

THREE TOPICS IN PARALLEL COMMUNICATIONS

THÈSE N° 3660 (2006)

PRÉSENTÉE LE 29 SEPTEMBRE 2006
À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
SECTION D'INFORMATIQUE

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Emin GABRIELIAN

Ingénieur physicien diplômé de l'Université d'État d'Erevan, Arménie
et de nationalité arménienne

acceptée sur proposition du jury:

Prof. Boi Faltings, président du jury
Prof. Roger D. Hersch, directeur de thèse
Prof. Claude Petitpierre, rapporteur
Prof. Jean-Frédéric Wagen, rapporteur
Prof. Pascal Lorenz, rapporteur

Lausanne, EPFL
2006

To Ani, Eva, and Léa

Acknowledgements

First, I would like to thank my research director, Professor Roger Hersch, who gave me guidance throughout this research project, and spent hours reading and commenting the drafts of my papers and dissertation.

Next I would like to thank all my colleagues who made the lab an interesting and entertaining place. First, those who already left the lab: Yvette Fishman, Victor Ostromoukhov, Benoît Gennart, Oscar Figueiredo, Jean-Christophe Bessaud, Joaquín Tárraga Giménez, Marc Mazzariol, Patrick Emmel, Edouard Forler, Emzar Panikashvili, Laurent Saroul, and those who are still here: Isaac Amidror, Sebastian Gerlach, Fabienne Allaire, and Basile Schaeli.

I am grateful to participants of the Swiss-Tx project, Ralf Gruber, Ivan Sipos, Bill Camp, and Martin Frey for an incredibly motivating atmosphere.

I would also like to thank Aram Nanasyan and Suren Simonyan, for motivating me to start and accomplish this research.

I am very grateful to my parents, Romik Gabrielyan and Nazik Mkrtchyan, who developed my curiosity. I am particularly indebted to my father, Romik, for his unlimited and unconditional efforts to support any of my endeavors. I am grateful to my wife, Sona Gabrielyan, for her unconditional love, and for her infinite patience. Additional thanks go to her and also to my brother, Aram Gabrielyan, for their full support of our VOIP businesses in Switzerland and in USA during my research.

Summary

The main objectives pursued by parallelism in communications are network capacity enhancement and fault-tolerance. Efficiently enhancing the capacity of a network by parallel communications is a non-trivial task. Some applications may also allow one to split the sources and destinations into multiple sources and destinations. An example is parallel Input/Output (I/O). Parallel I/O requires scalability, high throughput and good load balance. Low granularity enables good load balance but tends to reduce throughput. In this thesis we combine fine granularity with scalable high throughput. The network overhead can be reduced and the network throughput can be increased by aggregation of data into large messages. Parallel transmissions from multiple sources to multiple destinations traverse the network through many different paths which have numerous intersections in the network. In low latency high performance networks, serious congestions occur due to large indivisible messages competing for shared resources. We propose to optimally schedule parallel communications by taking into account the network topology. The developed liquid scheduling method optimally uses the potential transmission capacity of a network. Fault-tolerance is typically achieved by maintaining backup communication resources, which are kept idle as long as the primary resource is operational. A challenging idea, inspired by nature, is to simultaneously use all parallel resources. This idea is applied to fine-grained packetized communications. It also relies on erasure resilient codes for combating network failures.

KEYWORDS. Parallel communications, fault-tolerance, liquid scheduling, capillary routing, circuit-switching, circuit-switched networks, VOIP, Internet telephony, SIP, packetized telephony, real-time streaming, path diversity, redundancy overall requirement, ROR, coarse-grained networks, fine-grained networks, wormhole switching, optical lightpath routing, cut-through switching, graph coloring, congestion graph, traffic partitioning, mutually non-congesting subsets, conflict graph, low granularity striping, scalable I/O, parallel I/O, Message Passing Interface, MPI-I/O, network aggregation, I/O access aggregation, erasure resilient codes, channel coding, forward error correction

Résumé

Les communications parallèles ont pour objectif d'augmenter la capacité ainsi que la tolérance aux pannes des réseaux de transmission de données. Augmenter efficacement la capacité d'un réseau par des communications parallèles est une tâche non triviale, car les liens de communication parallèles peuvent être disposés selon une topologie arbitraire et peuvent partager certaines ressources. Certaines applications permettent aussi de séparer des sources et destinations uniques en multiples sources et destinations. Les entrées/sorties (E/S) parallèles constituent un tel exemple. Les E/S parallèles doivent permettre la croissance du système, un débit élevé, et un bon équilibrage des charges. Une granularité faible permet un bon équilibrage des charges, mais tend à réduire le débit. Dans cette thèse, nous combinons une granularité fine avec un débit élevé tout en permettant la croissance du système. L'agrégation des données dans des messages de grande taille permet d'augmenter le débit tout en réduisant les surcoûts sur le réseau. Des transmissions parallèles de sources multiples vers des destinations multiples traversent le réseau par de nombreux chemins s'intersectant en de nombreux points. Dans des réseaux haute-performance à faible latence, des congestions importantes sont causées par de gros messages indivisibles en compétition pour des ressources partagées. Nous proposons d'ordonnancer les communications parallèles de manière optimale en prenant en considération la topologie du réseau. La méthode d'ordonnancement liquide (liquid scheduling) développée utilise au maximum les capacités de transmission potentielles du réseau. La tolérance aux pannes est généralement obtenue en maintenant des ressources de communication supplémentaires qui ne sont pas utilisées tant que la ressource principale est opérationnelle. Une idée stimulante, inspirée par la nature, est d'utiliser simultanément toutes les ressources disponibles. Cette idée est appliquée à des communications par paquets à granularité fine. Elle s'appuie aussi sur des codages permettant de compenser les pertes d'informations lors des pannes du réseau.

MOTS CLÉ. Communications parallèles, tolérance aux pannes, ordonnancement liquide, routage par capillarité, commutation de circuits, réseau à communication de circuits, voix sur IP, téléphonie par Internet, SIP, téléphonie IP, réseaux à granularité grossière, réseaux à granularité fine, routage optique, graphes de congestion, partitionnement de trafic, distribution à granularité faible, E/S parallèles, Message Passing Interface, agrégation d'accès E/S, redondance, codage de canal, correction d'erreurs en boucle ouverte (FEC)

Table of Contents

Acknowledgements	v
Summary	vii
Résumé	ix
Table of Contents	xi
Chapter 1. Introduction	1
Section 1.1. Parallel communication challenges	1
Section 1.2. Capacity enhancement and fault-tolerance	3
Section 1.3. Fine-grained and coarse-grained network paradigms	4
1.3.1. Packet switching or hot potato routing	4
1.3.2. Wormhole routing	6
Section 1.4. Three topics in parallel communications	7
1.4.1. Problems and the objectives	7
1.4.2. Structure of the thesis	8
Chapter 2. Parallel I/O solutions for cluster computers	11
Section 2.1. Introduction	11
Section 2.2. Project framework	12
Section 2.3. File striping	14
Section 2.4. Implementation layers	15
Section 2.5. The SFIO Interface	16
Section 2.6. Optimization principles	18
Section 2.7. Functional architecture and implementation	21
Section 2.8. SFIO performance	23
2.8.1. Network and parallel I/O throughput when using Fast Ethernet	23
2.8.2. Network and parallel I/O throughput when using TNET	25
Section 2.9. MPI-I/O implementation on top of SFIO	28
Section 2.10. Conclusions and recent developments in parallel input-output	32
Chapter 3. Liquid scheduling of parallel transmissions in coarse-grained low-latency networks	35
Section 3.1. Introduction	35
3.1.1. Parallel transmissions in circuit-switched networks	35
3.1.2. Hardware solutions	36
3.1.3. Liquid scheduling - an application level solution	37
3.1.4. Overview of liquid scheduling	37
Section 3.2. Applicable networks	38
3.2.1. Wormhole switching	38
3.2.2. Optical networks	39

Section 3.3.	The liquid scheduling problem.....	42
Section 3.4.	Definitions.....	44
Section 3.5.	Obtaining full simultaneities	46
3.5.1.	Using categories to cover subsets of full simultaneities	47
3.5.2.	Fission of categories into sub-categories	47
3.5.3.	Traversing all full simultaneities by repeated fission of categories	48
3.5.4.	Optimization - identifying blank categories.....	49
3.5.5.	Retrieving full teams - identifying idle categories.....	49
Section 3.6.	Speeding up the search for full teams	50
3.6.1.	Skeleton of a traffic.....	50
3.6.2.	Optimization - building full teams based on full teams of the skeleton.....	51
3.6.3.	Evaluating the reduction of the search space	51
Section 3.7.	Construction of liquid schedules	53
3.7.1.	Definition of a liquid schedule.....	53
3.7.2.	Liquid schedule basic construction algorithm.....	55
3.7.3.	Search space reduction by considering newly emerging bottlenecks	56
3.7.4.	Liquid schedule construction optimization by considering only full teams.....	57
Section 3.8.	Experimental verification.....	57
3.8.1.	Swiss-Tx cluster supercomputer and 362 test traffic patterns.....	58
3.8.2.	Real traffic throughout measurements	61
Section 3.9.	Conclusions	62
Chapter 4.	Capillary routing for fault-tolerant real-time communications in fine-grain packet-switching networks	65
Section 4.1.	Introduction	65
Section 4.2.	Capillary routing	67
4.2.1.	Basic construction.....	67
4.2.2.	Numerically stable version	69
4.2.3.	Bottleneck hunting loop.....	71
Section 4.3.	Redundancy Overall Requirement (ROR)	73
4.3.1.	Definition of ROR	73
4.3.2.	Computing FEC block size	74
4.3.3.	Streaming with large FEC blocks	76
Section 4.4.	Redundancy Overall Requirement in capillary routing.....	77
Section 4.5.	Conclusions and perspectives	79
Conclusions		81
Parallel I/O.....		81
Liquid schedules		82
Capillary routing.....		83
Further work		84
Appendix A.	SFIO function calls	85
Section A.1.	File management operations	85
Section A.2.	Data access operations	85

Section A.3.	Error management operations	86
Appendix B.	Congestion graph coloring heuristic approach	87
Appendix C.	Comparison of the liquid scheduling algorithm with Mixed Integer Linear Programming	91
Appendix D.	Assembling a liquid schedule: Considering teams of the reduced traffic instead of the teams of the original traffic	93
Appendix E.	Assembling a liquid schedule: Considering full teams of the reduced traffic instead of all its teams	97
Appendix F.	Overall overview of all liquid schedule construction optimizations	99
Appendix G.	Probability of simultaneous link failures in multi-path routing patterns	101
Section G.1.	Limitations of the single link failure assumption.....	101
Section G.2.	Extension of ROR for considering also the overlapping failures.....	103
Bibliography		107
Biography		i
Personal Bibliography		iii
	Publications related to parallel I/O	iii
	Conference papers on liquid scheduling problem.....	iii
	Papers related to capillary routing	iv
Glossary		v
List of Figures		xi
List of Tables		xv
Links		xvii

Chapter 1. Introduction

In this chapter we briefly introduce the history of parallel communications and the topics of capacity enhancement and fault-tolerance. We present the fine-grained and coarse-grained network paradigms and introduce the topics of the present thesis.

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. The signal was dead on September 18.

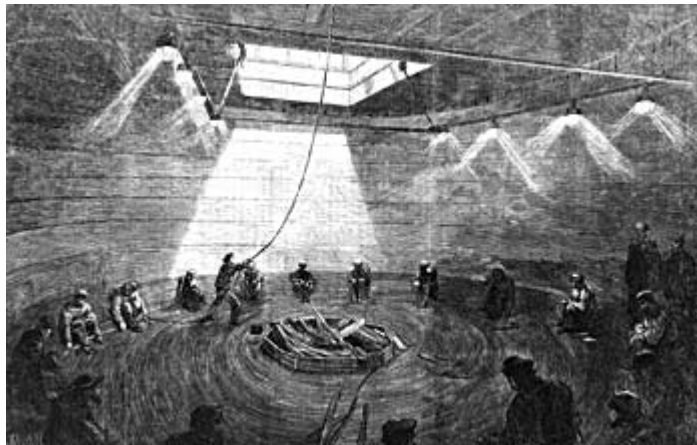


Figure 1. Loading the transatlantic cable into the ‘Great Eastern’ in 1865

Eight years later, on 13 July 1866, the Great Eastern, by far the largest ship, began laying another cable, this time made of a single piece, 2730 nautical miles long, insulated with a new resin from the gutta-percha tree found in the Malay Archipelago. When two weeks later, on 27th of July 1866, the cable began operating, for Cyrus Field the mission was not yet accomplished. He immediately sent the Great Eastern back to sea to lay the second parallel cable. By 17 September 1866, not one, but two parallel circuits were sending messages across the Atlantic.

The transatlantic cable station, operating those links, was transmitting messages for nearly 100 years. It was still in operation when in March 1964, in the middle of the cold war, an article appeared, entitled “On Distributed Communications Networks”. It was written by Paul Baran, who at that time 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 showed 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 forerunner of the global Internet, the Advanced Research Projects Agency Network (ARPANET) of the U.S. Department of Defense, was born.

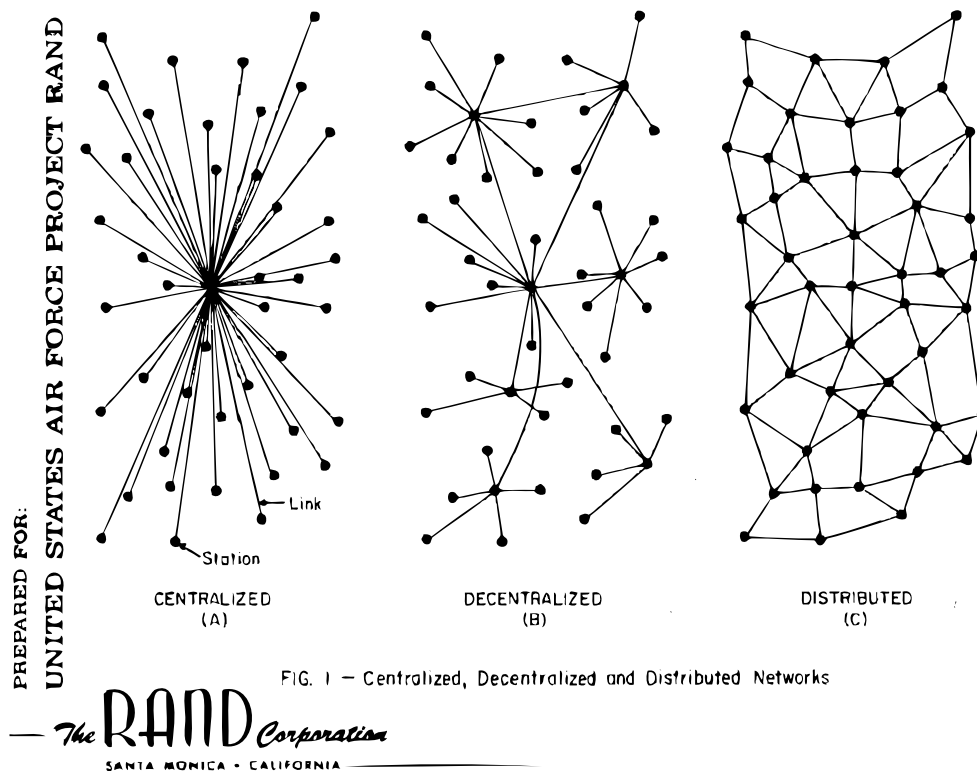


Figure 2. Diagrams from the 51-page report of Paul Baran to the U.S. Air Force, 1964

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. 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, or for power efficiency, e.g. in mobile networks [Ping06], [Luo06], [Kim06].

Section 1.2. Capacity enhancement and fault-tolerance

The focus of research in parallel communications aims mainly at maximizing capacity and 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, nature has created many systems relying on parallel structures. When developing his distributed network models (the seeds of the Internet), Paul Baran himself inspired by discussions with neurophysiologist Warren Sturgis McCulloch [Pitts47], [McEneaney02], [McCulloch43] about the capability of the brain to recover lost functions by bypassing a dysfunctional region thanks to parallel structures. Living multi-cellular organisms, from insects to vertebrates, demonstrate numerous other examples of duplicated organs that function in parallel. The evolution of life on earth made replicated organs nearly a universal property of living bodies [Gregory35].

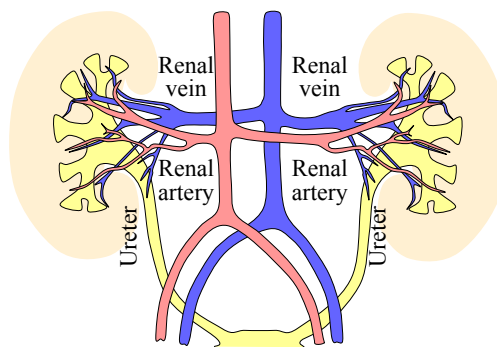


Figure 3. Kidney blood filtering in the human organism

The primary purpose of duplication of organs is the tolerance to failures. Often, 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].

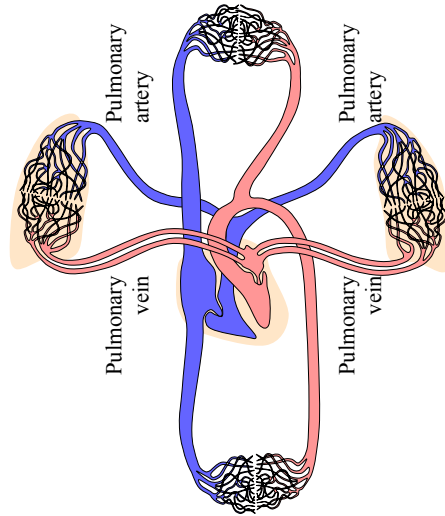


Figure 4. Pulmonary circuit of the human organism

Maintaining an idle parallel resource has already been used in many mission-critical man-made systems. In networking, communications 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 5).



