINET) that can meet strict latency bounds. In such networks, large messages are “copied” across the network from the source computer directly to the destination computer. During of such a “copy” process, all intermediate switches and links are simultaneously involved in directing the content of the message. Low latency however induces an increased tendency to congestions. When the network paths of several transmissions overlap, an attempt to carry out them in parallel will unavoidably cause a congestion. The system becomes more prone to congestions as the size of the messages and the number of parallel transmissions increase. The routing scheme and the topology of the underlying network have significant impact. Properly orchestrating the parallel communications is necessary to achieve a true benefit in terms of the overall throughput.

In the context of fine-grained packet-switching, achieving fault tolerance by streaming information simultaneously across multiple parallel paths is a very attractive idea. Naturally, this method minimizes losses occurring from individual failures on the parallel paths, but the large number of parallel paths increases also the overall probability of individual failures influencing the communication. Streaming across parallel paths can be combined with injection at the source of a certain amount of redundant packets generated with channel coding techniques. Such a combination ensures the delivery of the information content during individual link failures on parallel paths. We propose a novel technique to measure the advantageousness of parallel routing for this combined method of parallel streaming with redundant packets.

Each of the three topics is addressed by a detailed analysis of the corresponding problems and by proposing a novel method for their solutions.

1.4.2. Structure of the thesis

The parallelism in I/O access and communication relies on distribution of the storage resources. A high level of parallelism with a high load balance can be achieved thanks to fine granularity. The drawbacks of fine granularity are the network communication and storage access overheads. In Chapter 2, we present a library called Striped File I/O (SFIO) which combines fine granularity with high performance thanks to several important optimizations. We describe the interface and the functional architecture of the SFIO system along with the the optimization techniques and their implementation. Chapter 2 is concluded by benchmarking results.

Optimized parallel I/O results in simultaneous transmissions of large data chunks over the underlying network. Since parallel I/O is mostly used in supercomputer cluster networks having strict bounds on the latency and the throughput, the underlying network typically relies on coarse-grain switching. Such networks are prone to congestions when many parallel transmissions carry very large messages. Depending on the network topology, the rate of congestions may grow so rapidly that the overall throughput is reduced despite the increase of the number of the contributing nodes. The gain achieved from the aggregation of communications in parallel I/O at the connection layer can be outperformed by losses due to blocked messages occurring at the network layer. Solving congestions locally by default FIFO method may result in idle times of other critical resources. Scheduling of transmissions at their sources aiming at an efficient

utilization of communication resources can optimally increase the application throughput. In chapter 3 we present a collective communication scheduling technique, called *liquid scheduling*, which in coarse-grained networks achieves the throughput of a fine-grained network or that of a liquid flowing through a network of pipes.

Chapter 4 is dedicated to fault-tolerant multi-path streaming in packetized fine-grained networks. We demonstrate that in packet-switched networks combination of the channel coding at the packet level with the multi-path parallel routing, significantly improves the fault-tolerance of communication, especially in real-time streaming. We show that further development of the path diversity in multi-path parallel routing patterns often brings additional benefit to the streaming application. We designed *capillary routing* algorithm generating parallel routing patterns of increasing path diversity. We also introduced a method for rating multi-path routing patterns of any complexity with a single scalar value, called ROR, standing from *Redundancy Overall Requirement*.

Chapter 2. Parallel I/O solutions for cluster computers

This chapter presents the design and evaluation of a Striped File I/O (SFIO) library providing high performance parallel I/O within a Message Passing Interface (MPI) environment. Uniform parallelization of I/O access requests and a good load balance when accessing and transferring data to and from distributed global storage rely on small striping units. Small stripe unit size, however, increases the communication and disk access cost. Thanks to the optimizations of the communications and disk accesses, SFIO exhibits high performance at very small striping factors. We present the functional architecture of SFIO system. Using MPI derived datatype capabilities, we transmit highly fragmented data over the network by single network operations. By analyzing and merging the I/O requests at the compute nodes a substantial performance gain is obtained in terms of I/O operations. At the end of the chapter we present the parallel I/O performance benchmarks on the Swiss-Tx cluster supercomputer consisting of DEC Alpha computers, interconnected with both, Fast Ethernet and a coarse-grained low latency communication network, called TNET.

Section 2.1. Introduction

The parallelism in I/O access and communication relies on distribution of the storage resources. A high level of parallelism with a high load balance can be achieved thanks to fine granularity. The drawbacks of fine granularity are the network communication and storage access overheads. The overheads resulting from fine granularity may considerably reduce the gain in throughput achieved by parallelism.

Combination of extremely fine granularity for the best load balance with a very high throughput exhibiting nearly linear scalability is the subject of the topic. Scalability and the high performance at extremely small stripe unit sizes are achievable thanks to three proposed optimization techniques.

Firstly, multi-block user interface permitting the library to recognize the overall pattern of multiple user requests is a must. Multi-block interface permits a greater network and disk access aggregation. Highly fragmented multi-block patterns of the logical file may also turn out to be relatively contiguous patterns at the level of physical sub-files. Without considering multi-block interface the optimization potential of the parallel I/O library would be seriously constrained.

Secondly, the compute nodes are equipped with a caching system of I/O requests. It aggregates all network communication transfers to and from individual I/O nodes. Network aggregation of the incoming traffic is computed and requested also by the compute nodes. The data segments are therefore traversing the network combined into very large messages, reducing thus the communication overhead to the minimum. The drawback of this method is an increase of

the risk of congestions, which is the subject of the second topic addressed in this document (see Chapter 3).

Thirdly, the caching system preprocessor at the compute nodes the collected I/O requests addressed to each individual destination. It removes the overlapping segments and sorts the requests according to their offsets. The caching preprocessor merges multiple remote I/O requests into single contiguous I/O requests, whenever it is possible. Since network transmissions to individual destinations are already being aggregated in both directions, by merging multiple requests into simple ones, no additional gain is achieved in terms of network communication. However, the performance gain from request merging at the I/O nodes is however considerable.

All three forms of optimizations carried out on the cached I/O requests are realized only at the level of memory pointers and disk offsets without accessing or copying the actual data. Once the pointers and offsets stored in the cache are optimized, a zero-copy implementation is streaming the actual data directly between the network and the fragmented memory pattern. The zero-copy implementation relies on MPI derived datatypes, built on the fly.

Section 2.2. Challenges

In 1998, the Swiss Federal Institutes of Technology in Lausanne (EPFL) and Zurich (ETHZ), the Swiss Commission for Technology and Innovation (CTI), Supercomputing Systems (SCS), and Compaq Computer Corporation, in a cooperation with two laboratories in the United States: the Sandia National Laboratory (SNL) and the Oak Ridge National Laboratory (ORNL), started a common project called Swiss-Tx with the aim to develop and build the first Swiss teraflop supercomputer. The goal was to design supercomputing systems based mainly on commodity parts. During this project several Swiss-Tx supercomputers were installed, all based on commodity Compaq Alpha Servers. Only the communication hardware and communication software are custom-made, because available off-the-shelf products, such as Ethernet with the socket interface, do not offer the necessary bandwidth, latency, and functionality.

In the course of this project, was developed and installed a new efficient communication library for commodity supercomputing, called Fast Communication Interface (FCI) and a custom-made communication hardware for the Swiss-Tx supercomputers, called TNET. TNET is a proprietary high performance network aiming at low-latency and high-bandwidth. A full implementation of the standardized MPI for TNET network was also written (on top of FCI). Early Swiss-Tx supercomputers were using EasyNet, a bus-based low-latency network. The switch-based TNET network, which is designed specifically for large and complex network topologies, has replaced EasyNet in all recent Swiss-Tx architectures [Brauss99A], [SwissTx01].

Figure 8. Swiss-Tx supercomputer in June 2001

In many parallel applications I/O is a major bottleneck. In 1998 parallel I/O was a hot topic (Appendix A). At the Peripheral Systems Laboratory of EPFL we were in charge of the design of an MPI based parallel I/O system for the Swiss-Tx parallel supercomputer.

Although the I/O subsytems of parallel machines may be designed for high performance, a large number of applications achieve only about a tenth or less of the peak I/O bandwidth [Thakur98]. The main reason for poor application-level I/O performance is that parallel-I/O systems are optimized for large accesses (of the order of megabytes), whereas parallel applications typically make many small I/O requests (of the order of kilobytes or even less). The small I/O requests made by parallel programs are due to the fact that in many parallel applications, each process needs to access a large number of relatively small pieces of data that are not contiguously located in the file [Baylor96], [Crandall95], [Kotz96], [Smirni96], [Thakur96A].

We designed the SFIO library which optimizes not only large data size accesses but also small data size accesses of an order of a fraction of one kilobyte. The extremely small stripe unit size (e.g. hundred bytes) provides very high level of load balance and parallelism. The support of a multi block Application Program Interface (API) enables the underlying I/O system to better optimize accesses to fragmented data both in memory and in the logical file. The multi-block interface of SFIO allowed us also to implement a portable MPI-I/O interface [Gabrielyan01]. Finally, thanks to the overlapping of communications and I/O and the underlying optimizations of I/O requests cached at the compute nodes, SFIO exhibits a highly competitive performance and a nearly scalable throughput even at very low stripe unit sizes (such as 75 bytes).

Section 2.3. File striping

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 [Bancroft00].

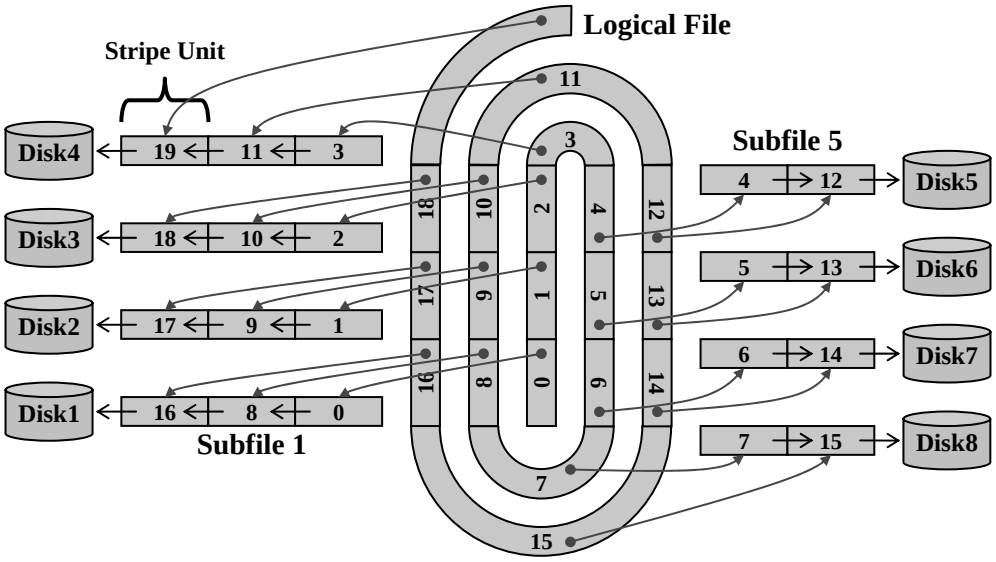

Figure 9. File Striping

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 Figure 9). A number of parallel file systems were designed ([More97], [Oldfield98], [Messerli99], [Chandramohan97], [Gorbett96], [Huber95], [Kotz97]), which rely on parallel file striping.

MPI is a widely used standard framework for creating parallel applications running on various types of parallel computers [Pacheco97]. A well known implementation of MPI, called MPICH, has been developed by Argone National Laboratory [Thakur99A]. MPICH is used on different platforms and incorporates MPI-1.2 operations [Snir96] as well as the MPI-I/O subset of MPI-II ([Gropp98], [Gropp99], [MPI2-97B]). 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 [Thakur99A], [Thakur98].

Due to the locking mechanisms needed to avoid simultaneous multiple accesses to the shared NFS file, MPICH MPI-I/O write operations could be carried out only at a very slow throughput (this version of MPICH was used in 2001).

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 fragments 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 data space within the file, network access overhead becomes dominant.

SFIO aims at offering scalable I/O throughput. The fine granularity, required for the best parallelization and load balance, dramatically increases the communication and disk access costs. Our SFIO parallel file striping implementation integrates efficient optimizations merging sets of fragmented network messages and disk accesses into single contiguous messages and disk access requests. The merging operation makes use of the MPI derived datatypes.

The SFIO library interface does not provide non-blocking operations, but internally, accesses to the network and disks are made asynchronously. Disk and network communications are overlapping in time resulting in additional gain in overall performance.

Section 2.4 presents the overall architecture of the SFIO implementation as well as the software layers providing an MPI-I/O interface on top of SFIO. The SFIO interface description and small examples are provided in Section 2.5. Optimization principles are presented in Section 2.6. The details of the system design, caching techniques and other optimizations are presented in Section 2.7. Throughput benchmarks are given for various configurations of the Swiss-Tx supercomputer [Kuonen99A]. The performances of SFIO on top of MPICH and on top of the native FCI communication system are given in Section 2.8.

Section 2.4. Implementation layers

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 Windows implementations.

The SFIO parallel file striping library offers a simple Unix like interface extended for multi-block operations. We then show how to provide an isolated MPI-I/O interface on top of SFIO [Gabrielyan01]. In MPICH's MPI-I/O implementation there is an intermediate level, called ADIO [Thakur96B], [Thakur98], which stands for Abstract Device interface for parallel I/O. We successfully modified the ADIO layer of MPICH to route calls to the SFIO interface (Figure 10).

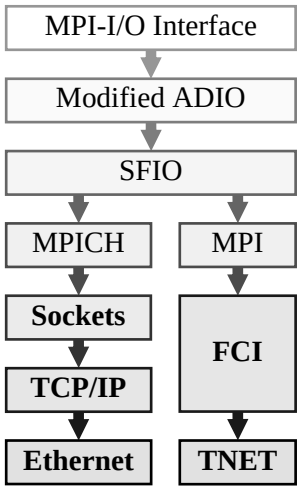

Figure 10. SFIO integration into MPI-I/O

On the Swiss-T1 machine, SFIO can run on top of MPICH as well as on top of MPI/FCI. MPI/FCI is an MPI implementation making use of the low latency and high throughput coarse-grained wormhole-routing TNET network [Horst95], [Brauss99A].

Unlike the majority of file access sub-systems SFIO is not a block-oriented library [Gennart99], [Chandramohan97], [Lee95], [Lee96], [Lee98]. Independence 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 resulting from SFIO calls.

Section 2.5. The SFIO Interface

This section presents the most frequent interface functions of SFIO. The full list of API functions is given in Appendix B. Two functions, *mopen* and *mclose* are provided to open and close a striped file. These functions are collective operations for all processing nodes. 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 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 function *mopen* returns a descriptor of the global parallel file. This function has a very simple interface. First argument of *mopen* is a single string specifying the global file name, which contains the locations and names of all subfiles. The second argument of *mopen* is the stripe unit size in bytes. The global file name format is a simple semi-column separated concatenation of local subfile names (including their hostnames) in the right order. The format is as follows:

```
“<host1>,<file1>;<host2>,<file2>;<host3>,<file3>;…”
```

For example, the following call opens a parallel file with a stripe unit size of 5 bytes consisting of two local subfiles located on hosts *node1* and *node2*:

```
f=mopen(“node1,/tmp/a.txt;node2,/tmp/a.txt”,5);
```

Other file management operations, such as *mdelete* or *mcreate* (see B.1 for file management operations) also rely on this simple format for the global file name. SFIO does not maintain any global metafile, neither it maintains any hidden metadata in the subfiles. The sum of sizes of all local subfiles is exactly the size of the logical parallel file.

The generic functions for read and write accesses to a file are respectively *mreadc* and *mwritec*. These functions have four arguments. The first argument is the previously opened parallel file descriptor, the second argument is the offset in the global logical file, the third argument is the buffer and the forth argument is its size in bytes. The multiple I/O request specification interface allows an application program to specify multiple I/O requests within one call. This permits the library to carry out additional optimizations which otherwise would not be possible. The multiple I/O request operations are *mreadb* and *mwriteb*. See Appendix B for the full list of SFIO API functions.

The following C source code shows a simple SFIO example. The striped file with a stripe unit size of 5 bytes consists of two subfiles. It is assumed that the program is launched with one computing MPI process. A single compute node opens a striped file with two subfiles */tmp/a1.dat* at *t0-p1* and */tmp/a2.dat* at *t0-p2*. Then it writes a message “*Hello World*” and closes the global file.

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

Below is an example of multiple compute nodes simultaneously accessing the same striped file. We assume that the program is launched with three compute nodes and two I/O MPI processes. The global striped file consisting of two sub-files has a stripe unit size of 5 bytes. 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();
    //Collective open operation
    f=mopen("t0-p1,/tmp/a.dat;t0-p2,/tmp/a.dat;", 5);
```

```
    //each process writes 13 characters at its own position
    mwritec(f,13*r,bu,13);
    mclose(f);  //Collective close operation
}
```

In MPI, the function *rank* returns to each compute process its unique identifier (0, 1 and 2 in this example). Thus each compute processor running the same MPI program can follow its own computing scenario. In the above example, the compute nodes use their ranks to write at their respective (different) locations in the global file. After the writing to the parallel file is completed in the above example, the global file contains the text combined from the fragments written by the first, second and third compute nodes, i. e:

```
"Hello*World!*Hello*World!*Hello*World!*"
```

The text is distributed across the two subfiles such that the first subfile contains:

```
"Hellod!*Heorld!o*Wor"
```

And the second subfile contains (see Figure 11):