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. One way to avoid congestions is to simply discard the new packets arriving at the switch, if no room is left in the buffer (e.g., UDP). The adjustable window method for avoiding congestion, gives the original sender the right to send N packets before getting permission to send more (e.g., TCP).

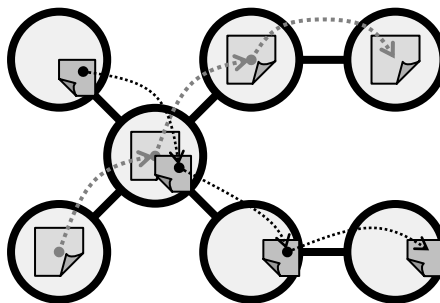


Figure 6. Packet switching network: packets are entirely stored at each intermediate switch and only then forwarded to the next switch

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 (Figure 6). 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 High Performance Computing (HPC), 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 the requirements of a computing cluster. Wormhole routing technology solves the problem of the propagation of the delay across a multi-hop communication path - a serious obstacle in store-and-forward switching.

The address is very short. It 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 entire 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 almost as soon as the sending node starts its transmission. The message is simply “copied” from the source to the destination without ever being 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 through the network. The latency between two nodes, even if separated by many hops, becomes similar to the latency of directly connected nodes.

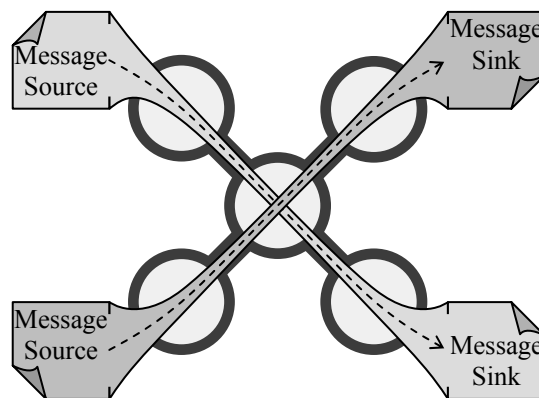


Figure 7. Wormhole or cut-through routing network: a packet is “copied” through the communication path from the source directly to the destination without being stored in any intermediate switch

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 aim 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 (VCT) routing are all broadband or local area network examples of the coarse-grained switching paradigm [Worster97], [Qiao99].

More information about wormhole and optical lightpath routing is given in Chapter 3 (Subsections 3.2.1 and 3.2.2 respectively).

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 most important in the domains covered by parallel communications. While 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 concurrently accessing a shared parallel file. Parallel I/O is used in computer clusters interconnected with a high performance coarse-grained network (such as MYRINET [Boden95]) 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 such a “copy” process, all intermediate switches and links are simultaneously involved in directing the content of the message. Low latency, however, is attained at a cost of an increased tendency toward congestion. When the network paths of

several transmissions overlap, an attempt to carry them out in parallel will unavoidably cause 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 a 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 also paths increases 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 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 the 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 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 in the number of contributing nodes. The gain achieved from the aggregation of communications in parallel I/O at the connection layer can be undermined by losses due to blocked messages occurring at the network layer. Solving congestions locally by FIFO techniques 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 grain networks achieves the throughput of a fine grain network or equivalently, that of a liquid flowing through a network of pipes.

Chapter 4 is dedicated to fault-tolerant multi-path streaming in packetized fine grain networks. We demonstrate that in packet-switched networks, combination of channel coding at the packet level with multi-path parallel routing significantly improves the fault-tolerance of communications, especially in real-time streaming. We show that further development of the path diversity in multi-path parallel routing patterns often brings an additional benefit to the streaming application. We create a *capillary routing* algorithm generating parallel routing patterns of increasing path diversity. We also introduce a method for rating multi-path routing patterns of any complexity with a single scalar value, called ROR, standing for *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. Thanks to small striping units one can achieve high efficiency and a good load balance. Small stripe unit size, however, increases the communication and disk access costs. By optimizing communications and disk accesses, SFIO exhibits high performance even for very small striping factors. We present the functional architecture of the SFIO system. Using MPI derived datatype capabilities, we transmit highly fragmented data over the communication 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 carried out on the Swiss-Tx cluster supercomputer consisting of DEC Alpha computers, interconnected with both Fast Ethernet and a coarse-grain low latency communication network, called TNET.

Section 2.1. Introduction

Parallelism in I/O access and communications relies on the distribution of 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.

We would like to combine an extremely fine granularity (providing a high load balance) with a very high throughput, and at the same time, ensure a linear scalability. Scalability and high performance at extremely small stripe unit sizes are achievable thanks to following three proposed optimization techniques.

Firstly, a multi-block user interface enables the library to recognize the overall pattern of multiple user requests. This multi-block interface permits the caching system (see below) to

aggregate the network and disk accesses which can also be fragmented due to the user memory layout (apart the striping of the global file across multiple disks).

Secondly, the compute nodes perform the caching of I/O requests. The caching system aggregates all network transfers to and from individual I/O nodes. Fragmentations due to both file striping and multi-block user layout are merged in the same caching system. Network aggregation of the incoming traffic is also performed by the compute nodes. The data segments traversing the network are therefore combined into very large messages, thus reducing the communication overhead to the minimum. The drawback of this method is an increased risk of congestion, which is the subject of the second topic addressed in this thesis (see Chapter 3).

Thirdly, at the compute nodes the caching system preprocesses the collected I/O requests addressed to each individual I/O destination. It removes the overlapping segments and sorts the requests according to their offsets. Whenever possible, the caching preprocessor merges multiple remote I/O requests into a single contiguous I/O request. Since network transmissions to individual destinations are already aggregated by both the compute nodes and the I/O nodes, merging multiple I/O requests into single ones does not yield an additional gain with respect to network communication performance. However, the performance gain from merging I/O access requests is considerable with respect to disk access performance.

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 streams the actual data directly between the network and the fragmented memory pattern. The zero-copy implementation relies on MPI derived datatypes [[Snir96](#)], which are built on the fly.

Section 2.2. Project framework

In 1998, EPFL, ETHZ, Supercomputing Systems (SCS), and Compaq Computer Corporation, in a cooperation with the Sandia National Laboratory (SNL) and the Oak Ridge National Laboratory (ORNL) started a common project called Swiss-Tx. The project aims at developing and building a teraflop supercomputer based mainly on commodity parts, such as Compaq Alpha Computers [[SwissTx01](#)]. The communication hardware and software were designed by SCS. It comprises an efficient communication library, called Fast Communication Interface (FCI) and custom-made communication hardware for the Swiss-Tx supercomputer, called TNET [[Brauss99A](#)]. TNET is a proprietary high performance, low-latency and high-bandwidth network. A full implementation of MPI for the TNET network is also available (on top of FCI).



Figure 8. Swiss-Tx supercomputer in June 2001

In many parallel applications I/O is a major bottleneck. I was in charge of the design of an MPI based parallel I/O system for the Swiss-Tx parallel supercomputer.

Although the I/O subsystems of parallel computers are designed for high performance, a large number of applications achieve only about one 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 data size accesses (on the order of megabytes), whereas parallel applications typically make many small I/O requests (of the order of kilobytes or 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 data size accesses as small as only one hundred bytes. Such an extremely small stripe unit size provides a 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 also allowed us to implement a portable MPI-I/O interface [Gabrielyan01]. Finally, thanks to the overlapping of communications and I/O, and to optimizations of I/O requests cached at the compute nodes, SFIO exhibits high 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].

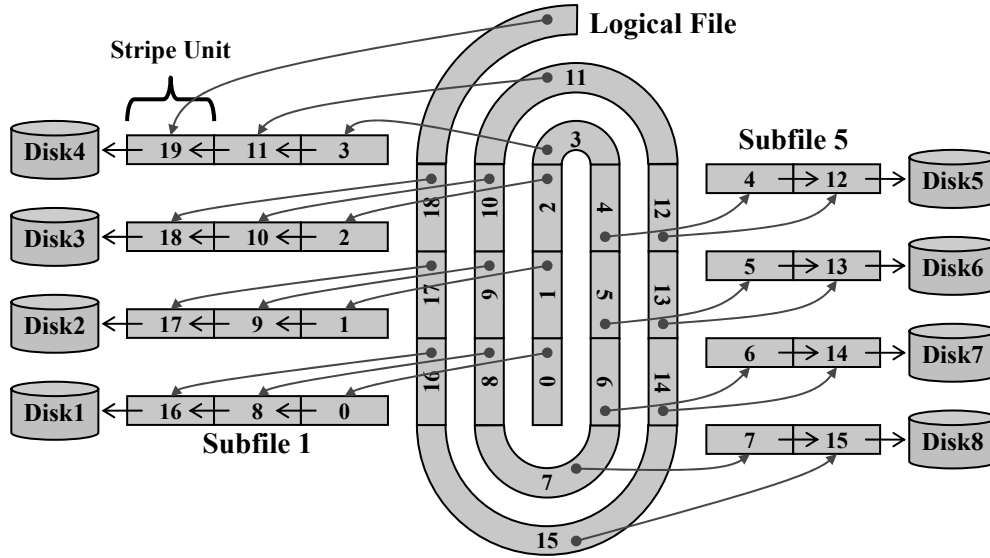


Figure 9. File Striping

A parallel I/O system should offer all parallel application processes highly concurrent access capabilities to the common data files. It should exhibit a linear increase in performance when increasing both the number of I/O nodes and the number of compute 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 Argonne 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. In 2001, the I/O implementation underlying MPICH's MPI-I/O was sequential, and based on NFS [Thakur99A], [Thakur98].

In the 2001 version of MPICH, due to the locking mechanisms needed to avoid simultaneous multiple accesses to the shared NFS file, MPICH MPI-I/O write operations could out be carried only at a very slow throughput.

Another factor reducing peak performance is the read-modify-write operation, useful for writing fragmented data to the target file. Read-modify-write requires reading the full contiguous extent 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 within the file over a large data space, network access overhead becomes dominant.

SFIO aims at offering scalable I/O throughput. However, the fine granularity, required for the best parallelization and load balance, increases the communication and disk access costs. Our SFIO parallel file striping implementation carries out efficient optimizations by merging sets of fragmented network messages and disk accesses into single contiguous messages and disk access requests respectively. The data 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 overlapped resulting in additional performance gain.

Section 2.4 presents the overall architecture of the SFIO implementation. 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 performances are given for various configurations of the Swiss-Tx supercomputer. 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 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).

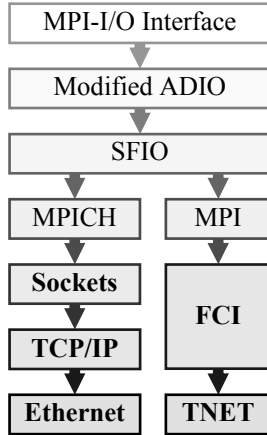


Figure 10. SFIO integration into MPI-I/O

On the Swiss-T1 machine (Swiss-T1 is a 64-processor implementation in scope of the Swiss-Tx project), 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 main interface functions of SFIO. The full list of API functions is given in Appendix A. Two functions, *mopen* and *mclose* are provided to open and close a striped file. In order to ensure the correct behavior of collective parallel I/O functions, these functions are collective operations performed in all contributing processing nodes. In addition, the operation of opening as well as that 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. The first argument of *mopen* is a single string specifying the global file name, which contains the locations and names of all subfiles, separated by semi-colons. The second argument of *mopen* is the stripe unit size in bytes.

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 handling operations, such as *mdelete* or *mcreate* also rely on this simple global file name format. SFIO does not maintain any global metafile, nor any hidden metadata in the subfiles. The sum of sizes of all subfiles is exactly the size of the logical parallel file.

The generic functions for read and write accesses to a file are *mreadc* and *mwritec* respectively. 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*.

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 from one MPI compute process. A single compute node opens a striped file with two subfiles */tmp/a1.dat* at *p1* and */tmp/a2.dat* at *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("p1/tmp/a1.dat;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.

```
#include <mpi.h>
#include "/usr/local/sfio/mio.h"
int _main(int argc, char *argv[])
{
    MFILE *f;
    int r=rank();
    //Collective open operation
    f=mopen("p1/tmp/a.dat;p2/tmp/a.dat;", 5);
    //each process writes 8 to 14 characters at its own position
    if(rank==0) mwritec(f,0,"Good*morning!",13);
    if(rank==1) mwritec(f,13,"Bonjour!",8);
    if(rank==2) mwritec(f,21,"Buona*mattina!",14);
    mclose(f); //Collective close operation
}
```

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 a different position simultaneously.

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 writing to the parallel file, the global file contains the text combined from the fragments written by the first, second and third compute nodes, i. e:

"Good*morning!Bonjour!Buona*mattina!"

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

"Good*ng!Bo!Buontina!"

And the second subfile contains (see Figure 11):

"morninjoura*mat "

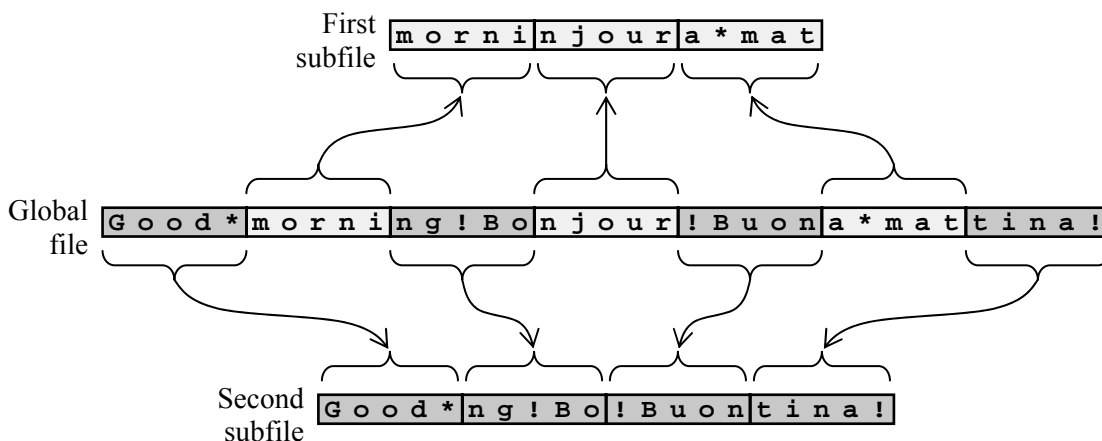


Figure 11. Distribution of a striped file across subfiles

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

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 node 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 of data chunks is followed by confirmation reply messages sent by the I/O listeners to the compute node.

Optimizations of network communications and the remote disk accesses are performed on the compute node. Requests queued for each I/O node are sorted according to their offsets in the remote disk subfile. Then all overlapping or consecutive I/O requests held in the cache are combined, and a new optimized set of requests is formed (Figure 12). This new set of requests creates a new fragmented access pattern within the user memory.

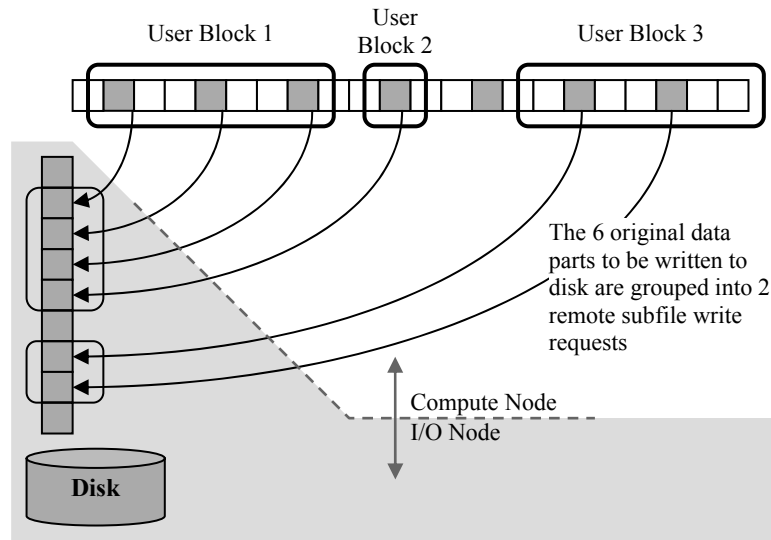


Figure 12. Disk access optimization

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 copied. 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 carried out 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 from the corresponding compute node.

These optimizations are especially valuable for low stripe unit sizes. Figure 13 shows a comparison of a typical non-optimized write operation and its optimized counterpart.

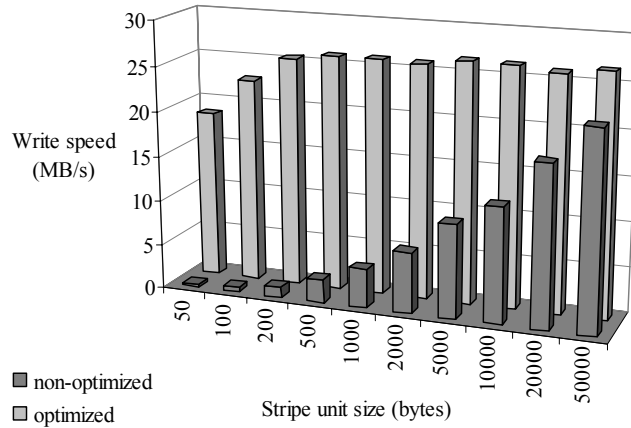


Figure 13. Comparison of the optimized write access with a non-optimized write access as a function of the file striping granularity (3 I/O nodes, 1 compute node, global file size is 660 Mbytes)

The multi-block interface of SFIO enables one to carry out several contiguous blocks of I/O access operations by a single multi-block operation. Thanks to the relevant network optimizations, the performance gain achieved by multi-block access operations is significant. Figure 14 compares the I/O throughput of a multi-block write operation with the throughput achieved by a set of corresponding non-optimized single-block operations.

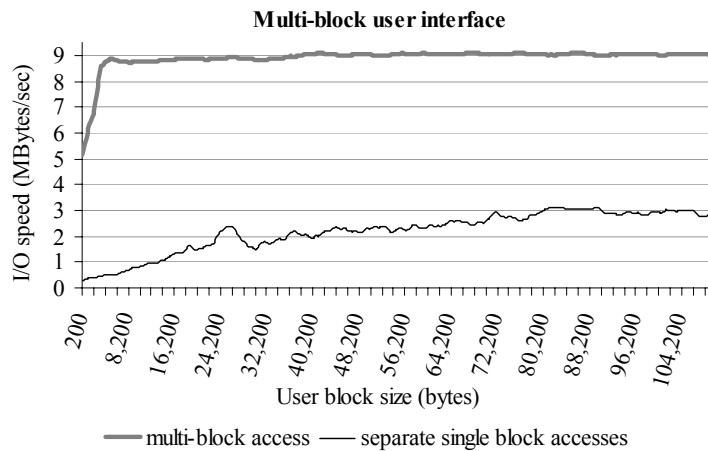


Figure 14. Comparison of the optimized multi-block write access with corresponding separate non-optimized single block accesses (Fast Ethernet, stripe unit size is 1005 bytes, 7 I/O nodes)

Since the single block operations of Figure 14 are not optimized, their total throughput is bounded by an upper limit related to the striping factor of the global file (the same for all user block sizes). Even at very large user block sizes the total throughput of the single block operations is below 3.3 Mbytes/sec due to the striping factor of 1005 bytes (see also Figure 13 for a reference). The multi-block interface permits one to fully benefit from the optimization subsystem [Gabrielyan00].

Section 2.7. Functional architecture and implementation

In this section we describe the functional architecture and the implementation of the access functions. An overall diagram of the implementation of the SFIO access function is shown in Figure 15.

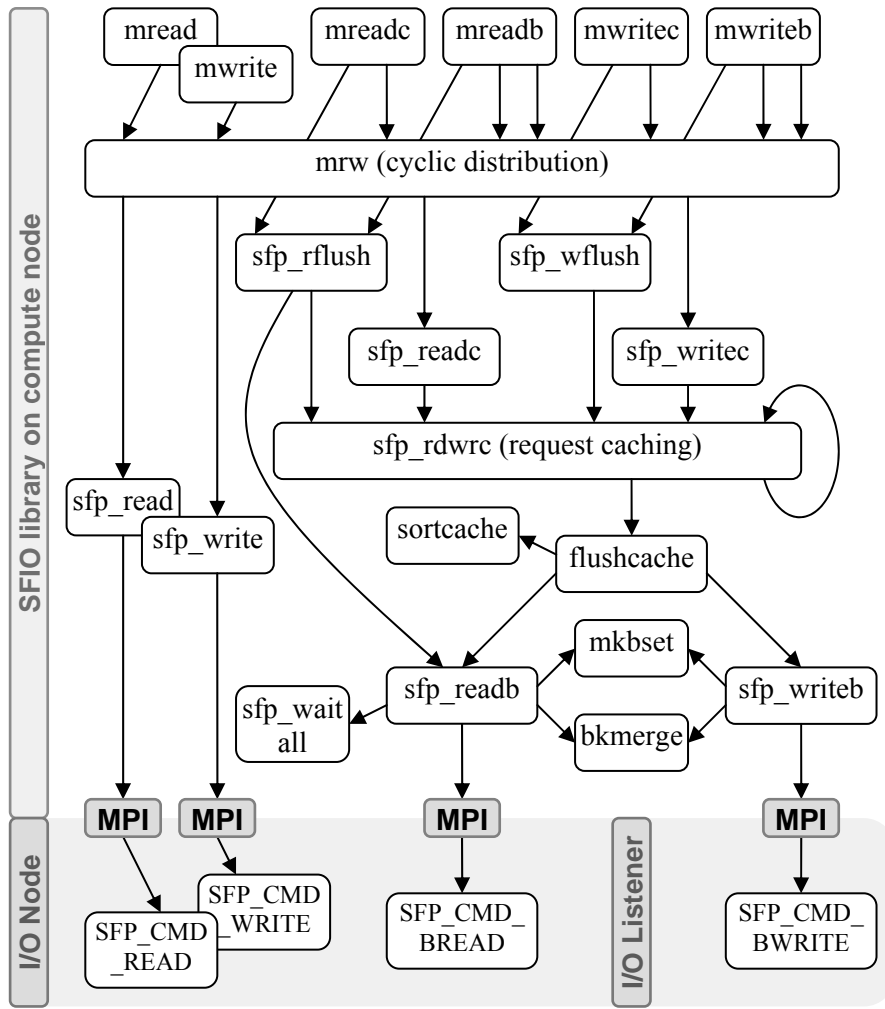


Figure 15. SFIO functional architecture

On 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.

All the *mread*, *mwrite*, *mreadc*, *mwritec*, *mreadb* and *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 for sending 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 2D cache. The 2D cache structure comprises the I/O nodes as one dimension, and the set of subfiles each I/O node is dealing with, as the second dimension. Each I/O node can have more than 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 call 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 in 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, a large list of the sub-requests has already been collected and needs to be processed. At this point the library can carry out effective optimizations in order to save network communications and disk accesses. Note that the data itself is never copied, and always remains in user space, thereby saving processor time and memory space. Three optimization procedures are carried out, before an actual transmission takes place. The requests are sorted by their offsets in the remote subfiles. This operation is carried out by the *sortcache* module. Overlapping and consecutive requests are merged into

single requests whenever possible by the *bkmmerge* 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 one to efficiently transmit over the network the data associated with many requests as a single contiguous stream. The data is transmitted or received without any memory copy at the application or library level. In a zero-copy MPI implementation relying on hardware Direct Memory Access (DMA), the entire process becomes copy-less and the actual data (even if fragmented) is transmitted directly from the user space to the network.

The transmission of data and instructions to the I/O nodes is performed by the *sfp_readb* and *sfp_writeb* functions.

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 [Fujita03]. Performance results have been measured on the Swiss-T1 machine. The Swiss-T1 supercomputer is based on Compaq Alpha Server DS20 machines and consists of 64 Alpha processors grouped in 32 nodes. Two types of network interconnect the processors, TNET and Fast Ethernet. The aggregate throughput of Fast Ethernet and the performance of SFIO on top of Fast Ethernet as a function of the number of contributing nodes are presented in Subsection 2.8.1. The aggregate raw throughput of the TNET network and the throughput of SFIO running on top of the TNET network are presented in Subsection 2.8.2.

2.8.1. Network and parallel I/O throughput when using Fast Ethernet

To obtain information about the Fast Ethernet network capabilities, throughput as a function of the 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 Swiss-T1 consists of a full crossbar switch and Figure 16 exhibits the corresponding linear scaling. Each pair of nodes (one receiver and one sender) contributes to the overall throughput through a single link capacity.

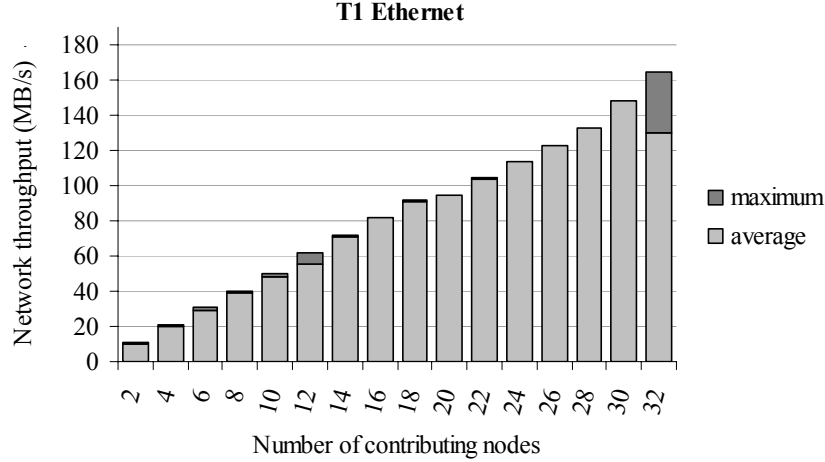


Figure 16. Aggregate throughput of Fast Ethernet as a function of the number of contributing nodes

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

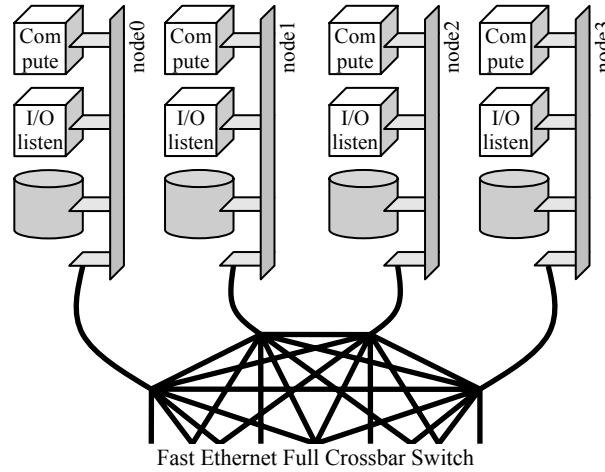


Figure 17. SFIO architecture on Swiss-T1

We consider concurrent write accesses from all compute nodes to all I/O nodes, the striped file being distributed over the disks of 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 only 200 bytes. The application's SFIO performance as a function of the number of compute and I/O nodes is measured for the Fast Ethernet network (see Figure 18). The white graph represents the average throughput and the gray graph the peak performance. Once the number of contributing nodes exceeds 12, the overall throughput decreases. The reduction in throughput

may possibly be due to a non-efficient implementation of data intensive collective operations in the 2001 version of MPICH.

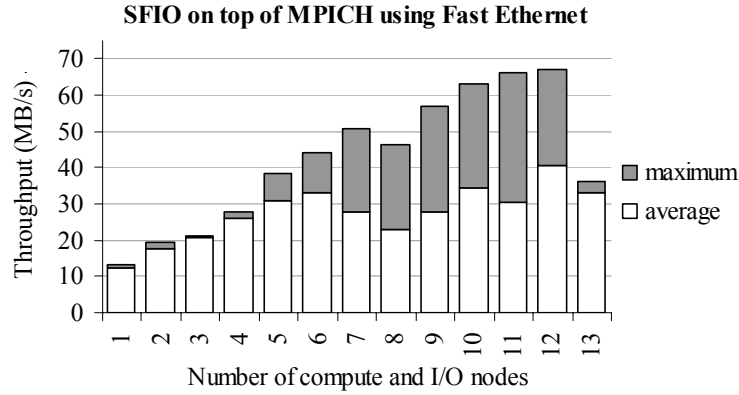


Figure 18. SFIO/MPICH all-to-all I/O performance for a 200 byte stripe size, Fast Ethernet

2.8.2. Network and parallel I/O throughput when using TNET

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].

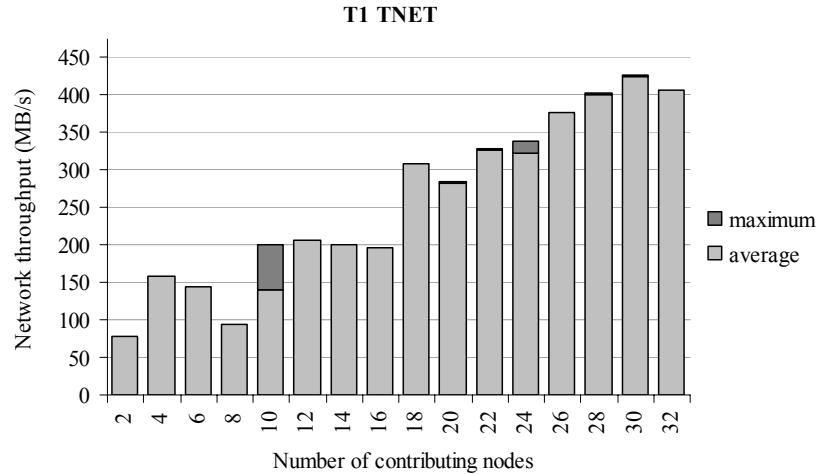


Figure 19. Aggregate throughput of TNET as a function of the number of the contributing nodes

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), the communication throughput does not increase smoothly as the number of contributing nodes increases. A significant increase in throughput occurs when the number of nodes increases from 8 to 10, from 16 to 18, and from 24 to 26.

The topology of the TNET network (Figure 20) is not equivalent to a full crossbar switch. Depending on the physical allocation of processors, contributing nodes may be grouped into clusters with limited communication capacities between them. Therefore, the overall throughput depends not only on the number of contributing nodes, but also on their particular allocation. For a given number of nodes, the overall throughput varies between a lower and an upper bound for different allocation patterns. In Subsection 3.8.1 of Chapter 3, for a given fixed number of allocated nodes we are analyzing the upper and lower bound of the underlying network's theoretical capacity depending on a particular allocation of nodes (see Figure 40).

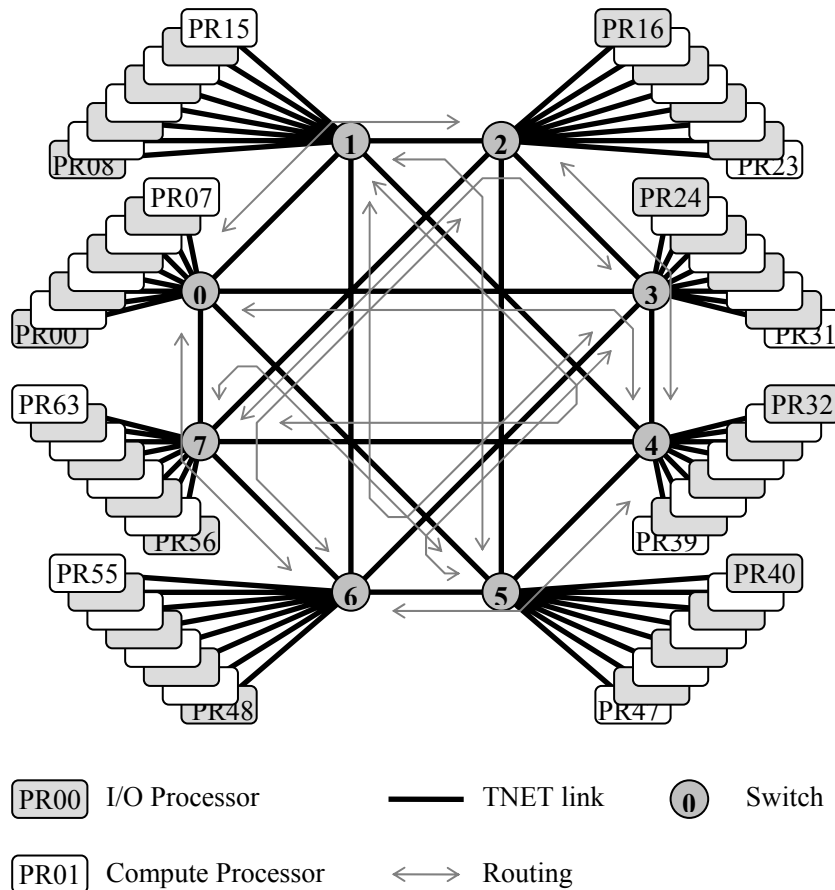


Figure 20. The Swiss-T1 network interconnection topology

The performance 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 node disks.

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 global file size ranges from 1 GB to 31 GB. 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.

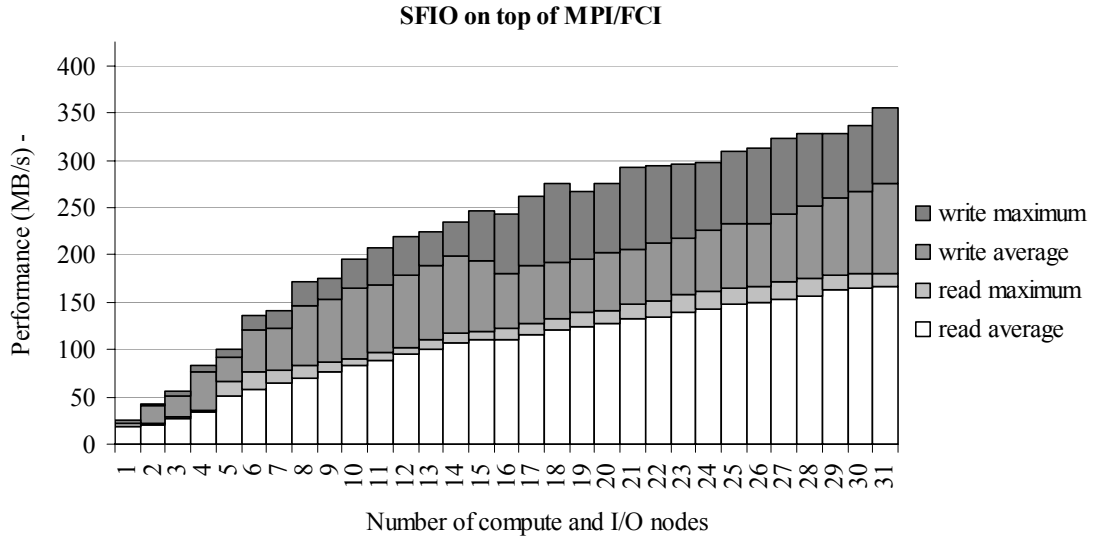


Figure 21. SFIO all-to-all I/O performance on TNET

The I/O throughput on MPI/FCI scales well when increasing the number of nodes. This configuration tests SFIO under extreme conditions in terms of the number of I/O nodes (scalability), the number of compute nodes (simultaneous concurrent accesses) and the extremely low stripe unit size (efficient optimizations of communications and disk accesses).