```
"*Worllo*W*Hellld!*"
```

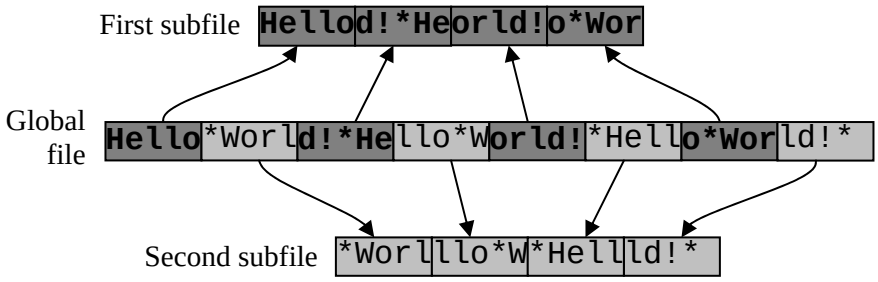

Figure 11. Distribution of a striped file across subfiles

The SFIO call *mclose* is a collective operation and is a global synchronization point for all three computing processes of the example.

Section 2.6. Optimization principles

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 for computation purposes only across the compute nodes. 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 holds the requests in the cache of the compute nodes queuing the requests individually for each I/O node. The library then tries to minimize the cost of disk accesses and network communications by preparing new aggregated requests taking care of

overlapped requests and their order. Transmission of the requests and data chunks is followed by confirmation replies sent by I/O listeners to the compute node.

Optimizations of network communications and the disk accesses on the remote I/O node are implemented on the compute node. Requests queued for each I/O node are sorted according to their offsets on the remote disk subfile. Then all overlapping or consecutive I/O requests hold in the cache are combined, and a new optimized set of requests is formed (Figure 12). This also creates new fragmentation patters in the memory of the computing processes.

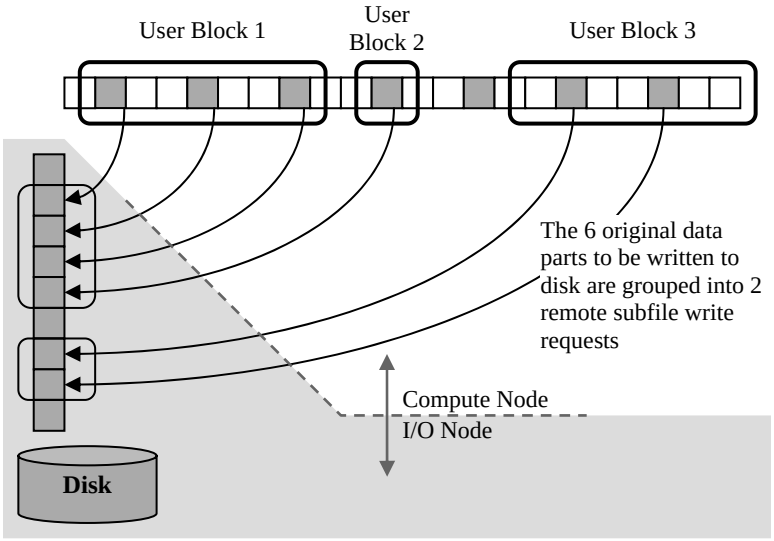

Figure 12. Disk access optimization

Optimized remote I/O node requests are kept in the cache of the compute nodes. They are launched either at the end of the SFIO function call or when the compute node estimates that the buffer size reserved on the remote I/O listener for data reception may not be sufficient. Memory is not a problem on the compute node, since data always remains in the 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 on the fly derived datatypes pointing to the fragmented memory patterns in user space associated to each of the I/O nodes. Thanks to these dynamically created derived datatypes, the data is transmitted to or from each I/O node in a single stream without additional copies. The I/O listener also receives or transmits the data as a contiguous chunk. Once the optimized data exchange pattern is established between the memory of a compute node and the remote I/O nodes, the corresponding local disk access operations are triggered by read/write instructions received at the I/O node through the MPI transport.

The efficiency of these optimizations is especially emphasized when dealing with the low stripe unit sizes. Figure 13 shows a comparison of the generic (un-optimized) write operation with its optimized counterpart.

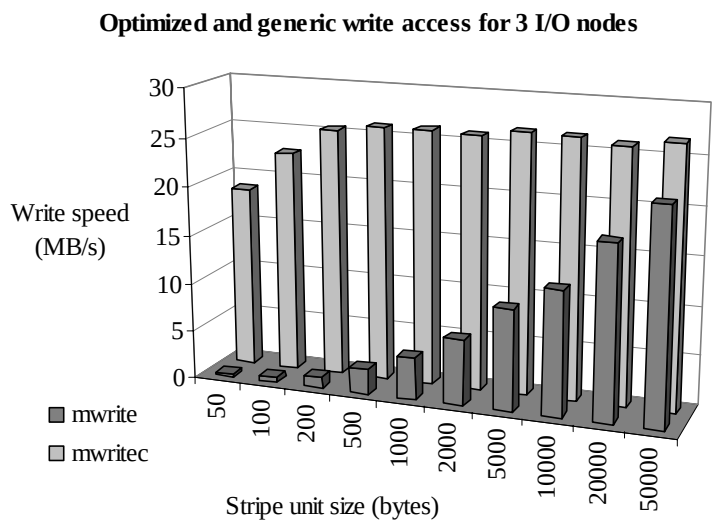

Figure 13. Comparison of the optimized write access with a generic write access on the scale of the file striping granularity (3 I/O nodes, 1 compute node, global file size is 660 Mbytes)

The performance gain achieved with multi-block access operations, thanks to the relevant network optimizations is presented in Figure 14. Support of the multi-block interface permits to fully benefit from the optimization subsystem.

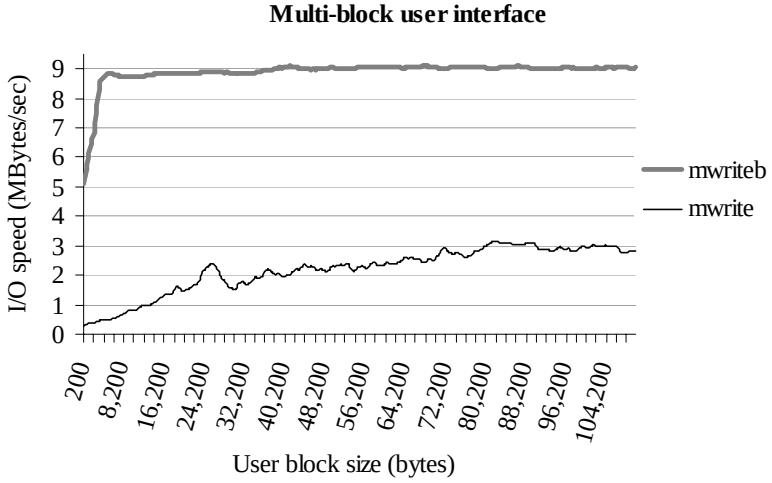

Figure 14. Comparison of the optimized multi-block write access with a generic write access on the scale of the user memory fragmentation (Fast Ethernet, stripe unit size is 1005 bytes)

Section 2.7. Functional architecture and implementation

In this section we describe the implementation of the access functions and the functional architecture of the underlying optimization methods.

An overall diagram of the implementation of the SFIO access function is shown in Figure 15. On the top of the diagram we have the application's interface to data access operations and at the bottom, 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.

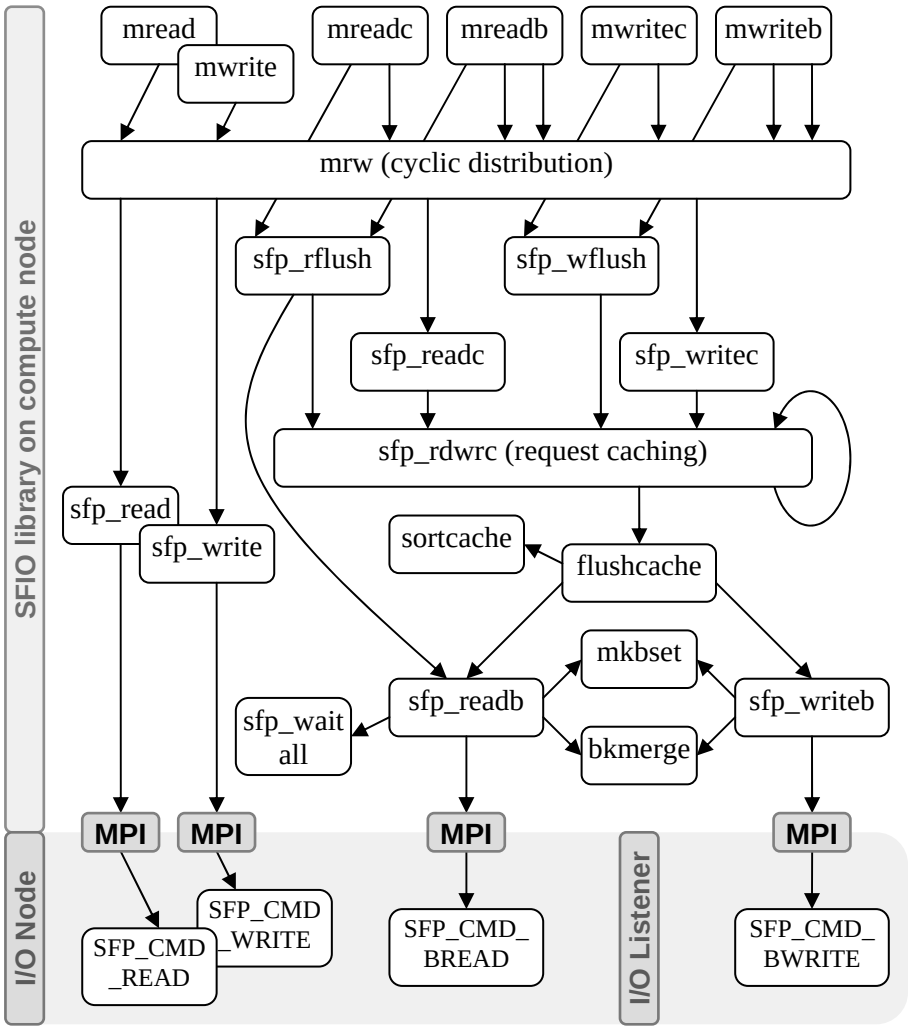

Figure 15. SFIO functional architecture

All the *mread*, *mwrite*, *mreadc*, *mwritec*, *mreadb*, *mwriteb* file access 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* operations) are split into small fragments according to the striping factor. The small requests generated by the *mrw* module contain information on 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.

Thus, 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 sub-requests to the I/O nodes using MPI as the transport layer. The rest of the diagram (the right half) is dedicated to optimized operations.

The network communication and disk access optimization is represented by the hierarchy below the *mreadc*, *mwritec*, *mreadb*, *mwriteb* access functions. For these optimized operations the *mrw* module routes the requests to the *sfp_readc* and *sfp_writec* functions. These functions access the *sfp_rdwrc* module which stores the sub-requests into a two-dimensional cache (2D cache). The 2D cache structure comprises as one dimension the I/O nodes and as a second dimension the set of subfiles each I/O node is dealing with. Often, on each I/O node there is 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* function makes sure that all generated requests fit within the buffers of the remote I/O nodes. The entries to be flushed are passed to the *flushcache* operation that also frees the corresponding resources within the 2D cache.

When the *flushcache* operation is invoked, typically a large list of the sub-requests is already been collected and needs to be processed. At this point 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 avoiding costly copies from the user memory to the system memory. 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 on the fly 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 in a single contiguous stream. The data can be transmitted or received without any memory copy at the application or library level. If a zero-copy MPI implementation, relying on hardware Direct Memory Access (DMA), is used, the entire process becomes copy-less, such that the actual data (even if fragmented) is transmitted directly from the user space to the network.

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

Section 2.8. SFIO performance

In this section we 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 [Kuonen99A]. 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, TNET and Fast Ethernet.

To have an idea about the Fast Ethernet network capabilities, throughput as a function of number of nodes is measured by a simple MPI program. The nodes are equally divided into transmitting and receiving nodes and an all-to-all traffic of relatively large blocks is generated. Figure 16 demonstrates the cluster's communication throughput scalability over Fast Ethernet. The Fast Ethernet network of T1 consists of a full crossbar switch and Figure 16 exhibits the corresponding linear scaling. Each pair of nodes (one receiver and one sender) is contributing to the overall throughput a capacity of a single link.

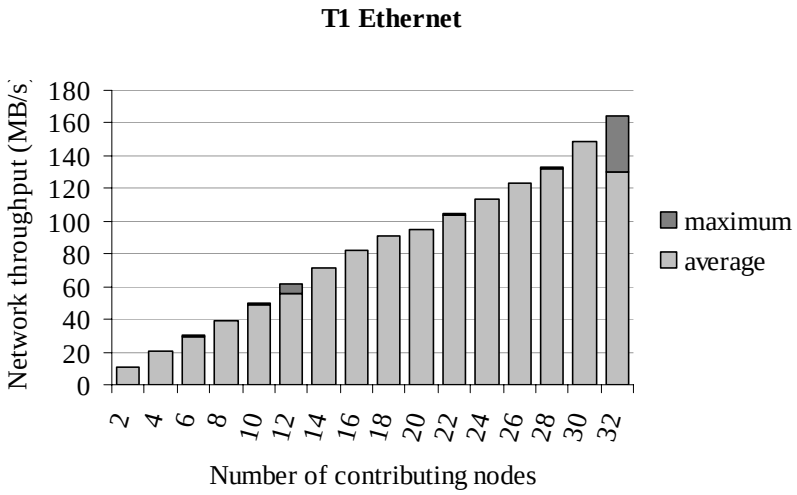

Figure 16. Aggregate throughput of Fast Ethernet as a function of the number of the contributing nodes

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 com¬pute node to a compute process and the second processor to an I/O listener (Figure 17).

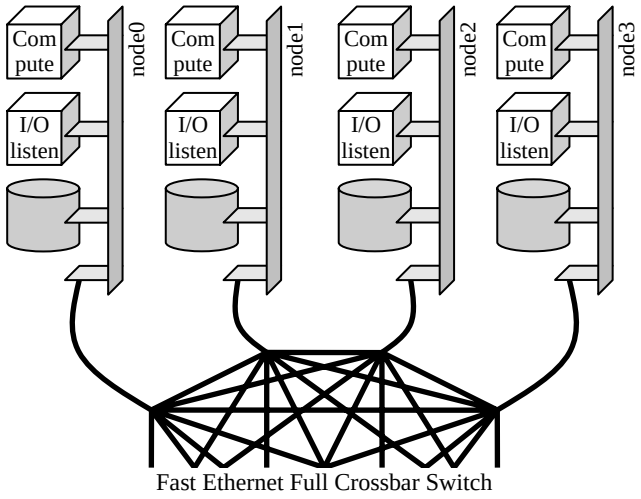

Figure 17. SFIO architecture on Swiss-T1

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. The size of the striped file is 2Gbyte and the striped unit size is 200 bytes only. The application's SFIO performance as a function of the number of compute and I/O nodes is measured for the Fast Ethernet network. It is presented in Figure 18. The white graph represents the aver¬age throughput and the gray graph the peak performance. The fall of the performance may be possi¬bly due to a non-efficient implementation of data intensive collective operations in the version of MPICH that we used (2001).

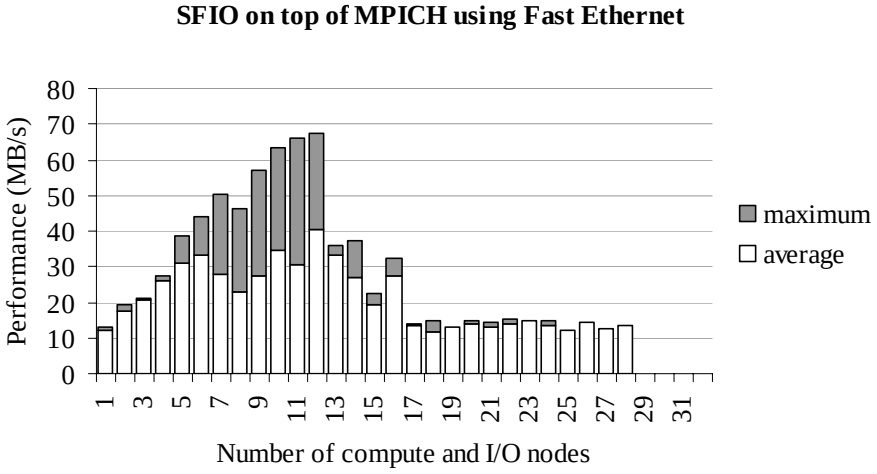

Figure 18. SFIO/MPICH all-to-all I/O performance for a 200 bytes stripe size

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) [Brauss99B]. A high performance MPI implementation called MPI/FCI is available for communication through TNET [Brauss99B].

The Swiss-T1's TNET network [Kuonen99B] consists of eight 12-port full crossbar switches (Figure 20). The gray arrows in the figure indicate the static routing between switches that do not have direct connectivity [Kuonen99A]. The topology together with the routing information defines the network's peak collective throughput over the subset of processors assigned to a given application.

The TNET throughput as a function of the number of nodes is measured by a simple MPI program. The contributing nodes are equally divided into transmitting and receiving nodes (Figure 19). Due to TNET's specific network topology (Figure 20), 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.

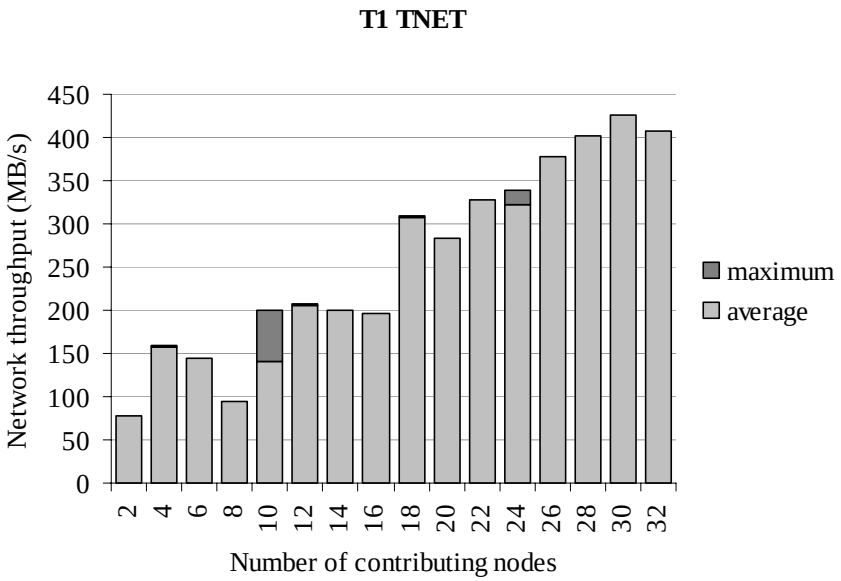

Figure 19. Aggregate throughput of TNET as a function of the number of the contributing nodes

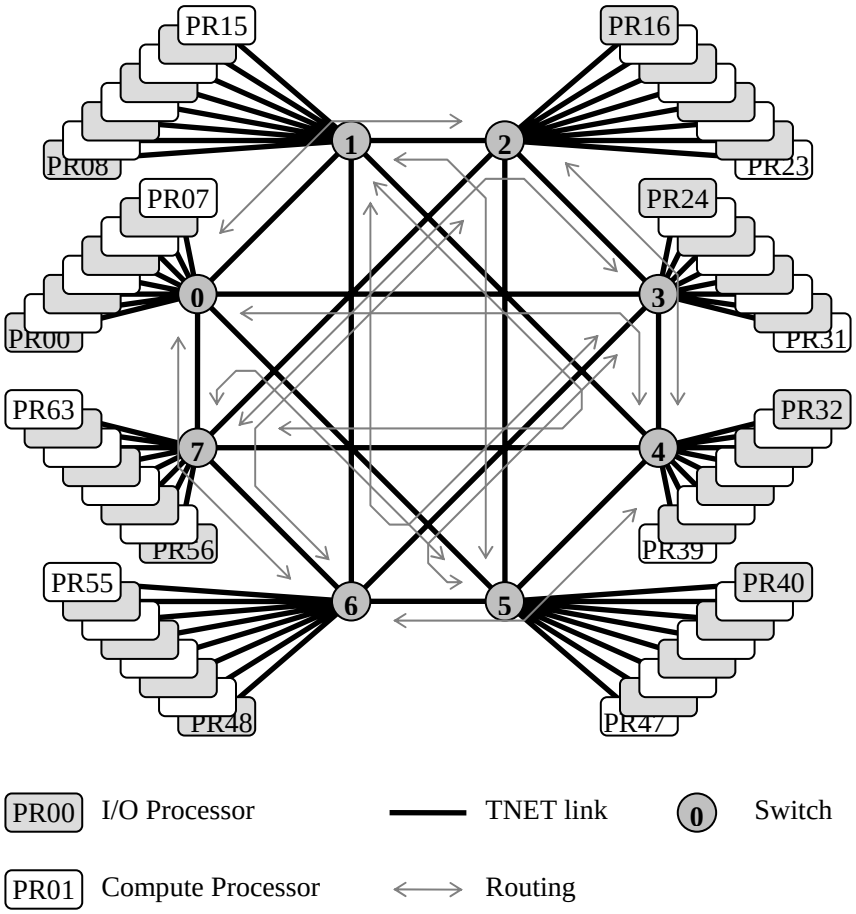

Figure 20. The Swiss-T1 network interconnection topology

The performances of the SFIO library relying on MPI/FCI using the proprietary TNET network of the Swiss-T1 supercomputer is measured according to an allocation of I/O and compute nodes identical to that of Figure 17. 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 as 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 (Figure 21). For each configuration, 53 measurements are carried out. At job launch time, pairs of I/O and compute processes are assigned randomly to processing nodes.

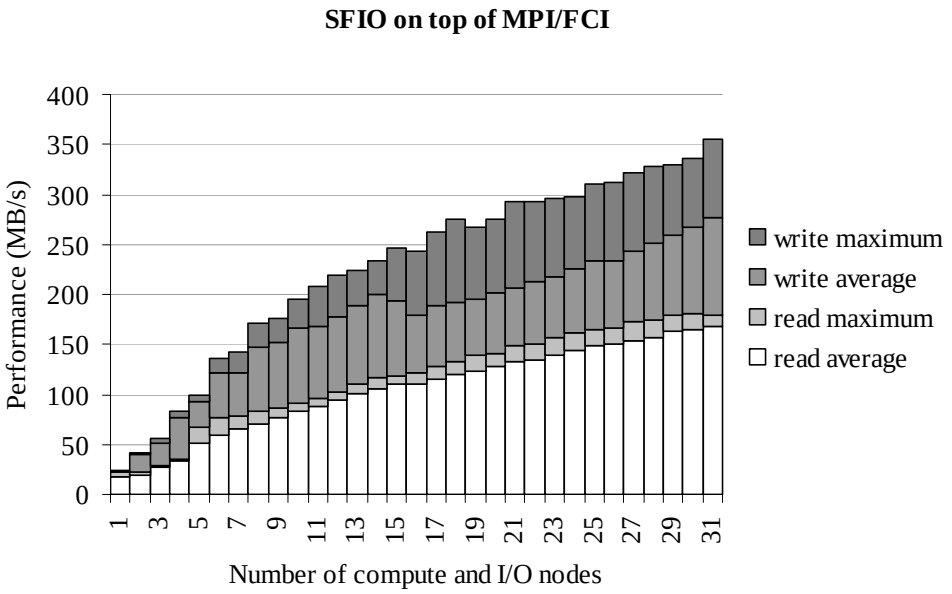

Figure 21. SFIO all-to-all I/O performance on TNET

The I/O throughput on MPI/FCI scales well when increasing the number of nodes. This configuration a stress test of SFIO system at extreme conditions in terms of the number of I/O nodes (scalability), the number of compute nodes (resistance to simultaneous concurrent access) and the extremely low stripe unit size (efficient optimizations of communication and disk access).

The speed-up may vary due to the communication topology of the TNET network (Figure 20) associated with the particular node allocation scheme. The effect of topology on I/O performance is studied in Chapter 3. It turns out that after the half of the cluster nodes are allocated the network topology becomes a major bottleneck, if the network transmissions are not properly coordinated and scheduled.

Section 2.9. MPI-I/O implementation on top of SFIO

Typical scientific applications make a large number of small I/O requests. A typical example is access to columns or blocks of out of core matrices resulting in a large number of highly fragmented non contiguous requests. MPI's derived datatypes provide the functionality for dealing with fragmented data in memory.

Most parallel file systems (at the time of the design of SFIO) allowed a user to access only a single, contiguous chunk of data at a time from a file. Noncontiguous data sets must therefore be accessed by making separate function calls to access each individual contiguous piece.

With such an interface, the file system cannot easily detect the overall access pattern. Consequently, the file system is constrained in the optimizations it can perform. To overcome the performance and portability limitations of existing parallel-I/O interfaces, the MPI Forum defined a new interface for parallel I/O as part of the MPI-2 standard [MPI2-97] referred as MPI-IO