The speed-up may vary due to the communication topology of the TNET network (Figure 20) associated with the particular node allocation scheme. Once half of the cluster nodes are allocated, the network becomes a major bottleneck if the network transmissions are not properly

coordinated and scheduled. Network performance for collective parallel transfers is studied in Chapter 3.

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. Non-contiguous 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 to as MPI-IO. It is a rich interface with many features designed specifically for performance and portability. It supports non-contiguous accesses, non-blocking I/O and a standard data representation via MPI derived datatypes.

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 top of which the 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 both be efficient and benefit from the advanced features of the MPI-I/O design.

For specifying fragmentation patterns for different purposes, the 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.

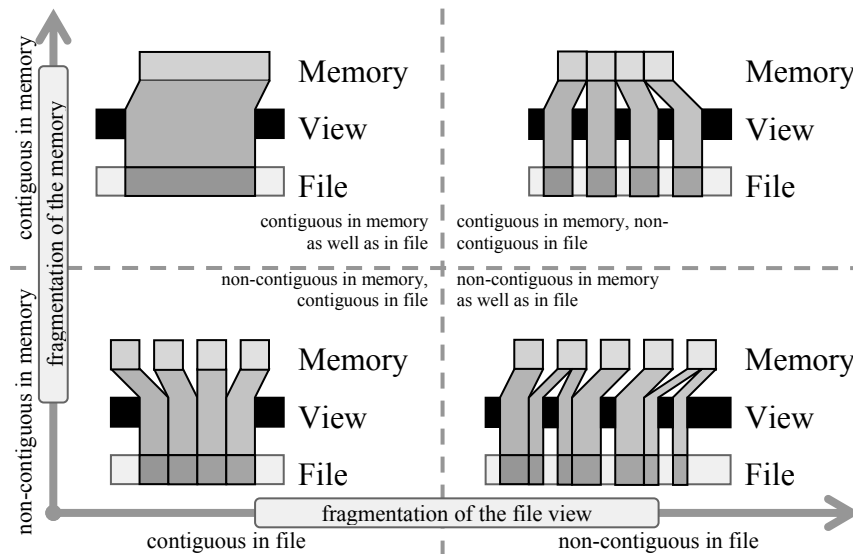


Figure 22. The use of derived datatypes in MPI-I/O interface

MPI-1 provides recursive techniques for creating datatype objects, which have an arbitrary memory data layout (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.

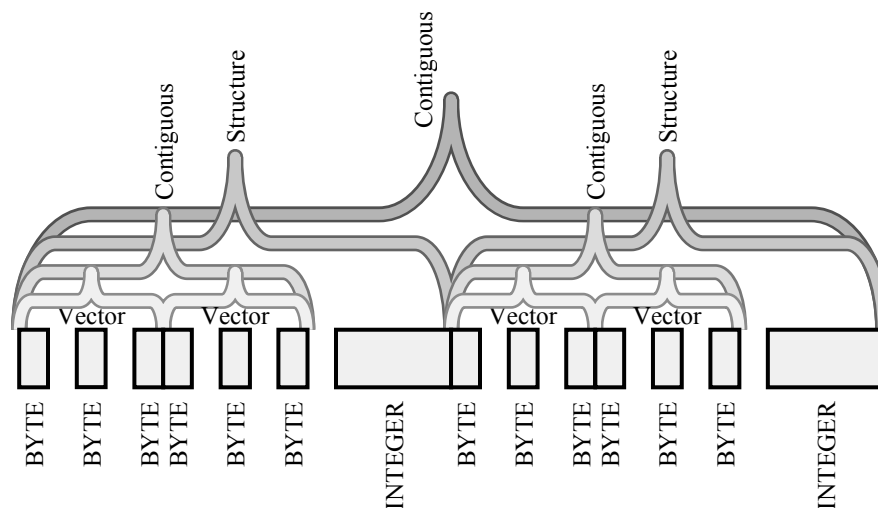


Figure 23. The recursive construction of derived datatypes in MPI
 (“Contiguous” is a derived datatype obtained by repeatedly joining another datatype which in turn can be fragmented)

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 from these opaque objects with standard MPI-1 functions (see Figure 24).

A solution for deducing the flattened fragmentation patterns (in the memory and in the file) may consist in understanding for 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 vary significantly from one implementation of MPI-1 to another and (2) the source code of a particular MPI-1 implementation is often not available or is subject to frequent updates. Our objective is to design a portable, implementation-independent solution for MPI-I/O running on top of any MPI-1 implementation.

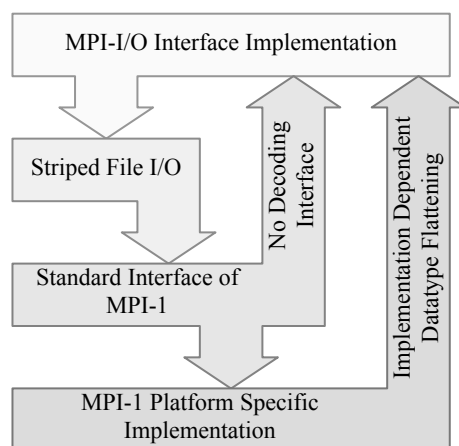


Figure 24. The MPI-I/O implementation requires a method for retrieving the fragmentation patterns of opaque MPI derived datatypes

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

The *extension* of a 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 the 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 bytes in the memory are all previously initialized with a constant value (e.g. by zeroes) referred to as “gray color”, and the network stream carries bytes all initialized by another constant value (e.g. by ones) referred to as “black color”, then analyzing the receiver’s memory after data reception will give us the necessary information on the derived datatype’s data layout.

Figure 25 shows the decoding of a derived datatype constructed in Figure 23. The size of this derived datatype is 20 bytes and its extension is 30 bytes. The sender initializes a contiguous block of the size of the derived datatype (i.e. a block of 20 bytes) with ones (appearing in black in Figure 25). The receiving side initializes with zeroes (gray color) a contiguous block of the size of the extension of the derived datatype (i.e. a block of 30 bytes). The sender transmits the bytes from its contiguous block and the receiver, using *MPI_Recv* operation, distributes the incoming data into the previously initialized memory block according to the corresponding derived datatype. Once the transmission is over, one can construct a vector of blocks representing the flattened datatype simply by reading the receiver's memory.

Derived datatypes with cross-ordered fragmentation patterns cannot be decoded with this technique. We rely on the fact that according to the MPI-2 specifications, derived datatypes used for characterizing the file view are restricted to specify only monotonically non-decreasing offsets in the file. For example, a derived datatype that specifies offsets in the order {2, 6, 5, 7, 4} cannot be used as a valid datatype for the MPI-I/O file view (see “Using MPI-2”, Section 3.3.1, p. 61, [Gropp99]).

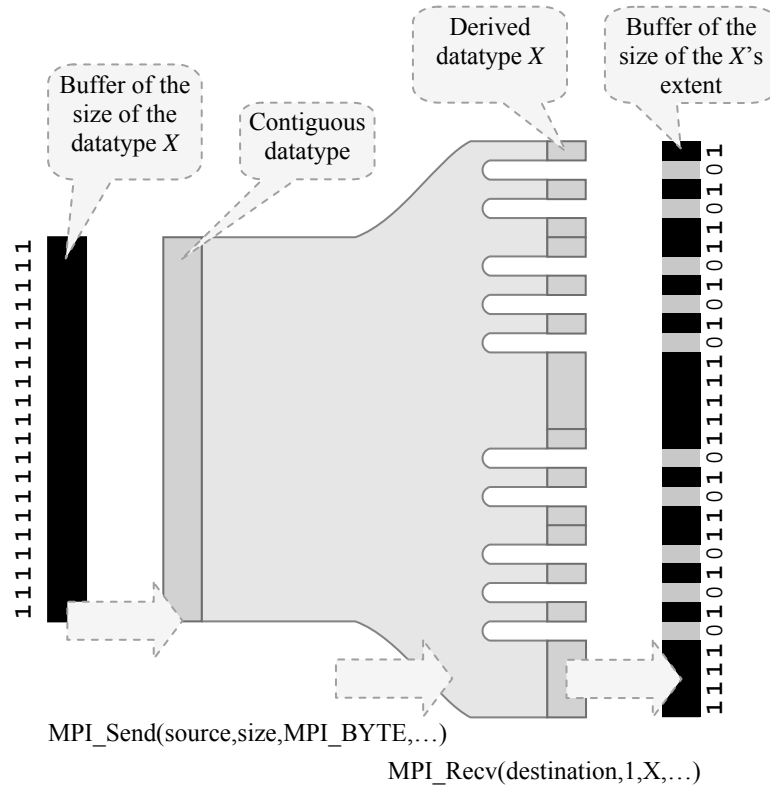


Figure 25. A reverse engineering method for discovery the fragmentation pattern of an opaque datatype built by the user

Instead of sending and receiving data it is possible to use the standard *MPI_Unpack* operation for carrying out this procedure in a single compute node. The operation *MPI_Unpack*

reads from a contiguous memory block with a size equal to the size of a single unit of a derived datatype, and writes to a contiguous block with a size equal to the extension of that derived datatype.

Once the pattern of the derived datatype is obtained, it is stored in a compact array of contiguous block lengths.

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

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

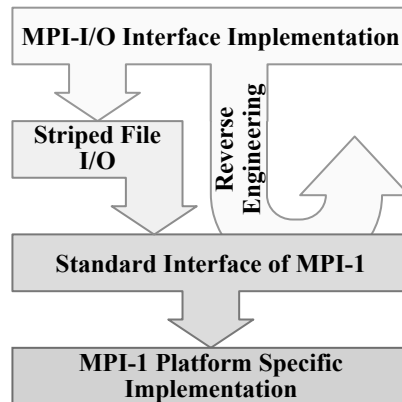


Figure 26. Isolated implementation of a portable MPI-I/O interface functional on any MPI-1 implementation

The MPI-I/O package making use of SFIO gives every MPI-1 owner an MPI-I/O interface with efficient parallel I/O facilities, without requiring any change or modification of the current MPI-1 implementation.

Section 2.10. Conclusions and recent developments in parallel input-output

For cluster computing, SFIO is a cheap alternative to specialized dedicated I/O hardware. It is a lightweight portable parallel I/O system for MPI programmers.

Since the design of SFIO, there were additional developments in 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 I/O access performance optimizations were achieved by taking into account global knowledge in the case of off-line access requests and by using prefetching relying on predictions of future access request patterns [Abawajy03], [Kallahalla02]. One may 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. The required file locking mechanisms may induce a significant performance drawback. Moreover, file locking is not always implemented in large systems. Several methods were proposed for allowing replications at I/O nodes and caching at compute nodes while maintaining the consistency of the global file by relying on orthogonal MPI level communications between compute nodes without using file locking mechanisms [Wu05B], [Coloma04]. Parallel communications between a compute node and each individual I/O node may produce a greater network throughput performance [Liu03] and [Ali05]. An overall throughput of 291 Mbps with 18 compute and I/O processors was reported [Ali05]. The throughput of SFIO (between 150 and 350 MB/s for 31 compute and I/O nodes) still remains competitive.

Regarding parallel I/O interfaces, portable implementations of the 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 support down to a 75-byte stripe size with only a negligible loss in performance. Usually the parallel I/O systems are optimized for stripe unit sizes not smaller than a few kilobytes [Thakur99B]. For balancing the I/O workload in the servers, a solution for dynamically adapting the striping factors and for dynamically distributing the data was suggested in [Ma03B].

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

The upper limit of a network's capacity is its liquid throughput. The liquid throughput corresponds to the flow of a liquid in an equivalent network of pipes. In coarse-grained networks, the aggregate throughput of an arbitrarily scheduled collective communication may be several times lower than the maximal potential throughput of the network. In wormhole and wavelength division optical networks, there is a significant loss of performance due to congestion during simultaneous transfers sharing a common communication resource. We propose to schedule the transfers of the traffic according to a schedule yielding the liquid throughput. Such a schedule, called a liquid schedule, relies on the knowledge of the underlying network topology and ensures an optimal utilization of all bottleneck links. To build a liquid schedule, we partition the traffic into time frames comprising mutually non-congesting transfers keeping all bottleneck links busy during all time frames. The search for mutually non-congesting transfers utilizing all bottleneck links is of exponential complexity. We present an efficient algorithm which non-redundantly traverses the search space. We efficiently reduce the search space without affecting the solution space. The liquid schedules for small problems (up to a hundred nodes) can be found in a fraction of a second.

Section 3.1. Introduction

3.1.1. Parallel transmissions in circuit-switched networks

It has been more than three decades since circuit-switched networks were successfully replaced by their packet-switched counterparts. In the early 1970's, this trend started by replacing data modems with connections to the X.25 network. Today, the entire concept of telephony is becoming packetized. It is commonly admitted that with fine-grained packet-switching technology, network resources are utilized more efficiently, flows are more fluid and resilient to congestion, network management is easier, and the networks can flexibly scale to large sizes.

Nevertheless, other networking approaches still based on coarse-grained circuit-switching have been emerging. These approaches offer low latencies, which are not attainable with packet switching technology. In addition, circuit switching is of importance for optical communications.

Examples of circuit switching networks are wormhole and cut-through routing (e.g. MYRINET [Boden95], InfiniBand, [Steen05], [InfiniBand], [Reinemo06], [Bermudez06]) and optical Wavelength Division Multiplexing (WDM). In contrast to packet switching, in wormhole and optical switching networks the number of network hops separating the end nodes has almost no impact on the communication latency. With respect to optical networks, due to the lack of optical memory, packet switching in optical networks does not exist at all today (at least not commercially).

All coarse-grained circuit-switching networks suffer from a common problem: inter-blocking of transfers and jamming of large indivisible messages occupying intersecting resources of the network (e.g. lightpaths of a given wavelength). Several parallel multi-hop transmissions cannot share the same link resource simultaneously. In contrast to the fluidity and resiliency of packet-switching, in coarse-grained circuit-switching networks, hard and complex interlocking contentions arise when the network topology grows and the load increases.

In WDM optical networks, a single fiber can carry several wavelengths: approximately 80 in standard WDM, 160 in DWDM, and 1000 in research prototypes [Kartalopoulos00]. However the contentions are still present, because wavelengths are typically conserved along the whole communication path. There is no switching from one wavelength to another in the middle of the network. The new wavelengths are simply increasing the network capacity. In wormhole switching, when the head of the message is blocked at an intermediate switch (due to contention), the transmission stays strung over the network, potentially blocking other messages.

WDM wavelength routing is briefly introduced in Subsection 3.2.2 and wormhole routing is introduced in Subsection 3.2.1.

3.1.2. Hardware solutions

In optical and wormhole switching the problem of contentions can be partially or fully solved at the hardware level.

For example, the optical switches of the network may be equipped with the capability to change the incoming wavelengths (not only to switch across the ports, i.e. to control the direction of the light, but also to change the wavelength). Wavelength interchange (changing of colors) requires expensive optical-electric (O/E) and electro-optical (E/O) conversions. Without O/E/O conversions, when the signal is constantly maintained in the optical domain, cost-effective optical networks can be built by relying only on switching by microscopic mirrors, using inexpensive Micro Electro-Mechanical Systems (MEMS). In addition, O/E/O conversions necessarily induce additional delays.

Regarding wormhole routing, the switches typically need only to buffer the tiny pieces of the message (flits) that are sent between the switches. However, the switches may be equipped with memories large enough to store the entire message (according to the estimation of the message size in the network). Thus, when the head of the message is blocked, the switch lets the tail continue, accumulating the whole message into a single switch. This hardware extension of wormhole routing is called *cut-through switching*. Storing the messages solves the contention problem only partially but requires a substantial increase of the switch's memory, up to multiples of the largest message size (depending on the number of ports). Virtual cut-through switching is yet another hardware extension, wherein the link is divided (similarly to WDM) into a certain number of virtual links sharing the capacity of the physical link.

In coarse-grained circuit switching the hardware solutions of contention-avoidance require costly modifications to hardware (e.g. O/E/O conversion in optical switching or substantial memory in wormhole switches) and often provide only partial solutions. The hardware solutions also reduce the benefits of low latency (for example in cut-through routing, the entire messages are stored in the switches).

3.1.3. Liquid scheduling - an application level solution

In wormhole routing, by keeping the architecture simple, switches with a large number of physical ports can be implemented in single chips at a very low cost. I propose liquid scheduling as an application level method for obtaining the network's highest overall throughput. The scheduling is performed at the edge nodes and requires no specific hardware solutions. Synchronization and coordination of edge nodes is required.

Numerous applications relying on coarse-grained circuit-switched networks require an efficient use of network resources for collective communications. Such applications include parallel acquisition and distribution of multiple video streams [Chan01], [Sitaram00], switching of simultaneous voice communication sessions [H323], [EWS04], [SIP], and high energy physics, where particle collision events need to be transmitted from a large number of detectors and filters to clusters of processing nodes [CERN04].

Liquid scheduling can be used in Optical Burst Switching (OBS) by the edge IP routers for an efficient utilization of the capacities of an interconnecting optical cloud (all-optical network providing interconnection for the edge routers).