interface. It is a rich interface with many features designed specifically for performance and portability. Some of the features are the support for noncontiguous accesses, non-blocking I/O, and a standard data representation.

The MPI-I/O interface design allows the underlying parallel I/O subsystem to optimize access operations. This is however possible only if the underlying I/O subsystem (on which MPI-I/O interface is to be implemented) supports and optimizes multi-block access requests.

Thanks to the optimizations of multi-block access in SFIO, an implementation of MPI-I/O on top of SFIO can be both efficient and will benefit from the advanced features of the MPI-I/O design.

For specifying fragmentation patterns for different purposes, MPI-I/O interface does not use arrays or vectors of locations and sizes. The fragmentation both in the memory and in the file is specified by derived datatype objects.

In MPI-I/O the file view is a global concept, which influences all data access operations. Each process obtains its own view of the shared data file. In order to specify the file view the user creates a derived datatype. Since each memory access operation may use another derived datatype that specifies the fragmentation in memory, there are two orthogonal aspects to data access: the fragmentation in memory and the fragmentation of the file view (see Figure 22). This figure presents four fragmentation scenarios from the perspective of one computing MPI process. The file view pattern can be different from one process to another.

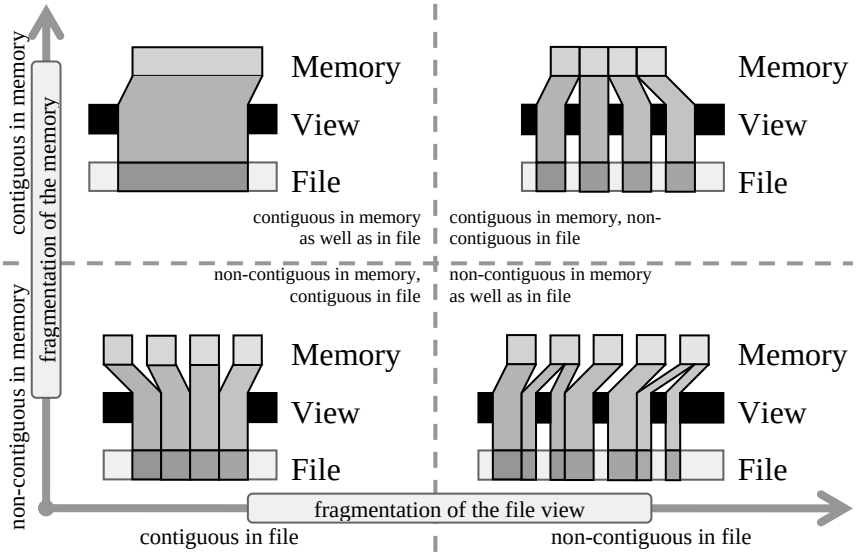

Figure 22. The use of derived datatypes in MPI-I/O interface

MPI-1 provides recursive techniques for creating datatype objects having an arbitrary data layout in memory (see Figure 23). A derived opaque datatype object can be used in various MPI operations (e.g. communication between compute nodes). The main obstacle for implementation

of a portable MPI-I/O interface is that the derived datatypes are opaque objects. Once created by the user they cannot be decoded.

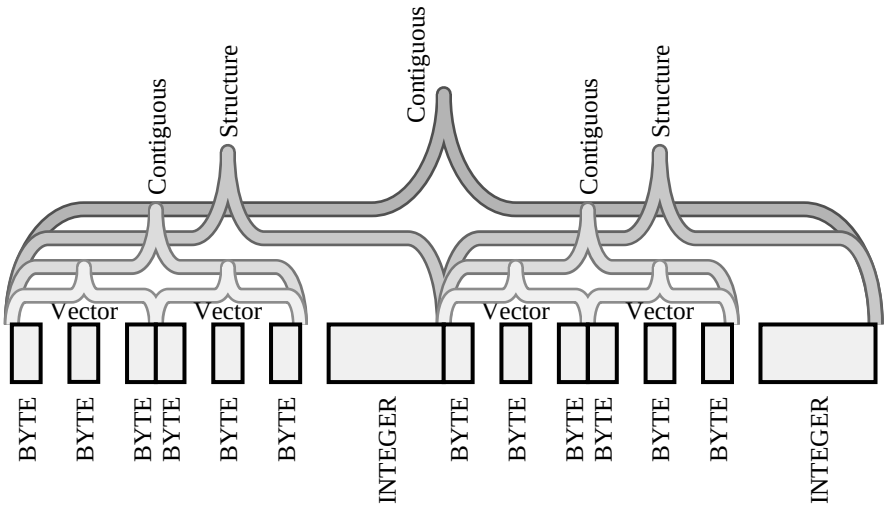

Figure 23. The recursive construction of derived datatypes in MPI ("Contiguous" is a derived datatype obtained by joining a repeated number of times another datatype, which in its turn can be fragmented)

To implement an MPI-I/O interface we need to access the flattened fragmentation pattern of a datatype created by a user. The difficulty is that the layout information, once encapsulated in a derived datatype, can not be extracted with standard MPI-1 functions. While the standard MPI-1 interface provides complete functionality for creating derived datatypes, once they are created the information cannot be retrieved back from these opaque objects with standard MPI-1 operations (see Figure 24).

A solution for figuring out the flattened fragmentation patterns (in the memory and in the file) could be to understand in each particular MPI-1 implementation the internal structure of the derived datatypes created by the user (see Figure 24). The disadvantage is that *(1)* only the operations for constructing the derived datatypes are standardized and the internal implementation of the opaque datatype objects can significantly vary from one implementation of MPI-1 to another and *(2)* the source code of a particular MPI-1 implementation is often not available or undergo to frequent updates by the vendor. Our objective is to design a portable implementation-independent solution for MPI-I/O running on top of any MPI-1 implementation.

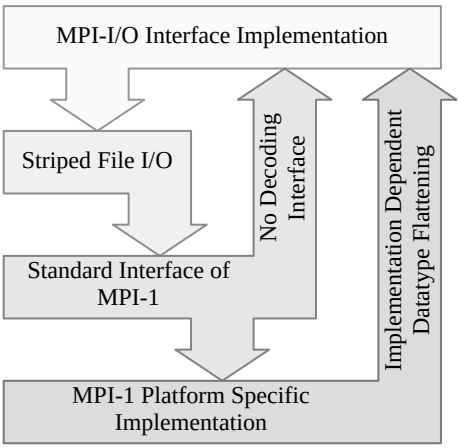

Figure 24. MPI-I/O implementation requires a method for retrieving the fragmentation patterns of opaque MPI derived datatypes

Our method relies on reverse engineering technique for discovering the flattened pattern of a user-created derived datatype.

Extension of the derived datatype is the size of the minimal contiguous space fitting the fragmented pattern of the derived datatype. The size of the derived datatype is the sum of sizes of all contributing contiguous pieces of the datatype. Standard MPI-1 provides functions for retrieving both, the extension and the size of a derived datatype.

Derived datatypes can be used in many MPI operations. A typical MPI receive operation, called *MPI_Recv*, receives a contiguous network stream and distributes it in memory according to the data layout of the datatype. If the memory is previously initialized with a “gray color”, and the network stream has a “black color”, then analysis of the memory after data reception will give us the necessary information on the data layout. Instead of sending and receiving, it is possible to use the *MPI_Unpack* standard MPI-1 operation for carrying this procedure in a single compute node. The operation *MPI_Unpack* reads from a contiguous memory block having a size equal to the size of a single unit of a derived datatype and writes to a contiguous block having a size equal to the extension of that derived datatype (see Figure 25).

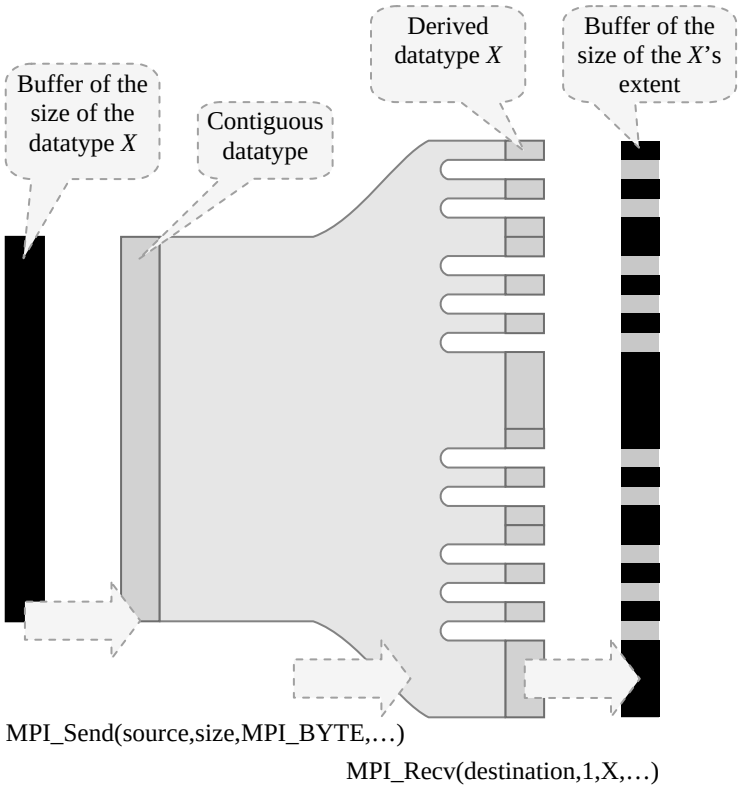

Figure 25. A reverse engineering method for discovery the fragmentation pattern of an opaque datatype built by the user

Typically the derived datatypes are used as repetition units to describe fragmentation pattern over large spaces. Decoding of only one unit is sufficient to discover the pattern. Once derived datatype is decoded its vector map is associated with the MPI opaque object for all further reuses.

With the technique for derived datatype decoding, it becomes possible to create an isolated MPI-I/O solution on top of any standard MPI-1. The Argonne National Laboratory's (ANL) MPICH implementation of MPI-I/O is used with our datatype reverse engineering technique and a subset of MPI-I/O operations has been implemented (Figure 26).

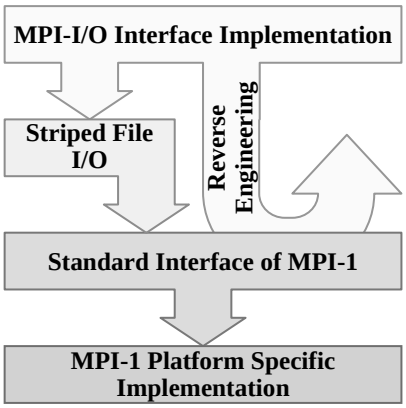

Figure 26. Isolated implementation of a portable MPI-I/O interface functional on any MPI-1 implementation

Isolated MPI-I/O package automatically gives to every MPI-1 owner MPI-I/O facilities, without any requiring to change or modify the current MPI-1 implementation.

Section 2.10. Conclusions and the recent developments in Parallel I/O

For cluster computing, SFIO is a cheap alternative to specialized dedicated I/O hardwires. It is a light-weight portable parallel I/O system for MPI programmers.

Since the design of SFIO, there were additional developments of parallel I/O. The impact of the underlying network topology and the allocation scheme of the I/O and compute nodes is studied in [Wu05A]. Further performance optimizations were achieved by scheduling of I/O access requests, taking into account the global knowledge in the case of off-line access requests and using pre-fetching relying on the predictions and estimations in the case of on-line access requests [Abawajy03], [Kallahalla02]. There is an implementation suggesting to increase the overall performance of collective read access operations not only by striping but also by simple replication of data across several I/O nodes [Wu05B] and [Liu03]. Replication and caching at I/O nodes requires a careful sequencing of all I/O operations in order to maintain the consistency of replicated copies and of a global parallel file from the perspective of all compute nodes. Relying on the file locking mechanisms may add a significant performance drawback. Moreover file locking is not always implemented in large scale systems. Several methods were proposed for allowing replications at I/O nodes and caching at compute nodes by maintaining the consistency of the global file by relying on orthogonal MPI level communications between compute nodes without using file locking mechanisms [Wu05B], [Coloma04]. There are implementations suggesting parallel communications not only between a compute node and different I/O nodes but also between a compute node and each individual I/O node. By doing this a greater network throughput performance can be achieved [Liu03] and [Ali05]. The author of [Ali05] reported an overall throughput of 291 Mbps with 18 compute and I/O processors. The throughput of SFIO from 150 to 350 Mbps with 31 compute and I/O nodes still remains competitive. In terms of the developments of parallel I/O interfaces, portable implementations of

MPI-I/O interface have been released [Thakur99B], [Baer04]. The fine granularity with the resulting high level of load balance remains the strong point of SFIO, whose underlying optimizations allows as small as a 75-byte stripe size with only negligible loss in performance. Usually the parallel I/O systems are optimized for striping unit sizes not smaller than a few kilobytes [Thakur99B]. For a balanced I/O workload in the servers an alternative suggestion for dynamically adapting striping factors and dynamic data distribution was suggested [Ma03B].

Chapter 3. Liquid scheduling of parallel transmissions in coarse-grained low-latency networks

Chapter 4. Capillary routing: parallel multi-path routing for fault-tolerant real-time communications in fine-grain packet-switching networks

In off-line streaming, packet level erasure resilient Forward Error Correction (FEC) codes rely on the unrestricted buffering time at the receiver. In real-time streaming, the extremely short playback buffering time makes FEC inefficient for protecting a single path communication against long link failures. It has been shown that one alternative path added to a single path route makes packet level FEC applicable even when the buffering time is limited. Further path diversity, however, increases the number of underlying links increasing the total link failure rate, requiring from the sender possibly more FEC packets. We introduce a scalar coefficient for rating a multi-path routing topology of any complexity. It is called Redundancy Overall Requirement (ROR) and is proportional to the total number of adaptive FEC packets required for protection of the communication. With the capillary routing algorithm, introduced in this chapter we build thousands of multi-path routing patterns. By computing their ROR coefficients, we show that contrary to the expectations the overall requirement in FEC codes is reduced when the further diversity of dual-path routing is achieved by the capillary routing algorithm.

Section 4.1. Introduction

Packetized IP communication behaves like an erasure channel. Information is chopped into packets, and each packet is either received without error or not received. Packet level erasure resilient FEC codes can mitigate packet losses by adding redundant packets, usually of the same size as the source packets.

In off-line streaming erasure resilient codes achieve extremely high reliability in many challenging network conditions [MacKay05]. For example, it is possible to deliver voluminous files (e.g. recurrent updates of GPS maps) via satellite broadcast channel without feed-backs to millions of motor vehicles under conditions of fragmental visibility (see [Honda04] and Raptor codes [Shokrollahi04]). In the film industry, the day's film footage can be delivered from the location it has been shot to the studio that is many thousands of miles away not via FedEx or DHL, but over the lossy internet even with long propagation delays (see [Hollywood03] and LT codes [Luby02]). Third Generation Partnership Project (3GPP), recently adopted Raptor [Shokrollahi04] as a mandatory code in Multimedia Broadcast/Multicast Service (MBMS). The benefit of off-line streaming from application of FEC relies on time diversity, i.e. on the receiver's right to not forward immediately to the user the received information. Long buffering is not a concern, the receiver can unrestrictedly hold the received packets, and as a result packets representing the same information can be collected at very distant periods of time.

In real-time single-path streaming FEC can only mitigate short failures of fine granularity. See [Choi06] using RS(24,20) packet level code with 20 source packets and 4 redundant packets or also [Johansson02], [Huang05], [Padhye00] and [Altman01]. Due to restricted playback buffering time, packets representing the same information cannot be collected at very distant periods of time. Instead of relying on time-diversity FEC in real-time streaming can rely on path-diversity. Recent publications show the applicability of FEC in real-time streaming with dual-path routes. Author of [Qu04] shows that strong FEC sensibly improves video communication following two disjoint paths and that in two correlated paths weak FEC codes are still advantageous. [Tawan04] proposes adaptive multi-path routing for Mobile Ad-Hoc Networks (MANET) addressing the load balance and capacity issues, but mentioning also the potential advantages for FEC. Authors of [Ma03A] and [Ma04] suggests replacing in MANET the link level Automatic Repeat Query (ARQ) by a link level FEC assuming regenerating nodes. Authors of [Nguyen02] and [Byers99] studied video streaming from multiple servers. The same author [Nguyen03] later studied real-time streaming over a dual-path route using a static Reed-Solomon RS(30,23) code (FEC blocks carrying 23 source packets and 7 redundant packets). [Nguyen03], similarly to [Qu04], compares dual-path scenarios with the single OSPF routing strategy and has shown clear advantages of the dual-path routing. The path diversity in all these studies is limited to either two (possibly correlated) paths or in the most general case to a sequence of parallel and serial links. Various routing topologies have so far not been regarded as a space to search for a FEC efficient pattern.

In this chapter we try to present a comparative study for various multi-path routing patterns. Single path routing is excluded from our comparisons, being considered too hostile. Steadily diversifying routing patters are built layer by layer with the *capillary routing* algorithm (Section 4.2).

In order to compare multi-path routing patterns, we introduce Redundancy Overall Requirement (ROR), a routing coefficient relying on the sender's transmission rate increases in response to individual link failures. By default, the sender is streaming the media with static FEC codes of a constant weak strength in order to tolerate a certain small packet loss rate. The packet loss rate is measured at the receiver and is constantly reported back to the sender with the opposite flow. The sender increases the FEC overhead whenever the packet loss rate is about to exceed the tolerable limit. This end-to-end adaptive FEC mechanism is implemented entirely on the end nodes, at the application level, and is not aware of the underlying routing scheme [Kang05], [Xu00], [Johansson02], [Huang05] and [Padhye00]. The overall number of transmitted adaptive redundant packets for protecting the communication session against link failures is proportional *(1)* to the usual packet transmission rate of the sender, *(2)* to the duration of the communication, *(3)* to the single link failure rate, *(4)* to the single link failure duration and *(5)* to the ROR coefficient of the underlying routing pattern followed by the communication flow. The novelty brought by ROR is that a routing topology of any complexity can be rated by a single scalar value (Section 4.3).

In Section 4.4, we present ROR coefficients of different routing layers built by the capillary routing algorithm. Network samples are obtained from a random walk MANET with several hundreds of nodes. We show that path diversity achieved by the capillary routing algorithm reduces substantially the amount of redundant FEC packets required from the sender.

Section 4.2. Capillary routing

In subsection 4.2.1 we present a simple Linear Programming (LP) method for building the layers of capillary routing. A more reliable algorithm is described in subsection 4.2.2. In subsection 4.2.3 we present the discovery of bottlenecks at each layer of capillary routing, required for construction of successive layers.

4.2.1. Basic construction

Capillary routing can be constructed by an iterative LP process transforming a single-path flow into a capillary route. First minimize the maximal value of the load of all links by minimizing an upper bound value applied to all links. The full mass of the flow will be split equally across the possible parallel routes. Find the bottleneck links of the first layer (see subsection 4.2.3) and fix their load at the found minimum. Minimize similarly the maximal load of all remaining links without the bottleneck links of the first layer. This second iteration further refines the path diversity. Find the bottleneck links of the second layer. Minimize the maximal load of all remaining links, but now without the bottlenecks of the second layer as well. Repeat this iteration until the entire communication footprint is enclosed in the bottlenecks of the constructed layers.

Figure 27, Figure 28 and Figure 29 show the first three layers of the capillary routing on a small network. The top node on the diagrams is the sender, the bottom node is the receiver and all links are oriented from top to bottom.

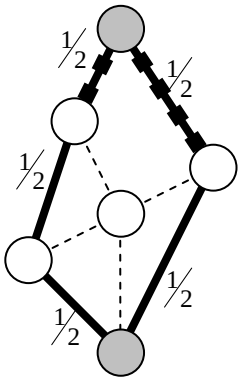

Figure 27. In the first layer the flow is equally split across two paths, two links of which, marked by thick dashes, are the bottlenecks.

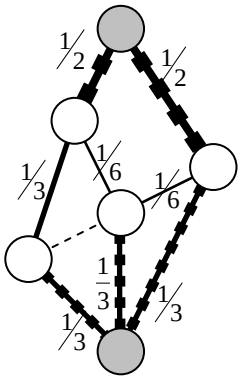

Figure 28. The second layer minimizes to 1/3 the maximal load of the remaining seven links and identifies three bottlenecks.

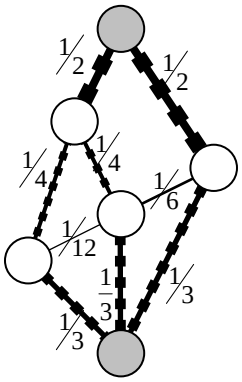

Figure 29. The third layer minimizes to 1/4 the maximal load of the remaining four links and identifies two bottlenecks.

Figure 30 shows the 10-th layer of capillary routing between a pair of end nodes on a network with 180 nodes and 1374 links. Links not carrying traffic are not shown. The solid lines of the diagram represent 55 bottleneck links belonging to one of the 10 layers. The dashed lines represent a min-cost solution of the remaining flow not enclosed in bottlenecks after the 10-th layer. There could be several tens of additional routing layers until complete capillarization is achieved.

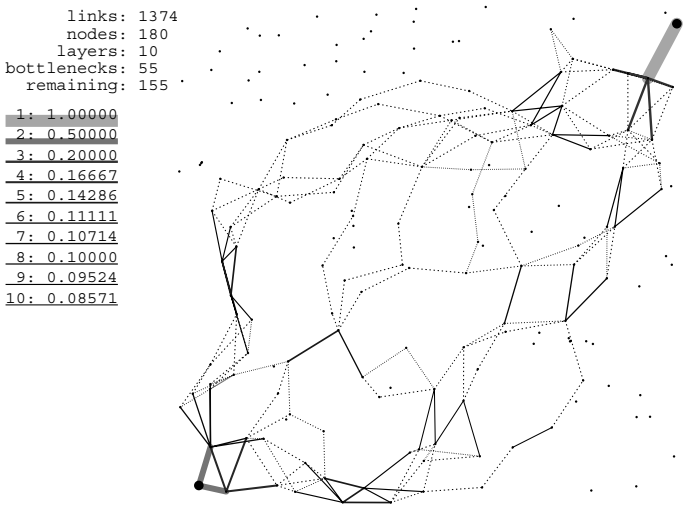

Figure 30. Routing pattern of layer 10 built by the capillary routing algorithm on a network sample with 150 nodes

4.2.2. Numerically stable version

Although the described LP process is completely valid, it is numerically instable. The precision errors propagating through the layers of capillary routing reach noticeable sizes and, when dealing with tiny loads, result in infeasible LP problems. We have found a different, stable LP method which maintains the values of parameters and variables in the same order of magnitude at all times.

Instead of decreasing the maximal value of loads of the links, the routing path is discovered by solving max flow problems defined by the flow-out coefficients at each node. Initially only the peer nodes have non-zero flow-out coefficients: +1 for the source and –1 for the sink (Figure 31 and Figure 32).

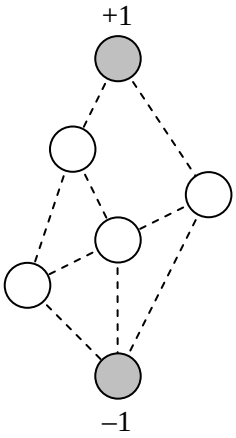

Figure 31. Initial problem with one source and one sink node

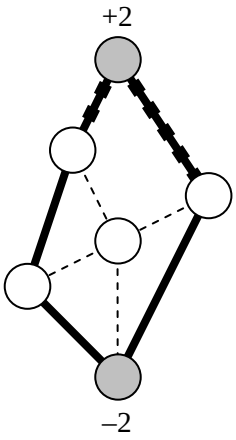

Figure 32. Maximize the flow, fix the new flow-out coefficients at the nodes and find the bottleneck links (layer 1, $F^1 = 2$)

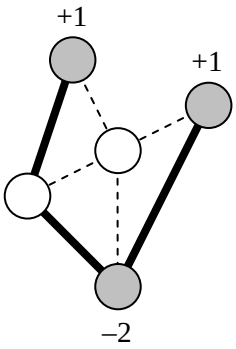

Figure 33. Remove the bottleneck links from the network and adjust the flow-out coefficients at the adjacent nodes

At each subsequent layer (Figure 33 to Figure 36) we have a bounded multi-source/multi-sink problem: a uniform flow from a set of sources to a set of sinks, where all rates of transmissions by sources and all rates of receptions by sinks increase proportionally in respect to each node's flow-out coefficient (either positive or negative). The multi-source/multi-sink problems arise since the LP problem at each successive layer is obtained by complete removal of the bottlenecks from the previous LP problem. By removing the bottlenecks we adjust correspondingly the flow-out coefficients of the adjacent nodes (to respect the flow conservation rule) and thus possibly produce new sources and sinks in the network. Except for the unicast problem of the first layer, the successive layer problems do not belong in general to the simple class of "network linear programs" [Fourer03].

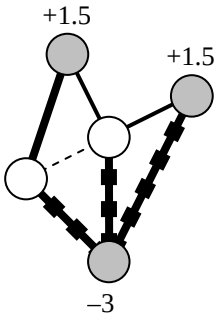

Figure 34. Maximize the flow in the new sub-problem, fix the new flow-out coefficients at the nodes and find the new bottlenecks (layer 2, $F^2 = 1.5$)

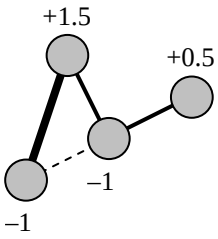

Figure 35. Again remove the bottleneck links from the network and adjust correspondingly the flow-out coefficients at the adjacent nodes

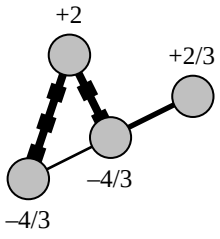

Figure 36. Maximize the flow in the obtained new problem, fixing the new resulting flow-out coefficients at the nodes and find the new bottlenecks (layer 3, $F^3 = 4/3$)

We define the bounded multi-source/multi-sink problem at layer *l* by the sets of nodes and links and by the flow-out coefficients for sources and sinks (all indexed with an upper index *l*) as follows:

- set of nodes N^l,
- set of links $(i,j) \in L^l$, where $i \in N^l$ and $j \in N^l$,
- flow-out coefficients f_i^l for all $i \in N^l$
- at layer l the max-flow solution yields the flow increase factor F^l and the set of bottlenecks B^l, where $B^l \subset L^l$

Then, the equations for computing the sets N^{l+1}, L^{l+1} and the flow-out coefficients f^{l+1} of the next layer are as follows:

$$N^{l+1} = N^l \tag{1}$$

$$L^{l+1} = L^l - B^l \tag{2}$$

$$f_j^{l+1} = f_j^l \cdot F^l \quad + \sum_{(i,j)\in B^l} (+1) \quad + \sum_{(j,k)\in B^l} (-1) \tag{3}$$

add 1 for each incoming bottleneck link (i,j) — **subtract 1 for each outgoing bottleneck link** (j,k)

After a certain number of applications of the max-flow objective with corresponding modifications of the problem, we will finally obtain a network having no source and sink nodes. At this point the iteration stops. All links followed by the flow in the capillary routing are enclosed in bottlenecks of one of the layers.

In order to restore the original proportions of the flow, the flow increases, induced by the preceding max-flow solutions must all be compensated. The true value of flow $r_{i,j}$ traversing the bottleneck link $(i,j) \in B^l$ of layer *l* is the initial single unit of flow divided by the product of the flow increase factors F^i (where $1 \le i \le l$) of the present and all preceding layers:

$$r_{i,j} = \frac{1}{\prod_{i=1}^{l} F^i} \qquad \text{where } l \text{ is the layer for which } (i,j) \in B^l \tag{4}$$

The max-flow approach proves to be very stable, because it maintains all values of variables and parameters in the same order of magnitude (even for very deep layers with tiny loads) and also because it enables us to detect and correct errors in the flow-out coefficients of the LP problem generated for the next layer of capillary routing.

In the next subsection we show how to identify bottlenecks after the max-flow solution of the capillary routing layer is found.

4.2.3. Bottleneck hunting loop