3.1.4. Overview of liquid scheduling

The aggregate throughput of a collective communication pattern (transmissions between pairs of end nodes) depends on the underlying network topology and the routing. The amount of data that has to pass across the most loaded links of the network, called bottleneck links, gives their utilization time. The total size of the traffic divided by the utilization time of one bottleneck link gives an estimation of the *liquid throughput*, which corresponds to the flow

capacity of a non-compressible fluid in a network of pipes [Melamed00]. Both in wormhole switching networks and WDM optical networks, due to possible link or wavelength allocation conflicts, not just any combination of transfer requests may be carried out simultaneously. The objective is to minimize the number of timeslots and/or wavelengths required to carry out a given set of transfer requests. Each transfer shall be allocated to one (and only one) time frame, such that no pair of transfers allocated to the same time frame uses a common resource (link, wavelength). The liquid scheduling problem is introduced and mathematically defined in Section 3.3 and Section 3.4.

The liquid scheduling problem cannot be solved in polynomial time. Solving the problem by Mixed Integer Linear Programming (MILP) [CPLEX02], [Fourer03] requires very long computation times (see Appendix C). Solving the problem by applying a heuristic graph coloring algorithm provides suboptimal solutions in short time. The throughputs corresponding to the heuristic solutions of the graph coloring problem are often 10% to 20% lower than the liquid throughput [Gabrielyan03] (see Appendix B). In the present contribution we propose an exact method for computing liquid schedules, which is fast enough for real time scheduling of traffic on small scale networks comprising up to a hundred nodes.

Section 3.2 gives a brief overview of the architectures of the optical and wormhole switching networks. Section 3.3 and Section 3.4 introduce the liquid scheduling problem. Section 3.5, Section 3.6 and Section 3.7 present the liquid schedule construction algorithm. In Section 3.8 we present for many network traffic patterns their overall communication throughputs when carried out according to both liquid schedules and to topology-unaware schedules.

Section 3.2. Applicable networks

This section briefly introduces two coarse-grained switching concepts: wormhole switching (Subsection 3.2.1) and lightpath routing (Subsection 3.2.2). The advantages of applying liquid schedules are discussed for both types of networks.

3.2.1. Wormhole switching

Wormhole routing is used in many High Performance Computing (HPC) networks. In wormhole routing, the links lying on the path of a message are kept occupied during the transmission of that message. Unlike packet switching (or store-and-forward switching) where each network packet is present at an intermediate router [Ayad97], wormhole switching [Liu01], [Dvorak05] transmits a message as a “worm” propagating itself across intermediate switches. The message “worm” is a continuous stream of bits which make their way through successive

switches. In a wormhole switching network [Duato99], [Shin96], [Rexford96], [Colajanni99], [Dvorak05] a message entering into the network is continuously broken up into small parts of equal size called flits (standing for flow-control digits). These flits are streamed across the network. All the flits of a packet follow the same path. The head flit contains the routing header for the entire message. As soon as a switch on the path of a message receives the head flit, it can direct the incoming flow to the corresponding outgoing link. If the message encounters a busy outgoing link, the wormhole switch stalls the message in the network along the already established path until the link becomes available. Occupied channels are not released. A channel is released only when the last tail flit of the message has been transmitted. Thus each link lying on the path of the message is kept occupied during the whole transmission time of a message. In virtual cut-through (VCT) networks, if the message encounters a busy outgoing link, the entire message is buffered in the router and previously allocated portions of the message path are released. A VCT switch has enough memory to store nearly as many messages as its number of ports. A simple wormhole switch architecture capable of only pipelining the messages requires no more than a very small buffer, regardless of the size of the largest possible message in the network. It enables a cost effective implementation of wormhole switches with a large number of ports on a single chip [Yocum97]. The ability of VCT switches to buffer large messages increases their cost substantially.

Compared to store and forward switching, wormhole switching considerably decreases the latency of message transmission across multiple routers. Wormhole switching makes the latency insensitive to the distance between the end nodes. Most contemporary research and high-performance commercial multi-computers use some form of wormhole or cut-through networks, e.g. Myrinet [Boden95], fat tree interconnections for clusters [Petrini01], [Petrini03], [Quadrics], InfiniBand [InfiniBand], [Steen05], [Reinemo06], [Bermudez06] and Tnet [Horst95], [Brauss99B].

Due to blocked message paths, wormhole switching quickly saturates as the load increases. The aggregate throughput can be considerably lower than the liquid throughput of the network. The rate of network congestion depends on the order in which a given set of message transfers is carried out. Liquid scheduling enables the partitioning of the transfers so as to avoid simultaneous transmissions of congesting messages.

3.2.2. Optical networks

In optical networks, data is transferred by lightpaths. Lightpaths are end to end optical connections from a source node to a destination node. In Wavelength Division Multiplexing (WDM) optical networks, a lightpath is typically established over a single wavelength (color) along the whole path. Different lightpaths in a WDM wavelength-routing network can use the same wavelength as long as they do not share any common link. Figure 27 shows an example of an optical wavelength-routing network. Switches of the optical network are called Optical Cross Connects (OXC). An OXC switches wavelengths from one port to another, usually without

changing the color [Ramaswami97], [Stern99]. The Optical Line Terminal (OLT) multiplexes multiple wavelengths into a single fiber and de-multiplexes a set of wavelengths from a single fiber into separate fibers. Often the OLT units are integrated with OXC.

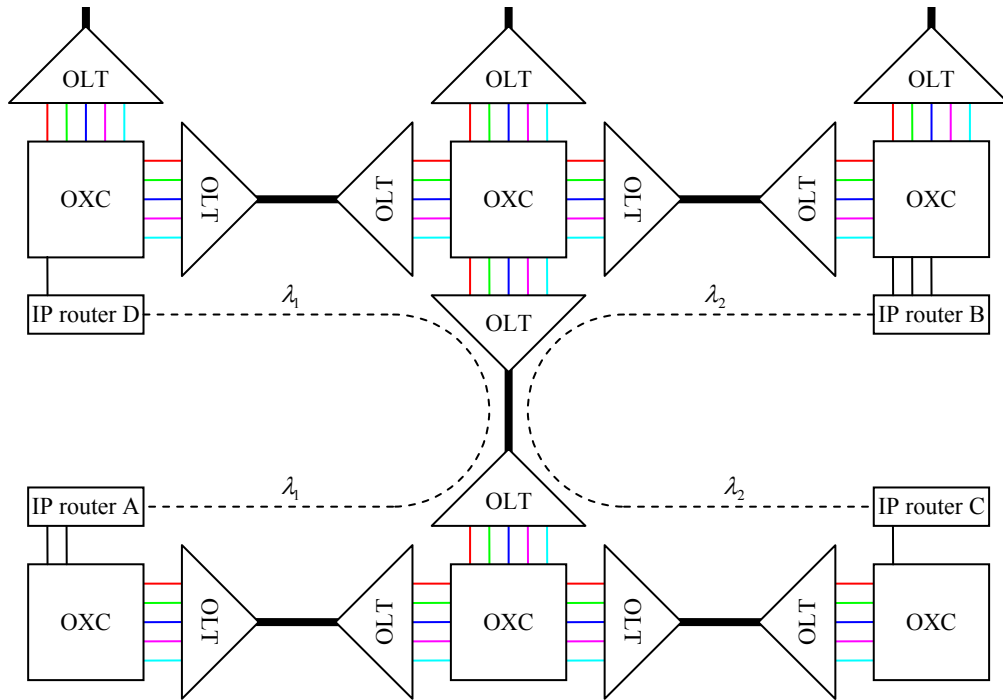


Figure 27. Wavelength routing in the optical layer

End nodes (or edge nodes) of an optical network (also called an optical cloud) are IP routers, SONET terminals, or ATM switches. They are plugged into OXC switches (as shown in Figure 27). In a simple design the end node can also be inserted into a fiber (statically) via an Optical Add/Drop Multiplexer (OADM). The purpose of the optical cloud is to provide lightpaths between the terminal edge nodes, for example between IP routers (as shown in Figure 27). The lightpaths between the end nodes can either be established permanently, or provided dynamically on demand.

Relatively inexpensive OXC switches can be implemented by an array of microscopic mirrors, build with Micro Electro-Mechanical Systems (MEMS). These switches only re-direct the incoming wavelengths to appropriate outgoing ports, without converting the color. They are called Wavelength-Selective Cross-Connect (WSXC). Changing of the wavelength is possible through Optical/Electro/Optical (O/E/O) conversions. Optical switches providing wavelength conversion features are called Wavelength-Interchanging Cross-Connects (WIXC). WIXC switches do both space switching and wavelength conversion.

When using WIXC switches, the lightpaths may be converted from one wavelength to another along their route. However from the optical network design point of view, it is essential

to keep transmissions in the optical domain as long as possible, i.e. to be able to provide the required services using only inexpensive WSXC switches.

Wavelength continuity (the fact that the basic optical transmission channel remains on a fixed wavelength from end to end) is the main constraint affecting the scalability of networks built with WSXC switches only.

For example assuming only WSXC switches in Figure 27, two connections from IP router *A* to *B* and from *C* to *D* must either be established on two different wavelengths λ_1 and λ_2 , or must be scheduled in different timeslots.

Given that any lightpath must be assigned the same wavelength on all the links it traverses and that two lightpaths traversing a common link must be assigned different wavelengths, the *wavelength assignment problem* requires minimizing the number of wavelengths needed for establishment of the required end to end connections. In this domain, the wavelength assignment problem is commonly solved by solving the corresponding congestion graph coloring problem [Bermond96], [Caragiannis02]. The vertices of the graph represent the lightpaths and two vertices are connected if the corresponding lightpaths are sharing a common link. The graph coloring problem requires the coloring of all vertices using a minimal number of colors such that two connected vertices always have different colors. Graph coloring is an NP-complete problem. Its solutions are generally based on heuristic methods.

Liquid scheduling is an efficient method for assigning transmissions a minimal number of lightpaths or timeframes. If a liquid schedule exists, the solution of the liquid scheduling algorithm corresponds to the optimal solution of the graph coloring algorithm. Our algorithm does not associate the set of transfers with a graph. It considers not only the congestion between pairs of transfers (congestion graph), but also the set of links occupied by each transfer. This permits the building of liquid schedules relatively quickly for networks comprising up to a hundred nodes. The corresponding congestion graphs comprise thousands of vertices. The heuristic graph coloring algorithms often propose solutions requiring more timeframes than the number of timeframes allocated by our liquid scheduling algorithm. The comparison of the liquid scheduling algorithm with a heuristic graph coloring method is given in Appendix B.

Application of liquid schedules in the optical domain assumes a collaboration of the edge nodes and therefore an appropriate signaling layer. Optical Burst Switching (OBS) is an example where the collaboration of the edge nodes is assumed and the application of liquid schedules may significantly improve the overall throughput of the optical cloud [Qiao99], [Turner99], [Turner02]. In the case of continuous incoming IP traffic, the filled buffers of the edge nodes are repeatedly emptied by applying liquid scheduling. For the buffered data, the liquid schedule finds the minimal number of partitions comprising non-congesting lightpaths. The same wavelength is allocated to all transfers of a partition. The number of wavelengths available in the network may not suffice for all partitions found by the liquid schedule. In such a case, when all transfers cannot be carried out within a single round (timeslot), new rounds (with a new set of wavelengths) are allocated until all transfers are carried out. Regardless of the

number of wavelengths available in the network, liquid scheduling minimizes the total number of required rounds.

Local strategies for avoiding congestion rely on an admission control mechanism [Jagannathan02], [Mandjes02] or on feed-back and flow control based mechanisms regulating the sending nodes' data transmission rate [Maach04], [Chiu89], [Loh96]. These mechanisms avoid congestion by rejecting the extra traffic. Local decision based strategies utilize only a fraction of the network's overall capacity. The global liquid scheduling strategy ensures that the network's potential capacity is used efficiently.

Section 3.3. The liquid scheduling problem

In our model, we neglect network latencies, we consider a constant message (or packet) size, we assume an identical link throughput for all links, and we assume a static routing scheme.

Consider a simple network example consisting of ten end nodes $t_1 \dots t_5$, $r_1 \dots r_5$, two wormhole cut-through switches s_a , s_b and twelve unidirectional links $l_{t1} \dots l_{t5}$, $l_{r1} \dots l_{r5}$, l_{ab} , l_{ba} all having identical throughputs (see Figure 28). Assume that the nodes $t_1 \dots t_5$ are only transmitting and the nodes $r_1 \dots r_5$ are only receiving. The routing is straight-forward, e.g. a message from t_4 to r_3 traverse links l_{t4} , l_{ba} and l_{r3} , a message from t_1 to r_2 uses only links l_{t1} and l_{r2} .

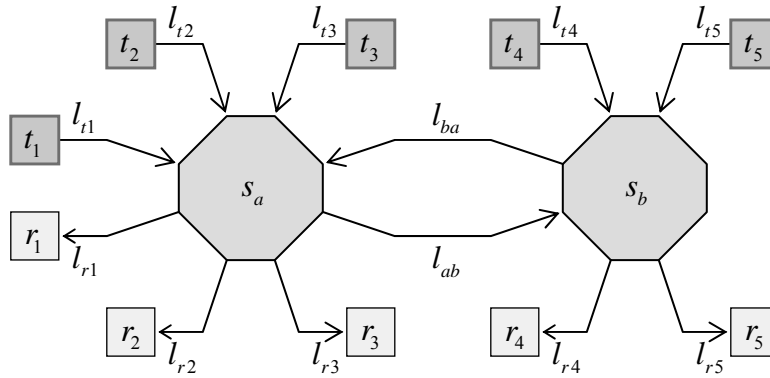



Figure 28. Example of a simple network

For demonstration purposes we represent the transfers of the network of Figure 28, symbolically via small pictograms highlighting the links used by the transfer. For example the transfer from t_4 to r_3 is symbolically represented as $\rightarrow \text{---} \text{---} \text{---} \rightarrow$, the transfer from t_1 to r_2 as $\rightarrow \text{---} \text{---} \rightarrow$. We may also represent a set of two or more simultaneous transfers by a pictogram highlighting

all occupied links. For example, a simultaneous transmission of the two previous transfers (from t_4 to r_3 and from t_1 to r_2) is represented as .

We assume that all messages have identical sizes [Naghshineh93]. Let each sending node have messages to be transmitted to each receiving node. There are therefore 25 transfers to carry out. The corresponding pictograms for these 25 transfers are shown in Figure 29.

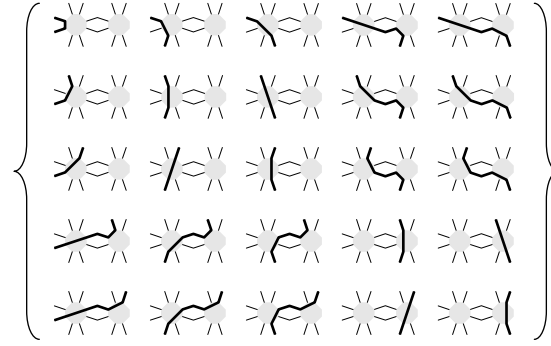









Figure 29. The pictograms representing the 25 transfers from all sending nodes to all receiving nodes of the network of Figure 28







Accordingly, each of the ten links $t_1 \cdots t_5$, $r_1 \cdots r_5$ must carry 5 transfers, but the two links l_{ab} , l_{ba} must each carry 6 transfers. Therefore, for the 25 transfers to be carried out, the links l_{ab} , l_{ba} are the network bottlenecks and have the longest active time. If the duration of the whole communication is as long as the active time of the bottleneck links, we say that the collective communication reaches its liquid throughput. In that case the bottleneck links are obviously kept busy at all times during the communication. Assume in this example a single link throughput of 1Gbps. The liquid throughput offered by the network is $(25/6) \cdot 1Gbps = 4.17Gbps$.

The *liquid throughput* of a traffic X is the ratio $\#(X)/\Lambda(X)$ multiplied by the single link throughput (identical for all links), where $\#(X)$ is the total number of transfers and $\Lambda(X)$ is the number of transfers carried out by one bottleneck link (the messages have identical sizes).

Now let us see if the order in which the transfers are carried out in this network has an impact on the overall communication throughput. A straight-forward schedule allowing one to carry out these 25 transfers is the round-robin schedule. At first, each transmitting node sends the message to the receiving node staying in front of it, then to the receiving node staying at the next position, etc. Such a round robin schedule consists of 5 phases.

The transfers of the first , second  and the fifth  phases of the round-robin schedule may be carried out simultaneously, but the third phase $\{\text{pictogram 1}, \text{pictogram 2}, \text{pictogram 3}\}$ and the forth phase $\{\text{pictogram 4}, \text{pictogram 5}, \text{pictogram 6}, \text{pictogram 7}, \text{pictogram 8}\}$ contain congesting transfers (we say that two transfers are *in congestion* if they cannot be carried out simultaneously due to a common resource). For example, the two transfers of the third phase:

 and , cannot be carried out at the same time since they are trying to simultaneously use link l_{ab} (see Figure 28). Similarly, two other transfers of the third phase ,  are also in congestion, since they are simultaneously competing for the same link l_{ba} . The forth phase of the round-robin schedule has two pairs of congesting transfers as well. Each of these phases cannot be carried out in less than two time frames and therefore the whole schedule lasts 7 time frames and not 5 (the number of phases in the round-robin schedule). Five timeframes would have been sufficient if there were additional capacities (links) between the switches s_a and s_b . The throughput of the collective communication carried out according to the round-robin schedule is $25/7 = 3.57$ messages per time frame, or $(25/7) \cdot 1Gbps = 3.57Gbps$, which is below the liquid throughput of $4.17Gbps$.

The 25 transfers can be scheduled within a fewer number of timeframes. The following schedule $\{\text{}, \text{}, \text{}, \text{}, \text{}, \text{}\}$ carries out the 25 transmissions in 6 timeframes. Each timeframe consists of 3 to 5 non-congesting transfers. The whole schedule yields the liquid throughput of $4.17Gbps$.

In the following sections we present algorithms permitting the construction of liquid schedules for arbitrary traffic patterns on arbitrary network topologies.

Section 3.4. Definitions

The method we propose allows us to efficiently build liquid schedules for non-trivial network topologies. Thanks to liquid schedules we may considerably increase the collective data exchange throughputs, compared with traditional topology unaware schedules such as round-robin or random schedules.

The present section introduces the definitions that will be further used for describing the liquid schedule construction method.

A single “point-to-point” transfer is represented by the set of communication links forming the network path between one transmitting and one receiving node according to the given routing. Note that we will be limiting ourselves to data exchanges consisting of identical message sizes.

We therefore define in our mathematical model a *transfer* as a set of all links lying on the path between one sending and one receiving node. A *traffic* is a set of transfers (i.e. a collective data exchange).

According to the definition of traffic, Figure 30 shows the traffic pattern of Figure 29 (corresponding to a collective data exchange carried out on the network of Figure 28) in the new set-represented notation. The traffic of Figure 30 represents a scenario, wherein each

transmitting node (the nodes $t_1 \cdots t_5$ at the top of Figure 28) sends one message to each receiving node (the nodes $r_1 \cdots r_5$ at the bottom of Figure 28). Any other collective exchange comprising transfers between possibly overlapping sets of sending and receiving nodes (a node obviously can receive and transmit) is a valid traffic according to our definition.

$$\left\{ \begin{array}{l} \{l_{t1}, l_{r1}\}, \{l_{t1}, l_{r2}\}, \{l_{t1}, l_{r3}\}, \{l_{t1}, \mathbf{l}_{ab}, l_{r4}\}, \{l_{t1}, \mathbf{l}_{ab}, l_{r5}\}, \\ \{l_{t2}, l_{r1}\}, \{l_{t2}, l_{r2}\}, \{l_{t2}, l_{r3}\}, \{l_{t2}, \mathbf{l}_{ab}, l_{r4}\}, \{l_{t2}, \mathbf{l}_{ab}, l_{r5}\}, \\ \{l_{t3}, l_{r1}\}, \{l_{t3}, l_{r2}\}, \{l_{t3}, l_{r3}\}, \{l_{t3}, \mathbf{l}_{ab}, l_{r4}\}, \{l_{t3}, \mathbf{l}_{ab}, l_{r5}\}, \\ \{l_{t4}, \mathbf{l}_{ba}, l_{r1}\}, \{l_{t4}, \mathbf{l}_{ba}, l_{r2}\}, \{l_{t4}, \mathbf{l}_{ba}, l_{r3}\}, \{l_{t4}, l_{r4}\}, \{l_{t4}, l_{r5}\}, \\ \{l_{t5}, \mathbf{l}_{ba}, l_{r1}\}, \{l_{t5}, \mathbf{l}_{ba}, l_{r2}\}, \{l_{t5}, \mathbf{l}_{ba}, l_{r3}\}, \{l_{t5}, l_{r4}\}, \{l_{t5}, l_{r5}\} \end{array} \right\}$$

Figure 30. Example of a traffic comprising 25 transfers of Figure 29 (over the network of Figure 28) each represented as set of links

A link l is *utilized* by a transfer x if $l \in x$. A link l is utilized by a traffic X if l is utilized by a transfer of X . Two transfers are in *congestion* if they share a common link, i.e. if their intersection is not empty.

A *simultaneity* of a traffic X is a subset of X consisting of mutually non-congesting transfers. Intersection of any two members of a simultaneity is always empty. A transfer is in congestion with a simultaneity if the transfer is in congestion with at least one member of the simultaneity. A simultaneity of a traffic is *full* if all transfers in the complement of the simultaneity in the traffic are in congestion with that simultaneity. A simultaneity of a traffic obviously can be carried out within one time frame (the time to carry out a single transfer).

The *load* $\lambda(l, X)$ of a link l in a traffic X is the number of transfers in X using link l .

$$\lambda(l, X) = \# \left(\{x \in X \mid l \in x\} \right) \quad (1)$$

The *duration* $\Lambda(X)$ of a traffic X is the maximal value of the load among all links involved in the traffic.

$$\Lambda(X) = \max_{\left\{ l \in \bigcup_{x \in X} x \right\}} \lambda(l, X) \quad (2)$$

The links having maximal load values, i.e. when $\lambda(l, X) = \Lambda(X)$, are called *bottlenecks*. In the example of the traffic of Figure 30, all bottleneck links are marked in bold. The *liquid throughput* of a traffic X is the ratio $\#(X) / \Lambda(X)$ multiplied by the single link throughput, where $\#(X)$ is the number of transfers in the traffic X .

$$t_{liquid} = \frac{\#(X)}{\Lambda(X)} \cdot t_{link} \quad (3)$$

We define a simultaneity of X as a *team* of X if it uses all bottlenecks of X . A liquid schedule must comprise only teams since all bottleneck links must be kept busy all the time. A team of X is *full* if it is a full simultaneity of X . Intuitively, there is a greater chance to successfully assemble a liquid schedule that covers all transfers of the initial traffic, if one considers only full teams during the construction, instead of also considering all possible non-full teams (see Subsection 3.7.4).

Let $\mathfrak{R}(X)$ be the set of all full simultaneousities of X . Let $\mathfrak{T}'(X)$ and $\mathfrak{T}(X)$ be the sets of all teams and the set of all full teams of X respectively. By definition, $\mathfrak{T}(X) \subset \mathfrak{R}(X)$, $\mathfrak{T}(X) \subset \mathfrak{T}'(X)$, and the intersection of all teams with all full simultaneousities is the set of all full teams:

$$\mathfrak{T}(X) = \mathfrak{T}'(X) \cap \mathfrak{R}(X) \quad (4)$$

In order to form liquid schedules, we try to schedule transfers in such a way that all bottleneck links are always kept busy. Therefore we search for a liquid schedule by trying to assemble non-overlapping teams carrying out all transfers of the given traffic, i.e. we partition the traffic into teams. To cover the whole solution space we need to generate all possible teams of a given traffic. This is an exponentially complex problem. It is therefore important that the team traversing technique be non-redundant and efficient, i.e. that each configuration be evaluated once and only once, without repetitions.

Section 3.5. Obtaining full simultaneousities

To obtain all full teams, we first optimize the retrieval of all simultaneousities and then use that algorithm to retrieve all full teams.

Recall that in a traffic X , any mutually non-congesting combination of transfers is a simultaneity. A full simultaneity is a combination of non-congesting transfers taken from X , such that its complement in X contains only transfers congesting with that simultaneity.

We can categorize full simultaneousities according to the presence or absence of a given transfer x . A full simultaneity is x -positive if it contains transfer x . If it does not contain transfer x , it is x -negative. Thus the entire set of all full simultaneousities $\mathfrak{R}(X)$ is partitioned into two non-overlapping halves: an x -positive and an x -negative subset of $\mathfrak{R}(X)$. For example, if y is another transfer, the set of x -positive full simultaneousities may be further partitioned into y -positive and y -negative subsets. Iterative partitioning and sub-partitioning permits us to recursively traverse the whole set of all full simultaneousities $\mathfrak{R}(X)$, one by one, without repetitions.

The rest of this section describes in detail the algorithm for sequentially traversing all possible distinct full simultaneousities.

3.5.1. Using categories to cover subsets of full simultaneities

Let us define a *category* of full simultaneities of X as an ordered triplet (*includer*, *depot*, *excluder*), where the includer is a simultaneity of X (not necessarily full), the excluder contains some transfers of X non-congesting with the includer, and the depot contains all the remaining transfers non-congesting with the includer.

We define categories in order to represent collections of full simultaneities from the set of all full simultaneities $\mathfrak{R}(X)$. The includer and excluder of a category are used as constraints for determining the corresponding full simultaneities.

We therefore say that a full simultaneity is *covered* by a category R , if the full simultaneity contains all the transfers of the category's includer and does not contain any transfer of the category's excluder. Consequently, any full simultaneity covered by a category is the category's includer together with some transfers taken from the category's depot. The collection of all full simultaneities of X covered by a category R is defined as the *coverage* of R . We denote the coverage of R as $\Phi(R)$. By definition, $\Phi(R) \subset \mathfrak{R}(X)$.

Transfers of a category's includer form a simultaneity (not full). By adding different variations of transfers from the depot, we may obtain all possible full simultaneities covered by the category.

The category $(\emptyset, X, \emptyset)$ is a *prim-category*. A prim-category covers all full simultaneities of X :

$$\Phi(\emptyset, X, \emptyset) = \mathfrak{R}(X) \quad (5)$$

Since the includer and excluder of the prim-category are empty, the prim-category represents no restrictions on full simultaneities. Therefore any full simultaneity is covered by the prim-category (or in other words, all full simultaneities contain the empty includer of the prim-category and do not contain a transfer of the excluder, because it is empty).

3.5.2. Fission of categories into sub-categories

By taking an arbitrary transfer x from the depot of a category R , we can partition the coverage of R into x -positive and x -negative subsets. The respective x -positive and x -negative subsets of the coverage of R are coverages of two categories derived from R : a positive subcategory and a negative subcategory of R .

The positive subcategory R_{+x} is formed from the category R by adding transfer x to its includer, and by removing from its depot and excluder all transfers congesting with x . Since transfers congesting with x are naturally excluded from a full simultaneity covered by R_{+x} , we may safely remove them from the excluder (and therefore avoid redundancy in the exclusion constraint). The negative subcategory R_{-x} is formed from the category R by simply moving the

transfer x from its depot to its excluder. The replacement of a category R by its two sub categories R_{+x} and R_{-x} is defined as a *fission* of the category.

By the definition of fission, the two sub-categories resulting from the fission are also valid categories, according to the definition of category.

Figure 31 and Figure 32 show a fission of a category into positive and negative sub categories.

$$R = \left(\begin{array}{ccc} \{\Theta_1\} & \{\Xi_1, x, \Xi_2, \Theta_2\} & \{\Xi_3, \Theta_3\} \\ \text{includer} & \text{depot} & \text{excluder} \end{array} \right)$$

Figure 31. An initial category before fission, where symbol Ξ , represents any transfer that is in congestion with x and symbol Θ represents any transfer which is simultaneous with x

Figure 31 shows an example of an initial category R and Figure 32 shows the resulting two sub categories obtained from it by a fission relative to a transfer x taken from the depot. The transfers $\Xi_1 \cdots \Xi_3$ congest with transfer x , and the transfers $\Theta_1 \cdots \Theta_3$ are simultaneous with x .

$$R = \begin{cases} R_{+x} = \left(\begin{array}{ccc} \{\Theta_1, x\}, & \{\Theta_2\}, & \{\Theta_3\} \\ \text{includer} & \text{depot} & \text{excluder} \end{array} \right) \\ R_{-x} = \left(\begin{array}{ccc} \{\Theta_1\}, & \{\Xi_1, \Xi_2, \Theta_2\}, & \{\Xi_3, \Theta_3, x\} \\ \text{includer} & \text{depot} & \text{excluder} \end{array} \right) \end{cases}$$

Figure 32. Fission of the category of Figure 31 into its positive and negative sub categories.

The coverage of R is partitioned by the coverages of its sub categories R_{+x} and R_{-x} , i.e. the coverage of a category is the union of coverages of its sub categories (equation (6)), and the coverages of the sub categories have no common transfers (equation (7)).

$$\Phi(R_{+x}) \cup \Phi(R_{-x}) = \Phi(R) \quad (6)$$

and

$$\Phi(R_{+x}) \cap \Phi(R_{-x}) = \emptyset \quad (7)$$

3.5.3. Traversing all full simultaneities by repeated fission of categories

A *singular* category is a category that covers only one full simultaneity. That full simultaneity is equal to the includer of the singular category. The depot and excluder of a singular category are empty.

We apply the binary fission to the prim-category (equation (5)) and split it into two categories. Then, we apply the fission to each of these categories. Repeated fission increases the number of categories and narrows the coverage of each category. Eventually, the fissions will lead to singular categories only, i.e. categories whose coverage consists of a single full simultaneity. Since at each stage we have been partitioning the set of full simultaneities, at the final stage we know that each full simultaneity is covered by one and only one singular category.

The algorithm recursively carries out the fission of categories and yields all full simultaneities without repetitions.

3.5.4. Optimization - identifying blank categories

A further optimization is performed. Consider a category; a full simultaneity must contain no transfer from that category's excluder in order to be covered by that category. In addition, since the full simultaneity is full, it is in congestion with all transfers that it does not contain. Obviously any full simultaneity covered by some category must congest with each member of that category's excluder. Therefore, transfers congesting with the transfers of the excluder must be available in the depot of the category (the category's excluder, according to the fission algorithm, keeps no transfer congesting with the includer). If the excluder contains at least one transfer for which the depot has no congesting transfer, then we say that this category is *blank*. The includer of a blank category cannot be further extended by the transfers of the depot to a simultaneity which is full (and therefore congests with every remaining transfer of the excluder). The coverage of a blank category is therefore empty and there is no need to pursue its fission.

3.5.5. Retrieving full teams - identifying idle categories

Let us now instead of retrieving all full simultaneities retrieve all full teams, i.e. those full simultaneities which ensure the utilization of all bottleneck links.

A category within X is *idle* if its includer and its depot together don't use all bottlenecks of X . This means that we can not grow the current simultaneity (i.e. the includer of the category) into a full simultaneity, which will use all bottlenecks. The coverage of an idle category therefore does not contain a full simultaneity, which is a team. Idle categories allow us to prune the search tree during the early stages of the algorithm and to pursue only branches leading to full teams.

Carrying out successive fissions, starting from the prim-category and continuously identifying and removing all the blank and idle categories, ultimately leads to all full teams.

Section 3.6. Speeding up the search for full teams

This section presents an additional method for speeding up the search for all full teams $\mathfrak{S}(X)$ of an arbitrary traffic X .

3.6.1. Skeleton of a traffic

Let us consider from the original traffic X only those transfers that use bottlenecks of X and call this set of transfers the *skeleton* of X . We denote the skeleton of X as $\varsigma(X)$. Obviously, $\varsigma(X) \subset X$.

According to equations (1) and (2), equation (8) specifies the skeleton of X so as to comprise only the transfers using links whose load is equal to the duration of the traffic:

$$\varsigma(X) = \left\{ x \in X \mid \max_{l \in x} \lambda(l, X) = \Lambda(X) \right\} \quad (8)$$

Figure 33 shows the relative sizes of skeletons compared with the sizes of their corresponding traffics. We consider 362 different traffic patterns across the K-ring network of the Swiss-T1 cluster supercomputer comprising 32 nodes (see Figure 39 and Table 2 in Subsection 3.8.1). In average, the skeleton size is 31.5% of its traffic size.

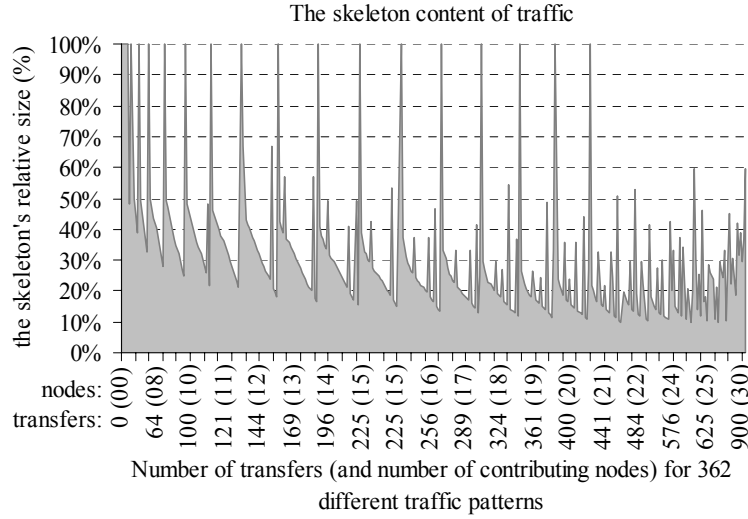


Figure 33. Fraction of transfers within a skeleton of a traffic, compared with the total number of transfers in the traffic

3.6.2. Optimization - building full teams based on full teams of the skeleton

When considering the skeleton of a traffic X as another traffic, the bottlenecks of the skeleton of a traffic are the same as the bottlenecks of the traffic. Consequently, a team of a skeleton is also a team of the original traffic.

We may first obtain all full teams of the traffic's skeleton by iteratively applying the fission algorithm on the traffic's skeleton and by eliminating the idle categories. Then, a full team of the original traffic is obtained by adding a combination of non-congesting transfers to a team of the traffic's skeleton.

We therefore obtain the set of a traffic's full teams $\mathfrak{I}(X)$ by carrying out the steps outlined in Table 1.

Table 1. Optimized algorithm for retrieving all full teams of a traffic

1.	Obtain the set of the skeleton's full teams $\mathfrak{I}(\zeta(X))$ by applying the fission algorithm on the traffic's skeleton.
2.	Create for each full team of the skeleton, a category by initializing:
2.1.	The includer with the transfers of the skeleton's full team;
2.3.	The excluder as empty;
2.2.	The depot with all transfers of X which are not congesting with the includer.
3.	Apply the fission to each category, discarding the check for idle categories, since the includer is already a team, i.e. it uses all bottlenecks.

By first applying the fission to the skeleton and then expanding the skeleton's full teams to the traffic's full teams, we considerably reduce the processing time.

3.6.3. Evaluating the reduction of the search space

Let us evaluate the reduction of the search space achieved due to the search space reduction methods proposed in Section 3.5 and in this section. We consider 23 different all-to-all traffic patterns across the network of the Swiss-T1 cluster supercomputer (see Section 3.8). The size of the algorithm's search space is the number of categories that are being iteratively traversed by the algorithm until all full teams are discovered.