In the example of Figure 37 with three transmitting nodes and two receiving nodes, the flow can be proportionally increased at most by a factor of 4/3 and the bottleneck links are among four maximally loaded suspected links {*a*, *b*, *d*, *e*}, marked in Figure 38 by thick dashes.

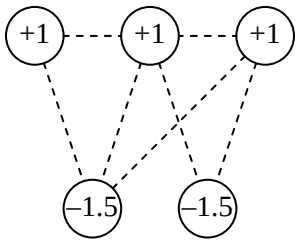

Figure 37. An example of a bounded multi-source/multi-sink problem (obtained during construction of the capillary routing from a network with one source and one destination node)

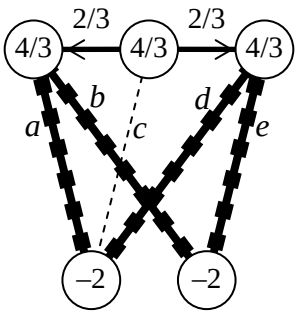

Figure 38. A max-flow solution with the flow increase factor of 4/3, containing four maximally loaded candidate links {*a, b, d, e*}

At each layer, after minimizing the maximal load of links, the bottlenecks of the layer are discovered in a bottleneck hunting loop. At each iteration of the hunting loop, we minimize the load of the traffic over all links having maximal load and being suspected as bottlenecks. Links not maintaining their load at the maximum are removed from the suspect list. The bottleneck hunting loop stops if there are no more links to remove.

In the example of Figure 38 the sum of loads of all four suspected links can be minimized (by an LP objective) to 3 (see Figure 39). Now only three links {*a*, *b*, *e*}, marked by thick dashes, continue to maintain the maximal load. The sum of loads of three remaining suspected links can be further reduced to 2 (see Figure 40). These two remaining links {*b*, *e*}, marked by thick dashes, maintained the maximal load at all times and are the true bottleneck links since the sum of their loads cannot be further reduced.

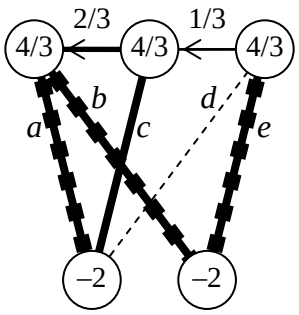

Figure 39. Cost reduction applied to four fully loaded links of Figure 38 reduces the load of suspected link *d*, and the suspect list is now {*a, b, e*}.

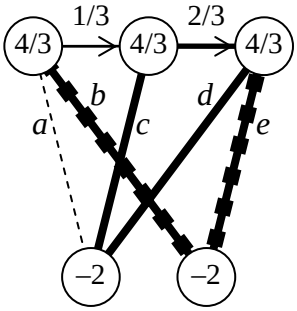

Figure 40. Cost reduction applied to the three fully loaded links of Figure 39 reduces the load of another suspected link *a*, and the true bottleneck links are {*b, e*}.

In this example the two bottlenecks are found in two iterations.

For capillary routing layers built simultaneously on 200 independent network samples each with 300 nodes (in average 2,555.7 links per network), Figure 41 shows the decrease in the number of suspected links during the bottleneck hunting loop of each capillary routing layer from 1 to 10.

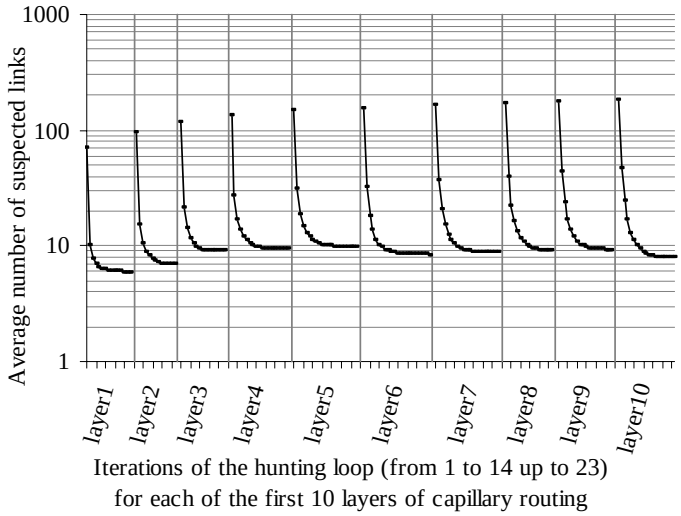

Figure 41. Decrease of the number of suspected links during the bottleneck hunting loop of each of 10 capillary routing layers

At the end of each hunting loop (from 14 to 23 iterations) the suspect list consists of only true bottleneck links, in average between 5.9 and 9.9 bottlenecks per network.

Section 4.3. Redundancy Overall Requirement (ROR)

The definition and equations of ROR are given in subsection 4.3.1. Computation of transmission FEC block size as a function of the packet loss rate *p* is presented in subsection 4.3.2. Equation of ROR for a particular case of very large FEC blocks is presented in subsection 4.3.3.

4.3.1. Definition of ROR

We assume a combination of a small static tolerance of the media stream to weak failures, with a dynamically added adaptive FEC for combating serious failures exceeding the tolerable packet loss rate.

For a given routing pattern, ROR is defined as the sum of all transmission rate overheads required from the sender for combating each non-simultaneous link failure in the route. For example, if the communication footprint consists of five links, and in response to each individual link failure the sender increases the packet transmission rate by 25%, then the ROR coefficient will be equal to the sum of these five FEC transmission rate increases, i.e. $ROR = 5 \cdot 25\% = 1.25$. If P is the usual packet transmission rate and P_l is the increased rate of the sender, responding to the failure of a link $l \in L$, where L is the set of all links, then:

$$ROR = \sum_{l \in L} \left(\frac{P_l}{P} - 1 \right) \tag{5}$$

Let us consider a long communication, and let D be the total failure time of a single network link during the whole duration of the communication. D is the product of the average duration of a single link failure, the frequency of a single link failure and the total communication time. According to equation (5):

$$D \cdot P \cdot ROR = D \cdot P \cdot \sum_{l \in L} \left(\frac{P_l}{P} - 1 \right) \tag{6}$$

$$= \sum_{l \in L} \left(D \cdot P_l - D \cdot P \right) \tag{7}$$

Assuming one single link failure at a time and a uniform probability and duration of link failures, according to equation (7), $D \cdot P \cdot ROR$ is the number of adaptive redundant packets that the sender actually needs to transmit in order to compensate for all network failures occurring during the total communication time. Therefore ROR is a routing coefficient for computing the overall number of required redundant packets.

Redundant packets are injected into the original media stream for every block of M source packets. During streaming, M is supposed to stay constant. However, the number of redundant packets for each block of M media packets is variable, depending on the conditions of the erasure channel. The M source packets with their related redundant packets form a FEC block. By FEC_p we denote the FEC block size chosen by the sender in response to a packet loss rate p. We assume that by default the media is streamed in FEC blocks of length of FEC_t such that the flow has a static tolerance to weak losses $0 \le t < 1$. When the loss rate p measured at the receiver is about to exceed the tolerable limit t, the sender increases its transmission rate by injecting additional redundant packets.

The random packet loss rate, observed at the receiver during the failure time of a link in the communication path, is the portion of the traffic still being routed toward the faulty link. Thus, a complete failure of a link l carrying a relative traffic load of $0 \le r(l) \le 1$ according to the routing pattern, produces at the receiver a packet loss rate equal to the same relative traffic load $r(l)$.

Equation (5) for ROR can thus be re-written as follows:

$$ROR = \sum_{l \in L \mid t \le r(l) < 1} \left(\frac{FEC_{r(l)}}{FEC_t} - 1 \right) \tag{8}$$

a sum over all links carrying a flow exceeding the tolerable loss limit

The links carrying the entire traffic are skipped in the sum index of equation (8), since the FEC required for the compensation of failures of such links is infinite. By construction (Section 4.2), none of the considered multi-path routing schemes pass their entire traffic through a non-critical single link.

4.3.2. Computing FEC block size

We compute the FEC_p function assuming a Maximum Distance Separable (MDS) code [Seroussi86], [Schwarz02]. With an MDS code we can successfully decode the *M* source packets if we receive any *M* packets of the transmission FEC block.

In order to collect a mean of *M* packets at the receiver under random loss rate *p*, $M/(1-p)$ packets must be transmitted at the sender. However the probability of receiving $M-1$ packets or $M-2$ packets (which makes the decoding impossible) remains high. In order to maintain a very low probability δ of receiving less than *M* packets, we must send many more redundant packets in the block than is necessary to receive an average of *M* packets at the receiver side. We must fix the acceptable Decoding Error Rate (DER), such that $\delta \le DER$, in order to compute the $FEC_p \ge M$ function.

The probability $P_n(n|N)$ of having exactly *n* losses (erasures) in a block of *N* packets with a random loss probability *p* is computed according to the binomial distribution:

$$P_p(n|N) = \binom{N}{n} \cdot p^n \cdot q^{N-n} \tag{9}$$

where $\binom{N}{n} = \frac{N!}{n! \cdot (N-n)!}$ **and** $q = 1-p$

The probability of having $N-M+1$ or more losses, i.e. the decoding failure probability, is computed as follows:

$$\delta = \sum_{n=N-M+1}^{N} \binom{N}{n} \cdot p^n \cdot q^{N-n} \tag{10}$$

Therefore for computing the carrier block's minimal length for a satisfactory communication (i.e. FEC_p function), it is sufficient to steadily increase the block length *N* until the desired decoding error rate (DER) is met.

FEC_p functions divided by *M* (i.e. transmission rate increase factors FEC_p/M) are bounded above by $\log_p(DER)$ when $M=1$ and below by $1/(1-p)$ when $M \to \infty$ (for packet loss rates much larger a very small DER). The higher the number of media packets in the block the closer the transmission rate increase can approach the lowest theoretical limit. For *M* from 1 to 10 these transmission rate increase factors are plotted in Figure 42 (for $DER = 10^{-5}$).

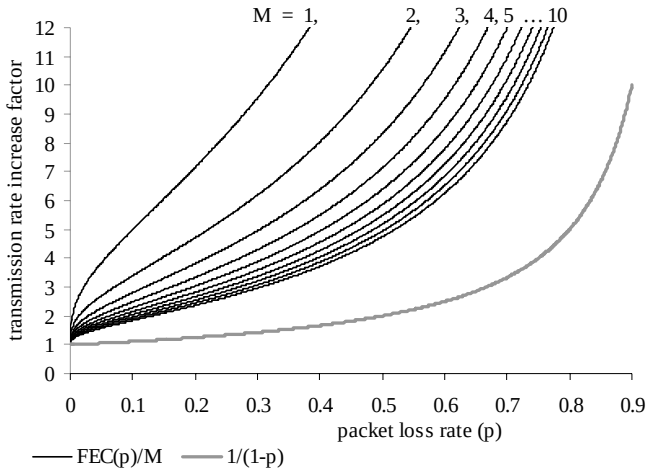

Figure 42. Transmission rate increase factor as a function from the packet loss rate ($DER = 10^{-5}$)

4.3.3. Streaming with large FEC blocks

The larger the number of media packets *M* in the FEC block, the smaller the cost of FEC overhead is, but the longer the buffering time at the receiver must be. For example VOIP with 20 ms sampling rate restricts the number of media packets *M* in a single FEC block to 20 – 25 packets.

If the playback buffering time can be a couple of minutes long, with thousands of source packets in a FEC block (for example in packetized TV) we can assume that $FEC_p = M/(1-p)$. Although for large numbers of source packets MDS codes do not exist, other capacity-approaching LDPC [MacKay96], [Richardson01] or fountain codes [MacKay05] can decode a large block of source packets requiring only a very little excess of packets (in this context this excess can be ignored).

In such case, taking into account the above assumptions and equation (8), the ROR coefficient of a multi-path routing pattern is computed according to the following equation:

$$ROR = \sum_{l \in L \mid t \le r(l) < 1} \left(\frac{1-t}{1-r(l)} - 1 \right) \qquad (11)$$

Path diversity can be required in off-line large file downloads aiming at avoiding the idle times of the last kilometer bottleneck occurring due to arbitrary failures elsewhere, within the lossy Internet. Thanks to multi-path routing, the sender with an adaptive transmission rate can feed the last kilometer bottleneck link constantly at its maximal bandwidth (see [Nguyen02] and [Byers99] for video streaming from multiple servers). In this case also, the choice of the multi-path routing pattern can be rated by equation (11). Note that according to equations (8) and (11) the ROR coefficient of a routing pattern depends also on the static tolerance *t* of the streaming media to weak failures.

Section 4.4. Redundancy Overall Requirement in capillary routing

For capillary routing layers 1 to 10, we compute the average ROR coefficients simultaneously over several networks. The network samples are drawn from timeframes of a random walk MANET. Initially the nodes are randomly distributed on a rectangular area, and then, at every timeframe, they move according to a random walk algorithm. If two nodes are close enough (and are within the coverage range) then there is a link between them. At the same time we consider also streaming media at 15 different strengths of static FEC codes which tolerate small packet loss rates from 3.6% to 7.8% respectively (with an increment of 0.3%).