Figure 34 shows the search space reduction for the four presented algorithms. The first one is the naive algorithm that builds full teams only according to the coverage partitioning

strategy (Subsection 3.5.3) without considering the other optimizations. We assume that the size of the search space of the naive algorithm is 100% and we use it as a reference for the other three algorithms. The naive algorithm is sufficiently “smart” to avoid repetitions while exploring all full simultaneities. The second algorithm, which adds identification of blank categories (see Subsection 3.5.4) permits, according to Figure 34, the reduction of the search space to an average of 28% of the search space of the naive algorithm. The third algorithm identifies idle categories and enables it to skip at an early stage the evaluation of all categories not leading to teams (see Subsection 3.5.5). This third algorithm encloses all optimizations presented in Section 3.5 and reduces the search space to an average of 20% of the search space of the initial algorithm.

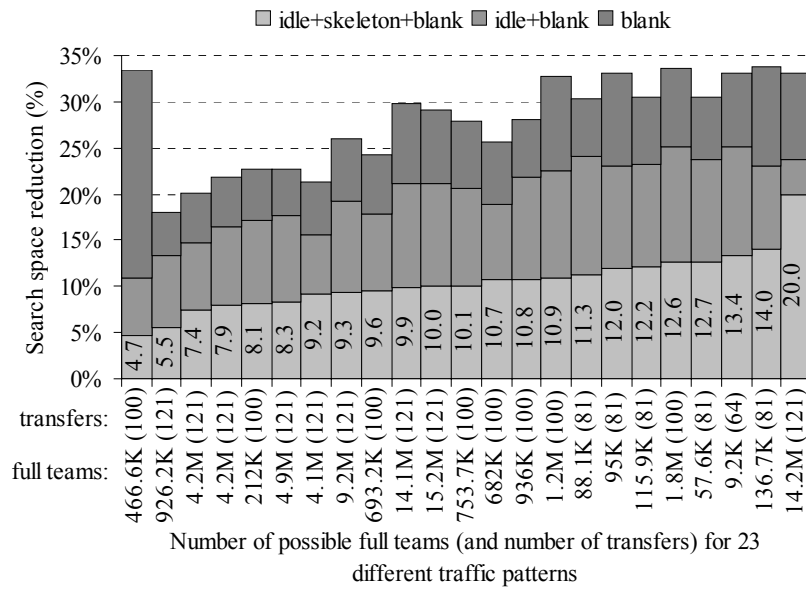


Figure 34. Search space reduction obtained by idle+skeleton+blank optimization steps

Finally, the skeleton algorithm presented in this section, which is carried out in two phases according to Table 1, reduces the search space to an average of 10.6% of the search space of the initial algorithm. Full teams are therefore retrieved, on average, 9.43 times faster than with the naive algorithm of Subsection 3.5.3, thanks to the three optimisation techniques presented in Subsections 3.5.4, 3.5.5 and 3.6.2.

Section 3.7. Construction of liquid schedules

In Section 3.5 and Section 3.6 we introduced efficient algorithms for traversing full teams of a traffic. Relying on the full team generation algorithms, this section presents methods for constructing liquid schedules for arbitrary traffic patterns on arbitrary network topologies.

3.7.1. Definition of a liquid schedule

Let us introduce the definition of a schedule. By recalling that a *partition* of X is a disjoint collection of non-empty subsets of X whose union is X [Halmos74], a *schedule* α of a traffic X is a collection of simultaneities of X partitioning the traffic X . An element of a schedule α is called a *time frame*. The *length* $\#(\alpha)$ of a schedule α is the number of time frames in α . A schedule of a traffic is *optimal* if the traffic does not have any shorter schedule. If the length of a schedule is equal to the duration of the traffic (the duration of a traffic X is the load of its bottlenecks), then the schedule is *liquid*. Thus a schedule α of a traffic X is liquid if equation (9) holds. See also equation (2) defining the duration of a traffic X .

$$\#(\alpha) = \Lambda(X) \quad (9)$$

Figure 35 shows a liquid schedule for the collective traffic shown in Figure 30, which in turn represents an all-to-all data exchange (see Figure 29) across the network shown in Figure 28.

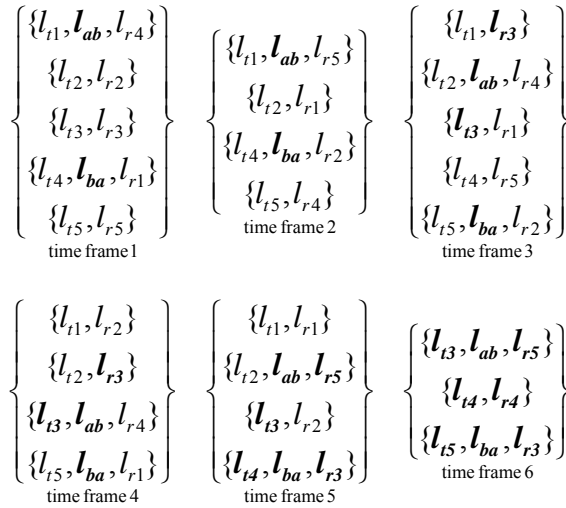

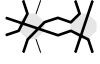


Figure 35. Time frames of a liquid schedule of the collective traffic shown in Figure 30

One can easily check that the timeframes of Figure 35 correspond to the following sequence $\{ \text{pictogram 1}, \text{pictogram 2}, \text{pictogram 3}, \text{pictogram 4}, \text{pictogram 5}, \text{pictogram 6} \}$, represented in the form of the pictograms introduced in Section 3.3. Recall that each pictogram in the sequence represents

several transmissions that can be carried out simultaneously. For example the sequence's second pictogram , visualizes four simultaneous transfers: t_1 to r_5 , t_2 to r_1 , t_4 to r_2 and t_5 to r_4 , wherein $t_1 \dots t_5$ are the source nodes and $r_1 \dots r_5$ are the destination nodes of the network of Figure 28. These four simultaneous transfers  correspond to the second time frame of Figure 35:

$$\text{Pictogram} = \{ \{l_{t1}, l_{ab}, l_{r5}\}, \{l_{t2}, l_{r1}\}, \{l_{t4}, l_{ba}, l_{r2}\}, \{l_{t5}, l_{r4}\} \} \quad (10)$$

If a schedule is liquid, then each of its time frames must use all bottlenecks. Inversely, if all time frames of a schedule use all bottlenecks, the schedule is liquid.

The necessary and sufficient condition for the liquidity of a schedule is that all bottlenecks be used by each time frame of the schedule. Since a simultaneity of X is defined as a *team* of X , if it uses all bottlenecks of X , a necessary and sufficient condition for the liquidity of a schedule α on X is that each time frame of α be a team of X .

A liquid schedule is optimal, but the inverse is not always true, meaning that a traffic may not have a liquid schedule. An example of traffic having no liquid schedule is shown in Figure 37. This traffic is to be carried across the network shown in Figure 36. There are three bottleneck links in the network $\{l_{ab}, l_{bc}, l_{ca}\}$. Since there is no combination of non-congesting transfers that can simultaneously use all three bottleneck links $\{l_{ab}, l_{bc}, l_{ca}\}$, this traffic contains no team and therefore has no liquid schedule.

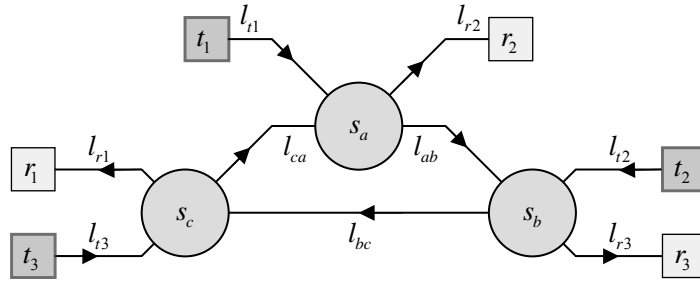


Figure 36. A traffic of three transmissions (shown in Figure 37) across this network has no team and therefore no liquid schedule

$$X = \left\{ \begin{array}{l} \{l_{t1}, l_{ab}, l_{bc}, l_{r1}\}, \\ \{l_{t2}, l_{bc}, l_{ca}, l_{r2}\}, \\ \{l_{t3}, l_{ca}, l_{ab}, l_{r3}\} \end{array} \right\}$$

Figure 37. A traffic consisting of three transmissions to be carried across the network shown in Figure 36

The rest of this section presents the liquid scheduling construction algorithm (Subsection 3.7.2) and two optimizations (Subsections 3.7.3 and 3.7.4).

In Appendix C, we show how to formulate the problem of searching for a liquid schedule with Mixed Integer Linear Programming (MILP), [CPLEX02], [Fourer03]. Appendix C presents a comparison of the performance of the liquid schedule search approach presented here with that of MILP. It shows that the computation time of the MILP method is prohibitive compared with the speed of our algorithm.

3.7.2. Liquid schedule basic construction algorithm

In this subsection we describe the basic algorithm for constructing a liquid schedule. The basic algorithm simply consists of recursive attempts to assemble a liquid schedule out of the teams of the original traffic, until a valid liquid schedule incorporating all transfers is successfully constructed. In the following subsections (Subsections 3.7.3 and 3.7.4), relying on the basic algorithm, we show how to apply further optimizations.

Our strategy for finding a liquid schedule relies on partitioning the traffic into a set of teams forming the sequence of time frames. Associate to the traffic X all its possible teams A_1, A_2, \dots, A_n (found by the algorithm presented in Section 3.6) which could be selected as the schedule's first time frame.

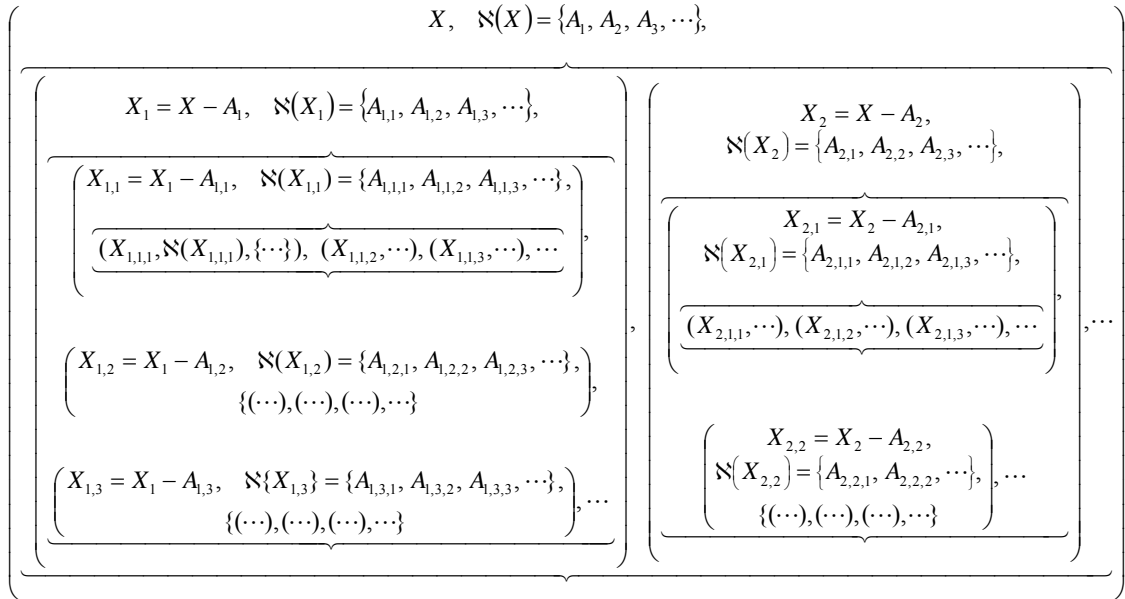


Figure 38. Liquid schedule construction tree: $X_{i_1 i_2 \dots i_n}$ denotes a reduced subtraffic at the layer $n+1$ of the tree and $A_{i_1 i_2 \dots i_n i_{n+1}}$ denotes a candidate for the time frame $n+1$; the operator \aleph applied to a subtraffic X_{sub} yields the set of all possible candidates for a time frame

The variety of possible subtraffics remaining after the choice of the first time frame is represented by the following: $X - A_1, X - A_2, \dots$. Each of the possible subtraffics X_i

remaining after the selection of the first time frame has its own set of possibilities for the second time frame $\aleph(X_i) = \{A_{i,1}, A_{i,2}, A_{i,3}, \dots\}$, where $\aleph(X_{sub})$ is a choice function. The choice of the second team for the second time frame yields a further reduced subtraffic (see Figure 38).

Dead ends are possible if there is no choice for the next time frame, i.e. no team of the original traffic may be formed from the transfers of the reduced traffic. A dead end situation may occur, for example, when the remaining subtraffic appears to be like the one shown in Figure 36 and Figure 37. Once a dead end occurs, backtracking takes place.

The construction recursively advances and backtracks until a valid liquid schedule is formed. A valid liquid schedule is obtained when the transfers remaining in the reduced traffic form one single team for the last time frame of the liquid schedule.

We rely on the construction tree of Figure 38 and assume that at any stage the choice $\aleph(X_{sub})$ for the next time frame is among the set of the original traffic's teams $\mathfrak{T}'(X)$. Thus the choice function is represented by the following equation:

$$\aleph(X_{sub}) = \{A \in \mathfrak{T}'(X) \mid A \subset X_{sub}\} \quad (11)$$

In the next subsections we improve equation (11) by considering newly emerging bottlenecks at each successive time frame.

3.7.3. Search space reduction by considering newly emerging bottlenecks

We observe in Figure 35 that when we step from one time frame to the next, additional new bottleneck links emerge. For example, from time frame 3 on, links l_{i3} and l_{r3} appear as new bottlenecks.

In the construction strategy presented in the previous subsection (3.7.2), according to equation (11) we consider as a possible time frame any team of the original traffic X that can be built from the transfers of the reduced subtraffic. A schedule is liquid if and only if (IFF) each time frame is not only a team of the original traffic but is also a team of the reduced subtraffic (see Appendix D for a formal proof). If α is a liquid schedule on X and A is a *time frame* of α , then $\alpha - \{A\}$ is a liquid schedule on $X - A$.

Thus a liquid schedule may not contain a time frame which is a team of the original traffic but is not a team of a subtraffic obtained by removing some of the previous time frames. Therefore, at each iteration, we can limit our choice by the collection of only those teams of the original traffic which are also teams of the current reduced subtraffic. Since the reduced subtraffic contains additional bottleneck links, there are fewer teams in the reduced subtraffic than teams remaining from the original traffic.

Therefore, in the liquid schedule construction diagram presented in Figure 38, regarding the choice function $\aleph(X_{sub})$ we can replace equation (11) by equation (12):

$$\aleph(X_{sub}) = \mathfrak{T}'(X_{sub}) \quad (12)$$

By considering in each time frame all occurring bottlenecks, with the new equation (12) we speed up the construction considerably.

3.7.4. Liquid schedule construction optimization by considering only full teams

In Appendix E we have shown that if a liquid schedule exists and if it can be constructed by the choice of teams, then a liquid schedule can be also constructed by limiting the choice to only full teams (see also [Gabrielyan03] and [Gabrielyan04A]).

Therefore in the construction algorithm represented by the diagram of Figure 38, the function $\aleph(X_{sub})$ for the choice of the teams may be further narrowed from the set of all teams, (as in equation (12)) to the set of full teams only:

$$\aleph(X_{sub}) = \mathfrak{T}(X_{sub}) \quad (13)$$

When upgrading the choice function $\aleph(X_{sub})$ equation from (11) to (12) and then from (12) to (13), we make sure that the new equations have no impact on the solvability of the problem. The liquid schedule construction is sped up, thanks to the reduction in choice summarized by expressions (14) and (15) below:

$$\{A \in \mathfrak{T}'(X) \mid A \subset X_{sub}\} \subset \mathfrak{T}'(X_{sub}) \subset \mathfrak{T}(X_{sub}) \quad (14)$$

and therefore also:

$$\#(\{A \in \mathfrak{T}'(X) \mid A \subset X_{sub}\}) \leq \#(\mathfrak{T}'(X_{sub})) \leq \#(\mathfrak{T}(X_{sub})) \quad (15)$$

Section 3.8. Experimental verification

In this section we present the results of application of liquid schedules to data communications carried out across a real network. In Subsection 3.8.1 we present the network on which the experiments were carried out. We select several hundred traffic patterns across the considered network. Measurements of aggregate communication throughputs, presented in Subsection 3.8.2, enable us to validate the efficiency of applying liquid schedules in real networks.

3.8.1. Swiss-Tx cluster supercomputer and 362 test traffic patterns

The experiments are carried out across the interconnection network of the Swiss-T1 cluster supercomputer (see Figure 39). The network of Swiss-T1 forms a K-ring [Kuonen99B] and is built on TNET switches. The routing between pairs of switches is static. The throughputs of all links are identical and equal to 86MB/s. The cluster consists of 32 nodes, each one comprising 2 processors [Kuonen99A], [Gruber01], [Gruber02], [Gruber05]. The cluster thus comprises a total of 64 computing processors. Each processor has its own individual connection to the network. The network enables transmissions of large messages at low latencies. Wormhole switching is employed for this purpose.