Figure 43, represents a MANET with 115 nodes and 300 timeframes (each representing one network sample) divided into seven sets of network samples. For each set of samples and for each static FEC strength we plot the average ROR coefficient (over all considered network samples) as the routing layer increases. Figure 43 shows that the overall requirement in adaptive FEC packets decreases with capillarization. The ROR coefficients of the routing samples are computed assuming a short playback buffering time according to equation (8), where the FEC block size (as function of the packet loss rate p) is computed according to equation (10), the number of media packets (M) per transmission block is 20 and the desired decoding failure rate (DER) is 10^{-5}.

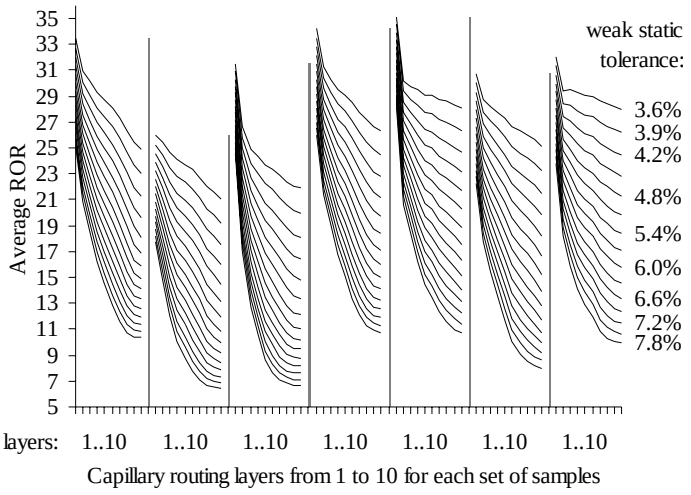

Figure 43. Average ROR as a function from the capillary routing layer

Figure 44 represents a MANET with 120 nodes and 150 timeframes divided into four sets of network samples. The upper 15 curves similarly to the curves of Figure 43 are computed according to equations (8) and (10), where $M = 20$ and $DER = 10^{-5}$. However, the lower 15 curves of Figure 44 are computed according to equation (11) for streaming with large FEC blocks.

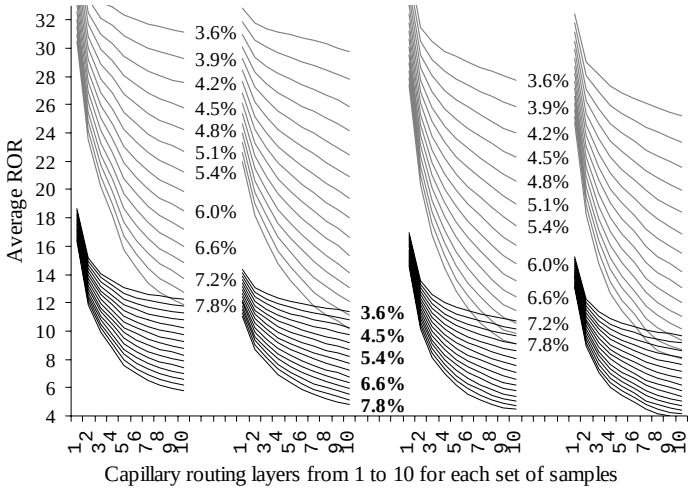

Figure 44. Average ROR computed assuming real-time streaming (the group of curves above) and off-line streaming (the group below)

When streaming with large blocks the Redundancy Overall Requirement is twice as low as in streaming with restricted playback buffering time, but the capillarization of routing is beneficiary in both cases.

Logically, the ROR curve of the media stream is shifted down as the statically added tolerance increases, but the increase of the weak static tolerance emphasizes the efficiency gain achieved by capillarization. The drawback of path diversity in general is that by forming long paths we increase the number of links in the communication footprint raising the overall failure rate and thus possibly increasing the overall requirement in FEC codes. However, Figure 43 and Figure 44 show that despite the communication footprint becomes larger; with the routing patters built by the capillary routing algorithm the requirement in redundant packets decreases noticeably most of the time.

Section 4.5. Conclusions

The reliability issues of packetized real-time streaming are of growing importance. Commercial real-time streaming applications however do not consider channel coding at the packet level as a serious solution for improving the reliability of communication. That is because in single path communications, even heavy FEC overheads cannot protect against failures lasting more than the short duration of the playback buffer. Recent studies demonstrated that path diversity makes FEC applicable for real-time streaming. By studying a wide range of routing topologies, we show that combination of channel coding with appropriate multi-path routing allows reliable real-time streaming with a low overall requirement in FEC codes.

For this purpose we introduced a layer by layer strategy for building multi-path capillary routing patterns. The first layer provides a simple multi-path solution. As the layer number increases, the underlying routing pattern relies on the network more securely. Unlike max-flow or

shortest path solutions, for a given source and destination, by construction (Section 4.2) there exists only one solution of capillary routing.

We introduced ROR coefficient, a method for rating multi-path routing patterns by a single scalar value. The ROR rating corresponds to the total redundancy overhead that the sending node must provide in order to combat the losses occurring from non-simultaneous failures of links in the communication path. Despite the fact that the spreading out of the routing results in the increase of the overall failure rate of underlying links, with capillarization the overall requirement in adaptive FEC packets decreases substantially.

Capillary routing can be applicable to multi-hop mobile wireless networks, where wireless content can be streamed to and from the user via multiple base stations; or to the public internet, where, if the physical routing cannot be accessed, an overlay network can be used [Guven04]. We hope that our investigation will provide some guidelines for future design of path diversity-based real-time streaming systems.

Appendix A. Rate of publications on parallel I/O

Parallel I/O was a hot topic in 1998. For a period from 1986 through 2006, the chart below shows the rate of IEEE publications related to parallel I/O on a relative scale.

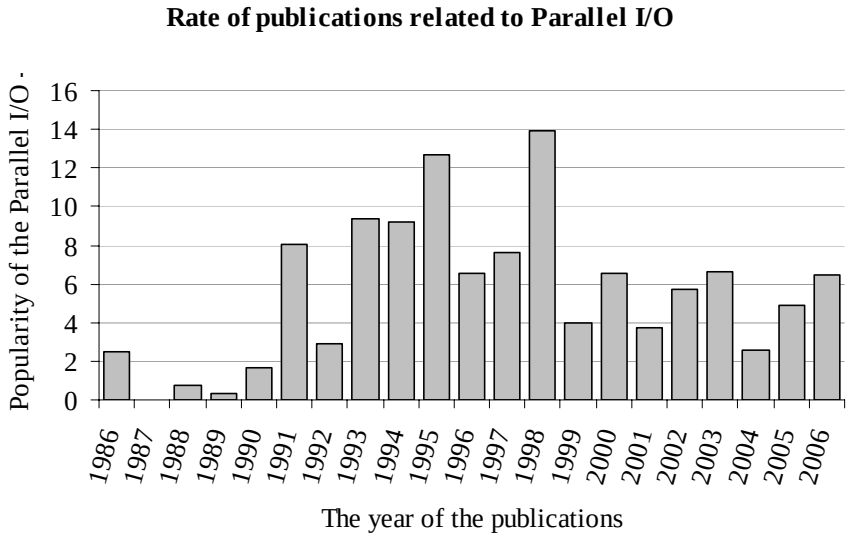

Figure 45. Yearly fractions of IEEE publications related to Parallel I/O

Appendix B. SFIO function calls

This appendix presents the API functions of the SFIO library. The SFIO interface consists of file management, data access and error management operations.

B.1. File management operations

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* and *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 changes the size of the logical file. If the specified size is smaller than the current, the operation truncates the logical file to the new size.

B.2. Data access operations

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. The functions *mreadc* and *mwritec* 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* array provides the size of each memory block in bytes.

B.3. Error management operations

Error management is provided by *merror* and its collective counterpart *merrora* functions.

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

Functions *merror* and *merrora* return an array of error statistics accumulated on all I/O nodes. At the same time, they reset the error counters at 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* local OS functions. The function *prioerrora* is a collective opera¬tion which prints the error statistics to the standard output of the application.

Bibliography

[Abawajy03] J.H. Abawajy, “Performance analysis of parallel I/O scheduling approaches on cluster computing systems”, 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid, CCGrid’03, 12-15 May 2003, pp. 724-729

[Ali05] Nawab Ali, Mario Lauria, “SEMPLAR: high-performance remote parallel I/O over SRB Cluster Computing and the Grid”, International Symposium on CCGrid, 9-12 May 2005, pp. 366-373 Vol. 1

[Altman01] Eitan Altman, Chadi Barakat, Victor M. Ramos, “Queueing analysis of simple FEC schemes for IP telephony”, INFOCOM 2001, Vol. 2, Ap 22-26, pp. 796-804

[Baer04] Troy Baer, Pete Wyckoff, “A parallel I/O mechanism for distributed systems”, International Conference on Cluster Computing, 20-23 Sept 2004, pp. 63-69

[Bancroft00] Martha Bancroft, Nick Bear, Jim Finlayson, Robert Hill, Richard Isicoff, Hoot Thompson, “Functionality and Performance Evaluation of File Systems for Storage Area Networks (SAN)”, 17-th IEEE Symposium on Mass storage systems, March 2000, http://esdis-it.gsfc.nasa.gov/msst/conf2000/PAPERS/A05PA.PDF

[Baran02] Paul Baran, “The beginnings of packet switching: some underlying concepts”, IEEE Communications Magazine, July 2002, pp 42-48 Vol. 40 Issue 7

[Baran64] Paul Baran, “On Distributed Communications: I. Introduction to Distributed Communications Networks”, Memorandum of the RAND corporation prepared for United States Air Force, August 1964

[Baran65] Paul Baran, “On Survivability of Networks”, IEEE Transactions on Communications, Sep 1965, pp. 379-380 Vol. 13 Issue 3

[Baylor96] S. J. Baylor, C. E. Wu, “Parallel I/O workload characteristics using Vesta”, IPPS’95 Workshop on Input/Output in Parallel and Distributed Systems, Apr. 1995, pp. 16-29

[Boehm64] Sharla P. Boehm, Paul Baran, “On Distributed Communications: II. Digital Simulation of Hot-Potato Routing in a Broadband Distributed Communications Network”, Memorandum of the RAND corporation prepared for United States Air Force, August 1964

[Bradley00] Daryl Bradley, Cesar Ortega-Sanchez, Andy Tyrrell, “Embryonics+immunotronics: a bio-inspired approach to fault tolerance”, The Second NASA/DoD Workshop on Evolvable Hardware, 13-15 July 2000, pp. 215-223

[Brauss99A] Stephan Brauss, Martin Frey, Martin Heimlicher, Andreas Huber, Martin Lienhard, Patrick Muller, Martin Naf, Josef Nemecek, Roland Paul, Anton Gunzinger, “An Efficient Communication Architecture for Commodity Supercomputers”, ACM/IEEE Supercomputing Conference, 13-18 Nov. 1999, pp. 19-35

[Brauss99B] 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

[Byers99] John W. Byers, Michael Luby, Michale Mitzenmacher, "Accessing multiple mirror sites in parallel: using Tornado codes to speed up downloads", INFOCOM 1999, Vol. 1, Mar 21-25, pp. 275-283

[Chandramohan97] Chandramohan A. Thekkath, Timothy Mann, Edward K. Lee, "Frangipani: A Scalable Distributed File System", 16th ACM Symposium on Operating Systems Principles, October 1997, pp. 224-237

[Choi06] Jeong-Yong Choi, Jitae Shin, "A Novel Design and Analysis of Cross-Layer Error-Control for H.264 Video over Wireless LAN", Springer-Verlag LNCS (WWIC'06), May 2006

[Coloma04] Kenin Coloma, Alok Choudhary, Wei-keng Liao, L. Ward, E. Russell, N. Pundit, "Scalable high-level caching for parallel I/O", 18th International Symposium on Parallel and Distributed Processing, 26-30 April 2004, pp. 96-105

[Crandall95] Phyllis E. Crandall, Ruth A. Aydt, Andrew A. Chien, Daniel A. Reed "Input-Output Characteristics of Scalable Parallel Applications", Supercomputing'95. ACM Press, December 1995

[Davies72] Donald Davies, "The Control of Congestion in Packet-Switching Networks", IEEE Transactions on Communications, Jun 1972, pp. 546-550 Vol. 20 Issue 3 Part 2

[Fourer03] Robert Fourer, "A modeling language for mathematical programming", Thomson – Brooks/Cole, second edition, 2003, page 343

[Gabrielyan01] Emin Gabrielyan, "Isolated MPI-I/O for any MPI-1", 5th Workshop on Distributed Supercomputing: Scalable Cluster Software, Sheraton Hyannis, Cape Cod, Hyannis Massachusetts, USA, 23-24 May 2001

[Gennart99] 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

[Gorbett96] Peter F. Gorbett and Dror G. Feitelson, "The Vesta parallel file system", ACM Transactions on Computer Systems – TOCS'96, August 1996, Vol. 14 Issue 3 pp. 225-264, http://www.cs.umd.edu/class/fall2002/cmsc818s/Readings/vesta-tocs96.pdf

[Gregory35] William K. Gregory, "Reduplication in Evolution", Quarterly Review of Biology, 1935, pp. 272-290 Vol. 10

[Gropp98] 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

[Gropp99] William Gropp, Ewing Lusk, Rajeev Thakur, Using MPI-2 Advanced Features of the Message-Passing Interface, MIT Press, pages 51-118, 1999

[Guven04] Tuna Guven, Chris Kommareddy, Richard J. La, Mark A. Shayman, Bobby Bhattacharjee “Measurement based optimal multi-path routing”, INFOCOM 2004, Vol. 1, Mar 7-11, pp. 187-196

[Hoang06] Vinh Dien Hoang, Zhenhai Shao, Masayuki Fujise, “Efficient Load balancing in MANETs to Improve Network Performance”, 6th International Conference on ITS Telecommunications - ITST'06, 21-23 June 2006, pp. 753-756

[Hollywood03] Mark Fritz, “Digital Dailies Flow Freely from Fountain”, April 1, 2003, http://www.emedialive.com/Articles/ReadArticle.aspx?CategoryID=45&ArticleID=5077

[Honda04] Loring Wirbel, “Deal pushes algorithms into digital radio”, April 13, 2004, http://www.commsdesign.com/showArticle.jhtml?articleID=18901216

[Horst95] Robert W. Horst, “TNet: a reliable system area network”, IEEE Micro, Feb. 1995, pp. 37-45 Vol. 15 Issue 1

[Huang05] Yicheng Huang, Jari Korhonen, Ye Wang, “Optimization of Source and Channel Coding for Voice Over IP”, ICME’05, Jul 06, pp. 173-176

[Huber95] Jay V. Huber, Christopher L. Elford, Daniel A. Reed, Andrew A. Chien, David S. Blumenthal, “PPFS: A High Performance Portable Parallel File System”, 9th ACM International Conference on Supercomputing - ACM Press, July 1995, pp. 385-394

[Johansson02] Ingemar Johansson, Tomas Frankkila, Per Synnergren, “Bandwidth efficient AMR operation for VoIP”, Speech Coding 2002, Oct 6-9, pp. 150-152

[Kallahalla02] Mahesh Kallahalla, Peter J. Varman, “PC-OPT: optimal offline prefetching and caching for parallel I/O systems”, IEEE Transactions on Computers, Nov. 2002, pp. 1333-1344 Vol. 51 Issue 11

[Kang05] Seong-ryong Kang, Dmitri Loguinov, “Impact of FEC overhead on scalable video streaming”, NOSSDAV’05, Jun 12-14, pp. 123-128

[Kim06] Dong-hyun Kim, Rhan Ha, Hojung Cha, “Traffic Load and Lifetime Deviation Based Power-Aware Routing Protocol for Wireless Ad Hoc Networks”, 4th International Conference on Wired/Wireless Internet Communications – WWIC’06, 10-12 May 2006, pp. 325-336

[Kotz96] Nils Nieuwejaar, David Kotz, Apratim Purakayastha, Carla Schlatter Ellis, Michael Best, “File-Access Characteristics of Parallel Scientific Workloads”, IEEE Transactions on Parallel and Distributed Systems, October 1996, Vol. 7 Issue 10 pp. 1075-1089

[Kotz97] David Kotz, “Disk-directed I/O for MIMD Multiprocessors”, ACM Transactions on Computer Systems – TOCS’97, February 1997, Vol. 15 Issue 1 pp. 41-74

[Kuonen99A] 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

[Kuonen99B] Pierre Kuonen, “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, April 1999, pp. 381-385

[Lee95] Edward K. Lee, “Highly-Available, Scalable Network Storage”, 40th IEEE Computer Society International Conference – COMPCON’95, March 1995, pp. 397-402

[Lee96] Edward K. Lee and Chandramohan A. Thekkath, “Petal: Distributed Virtual Disks”, Seventh International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS-VII, October 1996, pp. 84-92, ftp://ftp.digital.com/pub/DEC/SRC/publications/eklee/petal-paper.pdf

[Lee98] Edward K. Lee, Chandramohan A. Thekkath, Chris Whitaker, Jim Hogg, “A Comparison of Two Distributed Disk Systems”, Research Report, 30 April 1998, http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-155.html

[Liu03] Pangfeng Liu, Da-Wei Wang, Jan-Jan Wu, “Efficient parallel I/O scheduling in the presence of data duplication”, International Conference on Parallel Processing, 2003, pp. 231-238

[Luby02] Michael Luby, “LT codes”, FOCS’02, November 16-19, pp. 271-280

[Luo06] Jun Luo, “Mobility in Wireless Networks: Friend or Foe, Network design and Control in the Age of mobile Computing”, Thesis 3456 at EPFL, 7 April 2006

[Ma03A] Rui Ma, Jacek Ilow, “Reliable multipath routing with fixed delays in MANET using regenerating nodes”, LCN’03, Oct 20-24, pp. 719-725

[Ma03B] Xiaosong Ma, Xiangmin Jiao, M. Campbell, M. Winslett, “Flexible and efficient parallel I/O for large-scale multi-component simulations”, Parallel and Distributed Processing Symposium, 22-26 April 2003, pp. 10-19

[Ma04] Rui Ma, Jacek Ilow, “Regenerating nodes for real-time transmissions in multi-hop wireless networks”, LCN’04, Nov 16-18, pp. 378-384

[MacKay05] David J. C. MacKay, “Fountain codes”, IEE Communications, Vol. 152 Issue 6, Dec 2005, pp. 1062-1068

[MacKay96] D.J.C. MacKay and R.M. Neal, “Near Shannon limit performance of low density parity check codes”, Electronics Letters 1996, Vol. 32, Issue 18, Aug 29, pp. 1645-1646

[Messerli99] 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

[More97] Sachin More, Alok Choudhary, Ian Foster, Ming Q. Xu, “MTIO a multi-threaded parallel I/O system”, 11th International Parallel Processing Symposium – IPPS’97, pp. 368-373, http://www.ece.northwestern.edu/~choudhar/publications/pdf/MorCho97A.pdf

[MPI2-97A] Message Passing Interface Forum, MPI-2: Extensions to the Message-Passing Interface, July 1997, http://www.mpiforum.org/

[MPI2-97B] Message Passing Interface Forum, MPI-2 Extensions to the Message-Passing Interface, University of Tennessee, 1997, pp. 209-300

[Nguyen02] Thinh Nguyen, Avideh Zakhor, “Protocols for distributed video streaming”, Image Processing 2002, Vol. 3, Jun 24-28, pp. 185-188

[Nguyen03] Thinh Nguyen, P. Mehra, Avideh Zakhor, “Path diversity and bandwidth allocation for multimedia streaming”, ICME’03 Vol. 1, Jul 6-9, pp. 663-672

[Oldfield98] Ron Oldfield, David Kotz, “The Armada Parallel File System”, Scientific Report - Dartmouth College - Compute Science Department, 22 November 1998, http://www.cs.dartmouth.edu/~dfk/armada/

[Pacheco97] Peter S. Pacheco, Parallel Programming with MPI, by Morgan Kaufmann Publishers 1997, pp. 137-178

[Padhye00] Chinmay Padhye, Kenneth J. Christensen, Wilfrido Moreno, “A new adaptive FEC loss control algorithm for voice over IP applications”, IPCCC’00, Feb 20-22, pp. 307-313

[Ping06] Yuan Ping, Bai Yu, Wang Hao, “A Multipath Energy-Efficient Routing Protocol for Ad hoc Networks”, International Conference on Communications, Circuits and Systems - ICCCAS'06, 25-29 June 2006, pp. 14662-1466 Vol. 3

[Qiao99] Chunming Qiao, Myungsik Yoo, “Optical burst switching (OBS) – a new paradigm for an Optical Internet”, Journal of High Speed Networks, 1999, pp. 69-84 Vol. 8 Num. 1

[Qu04] Qi Qu, Ivan V. Bajic, Xusheng Tian, James W. Modestino, “On the effects of path correlation in multi-path video communications using FEC over lossy packet networks”, IEEE GLOBECOM’04 Vol. 2, Nov 29 - Dec 3, pp. 977-981

[Richardson01] Thomas J. Richardson and Rüdiger L Urbanke, Efficient Encoding of Low-Density Parity Check Codes, IEEE Transactions on Information Theory, Vol. 47, No. 2, February 2001, pp. 638-656

[Schwarz02] Thomas S. J. Schwarz, Generalized Reed Solomon codes for erasure correction in SDDS, In Workshop on Distributed Data and Structures, WDAS 2002, Paris, Mar 2002

[Seroussi86] Gadiel Seroussi, Ron M. Roth, On MDS extensions of generalized Reed-Solomon codes, IEEE Transactions on Information Theory, Vol. 32, Issue 3, May 1986, pp. 349-354

[Shokrollahi04] Amin Shokrollahi, “Raptor codes”, ISIT’04, June 27 – July 2, page 36

[Smirni96] Evgenia Smirni, Ruth A. Aydt, Andrew A. Chien, Daniel A. Reed, “I/O Requirements of Scientific Applications: An Evolutionary View”, Fifth IEEE International Symposium on High Performance Distributed Computing, 1996, pp. 49-59

[Snir96] 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

[SwissTx01] Swiss-Tx Project Report, June 2001, http://hefrweb01.eif.ch/~kuonen/grip/html/ficheSWISS.html

[Tawan04] Tawan Thongpook, "Load balancing of adaptive zone routing in ad hoc networks", TENCON 2004, Vol. B, Nov 21-24, pp. 672-675

[Thakur96A] Rajeev Thakur, William Gropp, Ewing Lusk, "An Experimental Evaluation of the Parallel I/O Systems of the IBM SP and Intel Paragon Using a Production Application", 3rd International Conference of the Austrian Center for Parallel Computation (ACPC) with Special Emphasis on Parallel Databases and Parallel I/O, Lecture Notes in Computer Science - Springer-Verlag, September 1996, pp. 24-35

[Thakur96B] R. Thakur, W. Gropp, and E. Lusk, "An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces," in Proc. of the 6th Symposium on the Frontiers of Massively Parallel Computation, October 1996, pp. 180-187

[Thakur98] Rajeev Thakur, William Gropp, Ewing Lusk "A Case for Using MPI's Derived Datatypes to Improve I/O Performance", Conference on High Performance Networking and Computing, 1998, pp. 1-10, http://www-unix.mcs.anl.gov/~thakur/dtype/

[Thakur99A] Rajeev Thakur, William Gropp, Ewing Lusk, "On implementing MPI-IO Portably and with High Performance", 6th Workshop on I/O in Parallel and Distributed Systems, 5 May 1999, pp. 23-32.

[Thakur99B] Thakur, R.; Gropp, W.; Lusk, E., "Data sieving and collective I/O in ROMIO", The Seventh Symposium on the Frontiers of Massively Parallel Computation, Frontiers'99, 21-25 Feb 1999, pp. 182-189

[Worster97] Tom Worster, Avri Doria, "Levels of aggregation in flow switching networks", Electronics Industries Forum of New England, 6-8 May 1997, pp. 51-59

[Wu05A] Jan-Jan Wu, Yih-Fang Lin, Pangfeng Liu, "Efficient distributed algorithms for parallel I/O scheduling", 11th International Conference on Parallel and Distributed Systems, 20-22 July 2005, pp. 460-466 Vol. 1

[Wu05B] Jan-Jan Wu, Pangfeng Liu, "Distributed Scheduling of Parallel I/O in the Presence of Data Replication", 19th IEEE International Symposium on Parallel and Distributed, 04-08 April 2005, pp. 49b - 49b

[Xu00] Youshi Xu, Tingting Zhang, "An adaptive redundancy technique for wireless indoor multicasting", ISCC 2000, Jul 3-6, pp. 607-614

Biography

Personal Bibliography

Glossary

SRI	Stanford Research Institute
UCLA	University of California, Los Angeles
IMP	Interface Message Processor
ARPANET	Advanced Research Projects Agency Network
DoD	The U.S. Department of Defense
MPI	Message Passing Interface
TCP	Transmission Control Protocol
WDM	Wavelength Division Multiplexing
DWDM	Dense Wavelength Division Multiplexing
MYRINET	is a high-speed local area networking system designed by Myricom to be used as an interconnect between multiple machines to form computer clusters
ATM	Asynchronous Transfer Mode, a telecommunication protocol
TDM	Time-Division Multiplexing, a technology in circuit-switched digital telephony
OBS	Optical Burst Switching
3G	3rd Generation mobile communication
3GPP	3rd Generation Partnership Project
ADSL	Asynchronous Digital Subscriber Line
AMR	Adaptive Multi-Rate voice codec 4.75 - 12.2 kbps
ROR	Redundancy Overall Requirement
ARQ	Automatic Repeat reQuest
BER	Bit Error Rate
CPU	Central Processing Unit
DER	Decoding Error Rate
DoS	Deny of Service
EIGRP	Enhanced Interior Gateway Routing Protocol
FEC	Forward Error Correction
FIFO	First In, First Out
g723r53	High complexity voice codec G.723.1 5300 bps
g723r63	High complexity voice codec G.723.1 6300 bps
g729r8	Low complexity voice codec G.729 8000 bps
gsmfr	High complexity voice codec GSMFR 13200 bps
HTTP	HyperText Transfer Protocol

IOS	Internet Operating System
IP	Internet Protocol
ISP	Internet Service Provider
ITSP	Internet Telephony Service Provider
LP	Linear Programming
LT	Luby Transform Code
MANET	Mobile Ad-hoc Network
MBMS	Multimedia Broadcast/Multicast Service
MDS	Maximum Distance Separable
MPEG	Moving Picture Experts Group
NAT	Network Address Translation
QoS	Quality of Service
RS	Reed-Solomon
RTP	Real-time Transport Protocol
RTT	Round Trip Time
SIP	Service Initiating Protocol
UA	User Agent
UDP	User Datagram Protocol
VOIP	Voice Over IP
VPN	Virtual Private Network
XOR	eXclusive OR
SFIO	Striped File I/O
TNET	High-performance switch-based communication network aiming at low-latency and high-bandwidth
I/O	Input-Output
EPFL	École Polytechnique Fédérale de Lausanne, Swiss Federal Institute of Technology Lausanne, http://www.epfl.ch/
FIFO	First In, First Out
FCI	Fast Communication Interface
ETHZ	Eldgenössische Technische Hochschule Zürich, Swiss Federal Institute of Technology Zurich
CTI	Swiss Commission for Technology and Innovation
SCS	Supercomputing Systems
LSP	Laboratoire de Systèmes Périphériques, Peripheral Systems Laboratory of EPFL, http://diwww.epfl.ch/w3lsp/
API	Application Program Interface

SAN	Storage Area Networks
OS	Operating System
GPS	Global Positioning System
SNL	Sandia National Laboratories, http://www.sandia.gov/
ORNL	Oak Ridge National Laboratory, http://www.ornl.gov/
MPICH	“CH” in MPICH stands for “Chameleon”, symbol of adaptability to one’s environment and thus of portability
ADIO	Abstract Device Interface for Portable Parallel I/O
DMA	Direct Memory Access
ANL	Argonne National Laboratory, http://www.anl.gov/

Table of Figures