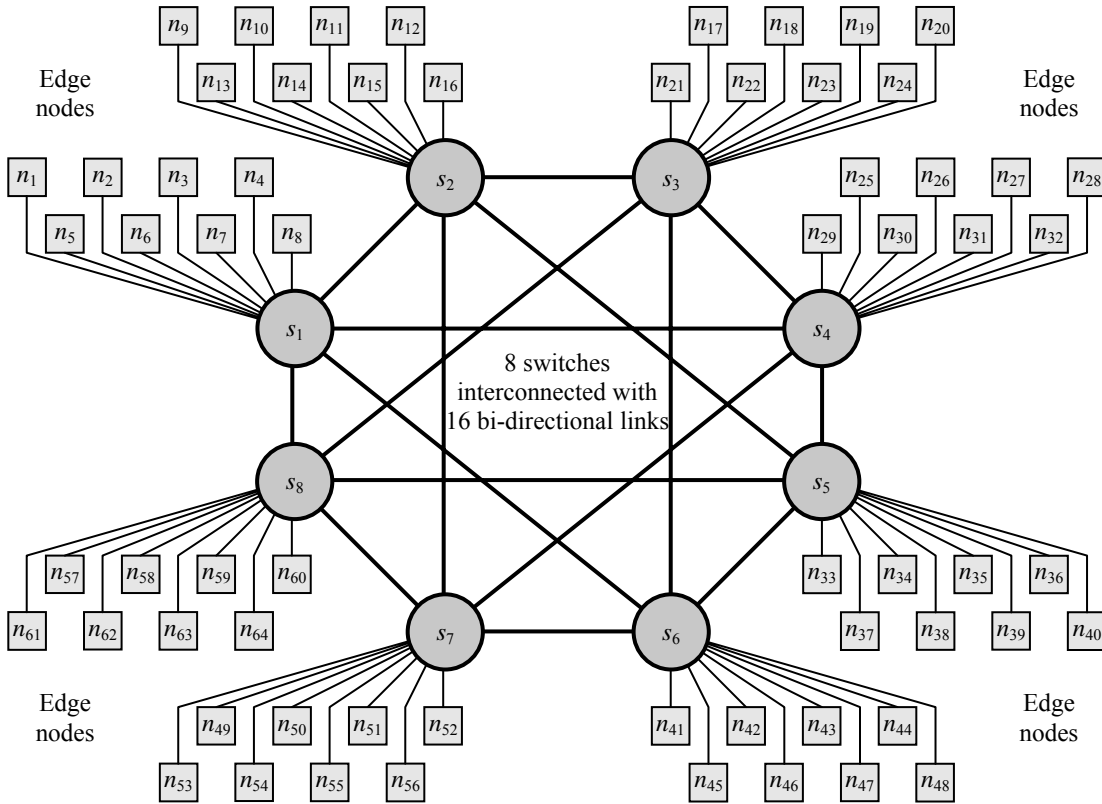


Figure 39. Architecture of the Swiss-T1 cluster supercomputer interconnected by a high performance wormhole switch fabric

Communication between a pair of any two switches requires at most one intermediate switch. The routing is summarized in Table 2. Transmissions from switch i to switch j are routed through the switch whose number is located at position (i, j) of the table. Symbol “ \leftrightarrow ” indicates that the two switches are connected by a direct link.

Table 2. The routing table of the Swiss-Tx supercomputer shown in Figure 39

Routing table								
	1	2	3	4	5	6	7	8
1		↔	2	↔	4	↔	8	↔
2	↔		↔	7	↔	3	↔	5
3	2	↔		↔	4	↔	8	↔
4	↔	7	↔		↔	7	↔	3
5	4	↔	4	↔		↔	6	↔
6	↔	3	↔	7	↔		↔	1
7	8	↔	8	↔	6	↔		↔
8	↔	5	↔	3	↔	1	↔	

We perform our experiments on a number of different data intensive traffic patterns across the network of the Swiss-T1 cluster. We limit ourselves to only those traffic patterns where within each node one of the processors only transmits and the other one only receives. For any given allocation of nodes we have an equal number of sending and receiving processors and we assume a traffic pattern where each sending processor transmits a distinct message (of the same size) to each receiving processor. Thus, according to our assumptions, if there are n allocated nodes (i.e. pairs of processors), then there are n^2 transmissions to be carried out.

The Swiss-T1 cluster supercomputer comprises 32 nodes, 8 switches and 4 nodes per switch. We have therefore 5 possibilities of allocating nodes to each switch (from 0 to 4 nodes). This yields $5^8 = 390625$ different node allocation patterns. To limit our choice to substantially different patterns of underlying topologies, we have computed the liquid throughputs for each of the 390625 topologies (taking into account the static routing). Because of various symmetries within the network, many of these topologies yield identical liquid throughputs and only 362 unique liquid throughput values were obtained. We selected one representative topology for each throughput value. Therefore 362 topologies yielding different liquid throughput values are obtained.

Figure 40 shows these 362 traffic patterns (topologies), each one characterized by the number of contributing nodes and by its liquid throughput. Depending on how a given number of nodes are allocated in the cluster, the corresponding underlying network changes its topology considerably. Therefore for any given number of nodes, Figure 40 shows that the liquid throughput varies considerably.

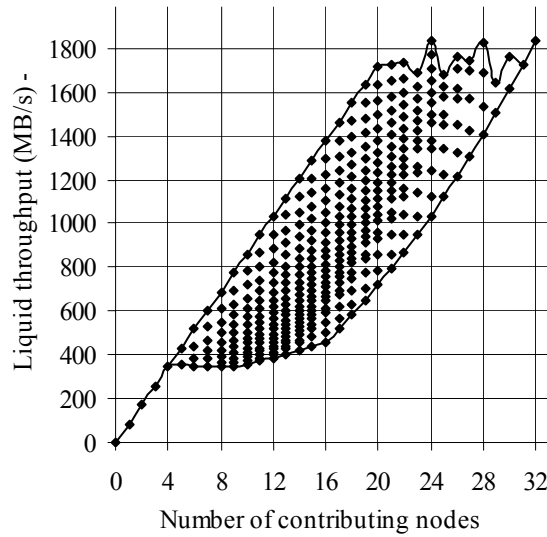


Figure 40. For a given number of contributing nodes, all possible allocations of nodes yielding different liquid throughputs

The management system for Computing in Distributed Networked Environment (CODINE) and the Load Sharing Facility (LSF) are the job allocation and the scheduling consoles used in Swiss-T1 [Byun00], [Hassaine02]. Taking into account the data of Figure 40 the CODINE and LSF job allocation systems of Swiss-T1 are experimentally tuned for communication intensive programs (of high priority). In these experiments the allocation strategy is simple and the fairness among several communication intensive jobs is not considered.

These 362 topologies may be also placed along one axis, sorted first by the number of nodes and then according to their liquid throughput, as shown in Figure 41.

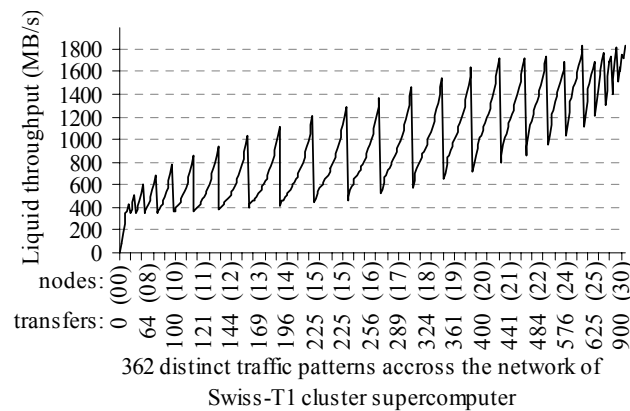


Figure 41. The 362 topologies of Figure 40 yielding different liquid throughput values placed along one axis, sorted first by the number of contributing nodes and then by their liquid throughputs

3.8.2. Real traffic throughput measurements

The 362 traffic patterns of Figure 40 and Figure 41 were scheduled both by schedules found according to our liquid scheduling algorithms, and by a topology-unaware round-robin schedule (or randomly). Overall throughput results for each method are measured and presented for comparison. In each chart, the theoretical liquid throughput values of Figure 41 are given for comparison with the measured values.

Figure 42 shows the overall communication throughput of the 362 traffic patterns carried out by a topology-unaware round-robin schedule. The size of messages, i.e. the amount of data transferred from each transmitting processor to each receiving processor, is equal to $2MB$. 20 measurements were made for each traffic pattern, and the chart shows the median of their throughputs (the black dots). According to the chart, the round-robin schedule yields a throughput which is far below the liquid throughput of the network. Tests with various other topology-unaware methods (such as transmission in random order or in FIFO order) yield throughputs which are not better than that of the round-robin schedule.

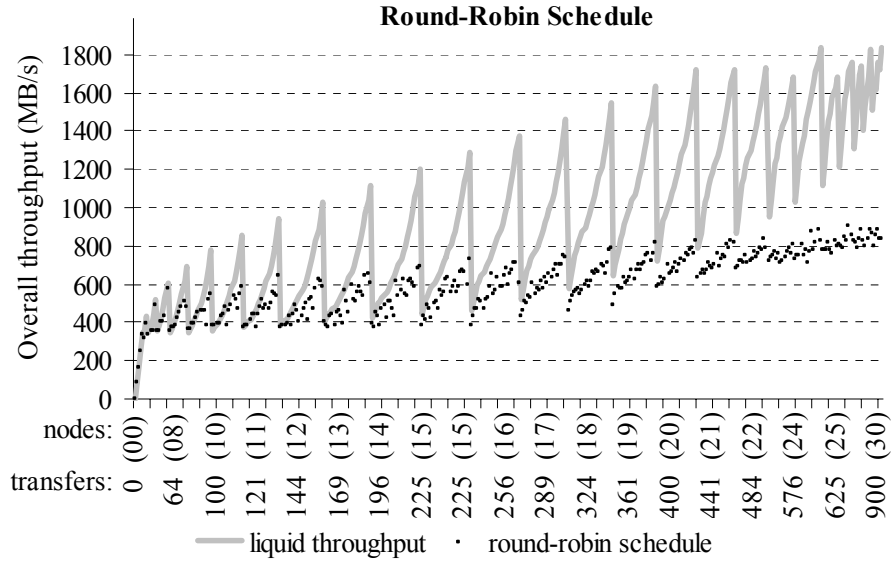


Figure 42. Theoretical liquid throughput and measured round-robin schedule throughput for 362 network sub topologies.

Next, we carried out the same 362 traffic patterns but scheduled according to the liquid schedules found by our algorithms. The overall throughput results are shown in Figure 43. The size of the messages (processor to processor transfers) is of $5MB$ (larger than for the measurements of Figure 42). Each black dot represents the median of 7 measurements. The chart shows that the measured aggregate throughputs (black dots) are very close to the theoretically expected values of the liquid throughput (gray curve).

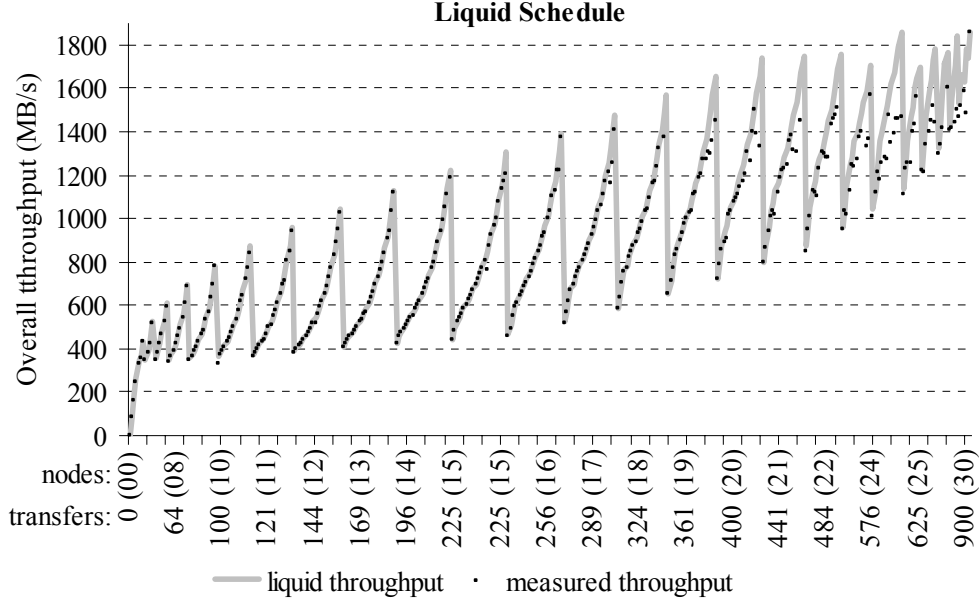


Figure 43. Predicted liquid throughput and measured throughput according to the computed liquid schedule

Comparison of the chart of Figure 42 with that of Figure 43 demonstrates that for many traffic patterns, liquid scheduling allows an increase in the aggregate throughput by a factor of two compared with topology-unaware round-robin scheduling. The gain is especially significant for large topologies and heavy traffics.

Thanks to the full team space reduction algorithms (Section 3.5 and Section 3.6) and liquid schedule construction optimizations (Section 3.7), the computation time of a liquid schedule for more than 97% of the considered topologies takes no more than 1/10 of a second on a single PC.

Section 3.9. Conclusions

In circuit-switching coarse-grained networks (e.g. optical lightpath routing and wormhole switching), significant throughput losses occur due to attempts to simultaneously carry out transfers sharing common communication resources. The communications must be scheduled such that congesting transmissions are not carried out simultaneously. We propose a *liquid scheduling algorithm*, which properly schedules the transmissions within a time as short as the utilization time of a bottleneck link. A liquid schedule therefore yields an aggregate throughput equal to the network's theoretical upper limit, i.e. its *liquid throughput*. To construct a liquid

schedule, we must choose time frames utilizing all bottleneck links and perform as many transfers as possible within each timeframe.

These saturated subsets of non-congesting transfers using all bottleneck links are called *full teams* and are needed for the construction of a liquid schedule. Efficient construction of liquid schedules relies on the fast retrieval of *full teams*. We obtained a significant speed up in the construction algorithm by carrying out optimizations in the retrieval of full teams and in their assembly into a schedule. The liquid schedule construction algorithm and its optimizations are briefly outlined in Appendix F.

Measurements on the traffic carried out on various sub-topologies of the Swiss-T1 cluster supercomputer have shown that for most sub-topologies, we are able to increase the overall communication throughput by a factor between 1.5 and 2 (see Figure 66 of Appendix F).

In congestion-prone coarse-grain transmission networks, liquid scheduling considerably improves the overall throughput by ensuring an optimal utilization of the transmission resources (e.g. the bottleneck communication links, optical wavelengths and time frames). By preventing contention, liquid scheduling minimizes the overall transmission time of large communication patterns containing many congesting transfers.

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

In off-line streaming, packet level erasure resilient codes rely on 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. However, path diversity increases the number of underlying links, thereby increasing the total link failure rate, which may possibly require more FEC packets from the sender. 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 protecting 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 expectations, the overall requirement in FEC codes is reduced when increasing the path diversity according to a new 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 Forward Error Correction (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 a satellite broadcast channel (without feedback) to millions of motor vehicles under conditions of fragmentary visibility [[Honda04](#)]. In the film industry, instead of relying on the 48-hour delivery time of FedEx, the day's film footage can be

delivered from the location where it has been shot to the studio that is many thousands of miles away over the lossy Internet even with long propagation delays (see [Hollywood03] and LT codes [Luby02]). The third Generation Partnership Project (3GPP) recently adopted Raptor [Shokrollahi04] as a mandatory code in Multimedia Broadcast/Multicast Service (MBMS). Off-line streaming benefits from application of FEC thanks to time diversity, i.e. the receiver's right to not forward the received information to the user immediately. When long buffering time is not a concern, the receiver can hold the received packets without restriction, and as a result, packets representing the same information can be collected at distant periods of time.

In real-time single-path streaming, FEC can only mitigate short failures of fine granularity [Choi06], [Johansson02], [Huang05], [Padhye00], [Altman01]. Due to the restricted playback buffering time, packets representing the same information cannot be collected at very distant periods of time. For application of FEC in real-time streaming, instead of relying on time-diversity, one can rely on path-diversity. Recent publications show the applicability of FEC in real-time streaming when using dual-path routes. It has been shown that strong FEC sensibly improves video communication established along two disjoint paths and that in two correlated paths, weak FEC codes are still advantageous [Qu04]. Tawan proposes adaptive multi-path routing for Mobile Ad-Hoc Networks (MANET) mainly for load balancing and capacity issues, but mentions also the potential advantages in respect to FEC [Tawan04]. Ma suggests simple multi-path patterns in MANET and injection of FEC codes not only at the end nodes but also at each intermediate node [Ma03A], [Ma04]. Nguyen and Byers study video streaming from multiple servers [Nguyen02], [Byers99]. Nguyen later studies real-time streaming over a dual-path route [Nguyen03]. He used a static amount of redundancy, streaming the media with FEC blocks carrying 23 source packets and 7 redundant packets (using Reed-Solomon RS(30,23)). Then, similarly to [Qu04], he compares dual-path scenarios with the single Open Shortest Path First (OSPF) routing strategy and shows 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 an FEC efficient routing pattern.

In this chapter we try to present a comparative study for various multi-path routing patterns. Since it is too hostile, single path routing is excluded from our comparisons. We build steadily diversifying routing patterns layer by layer thanks to the *capillary routing* algorithm (Section 4.2).

In order to rate the effectiveness of multi-path routing patterns, we introduce the *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 streams the media with static FEC codes, allowing the application 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 reverse 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], [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 which is 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 thanks to the capillary routing algorithm. Network samples are obtained from a random walk MANET with several hundred 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 show how to detect the bottlenecks at each layer of the capillary routing algorithm so as to construct the 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 44, Figure 45 and Figure 46 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.

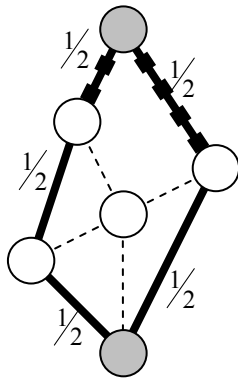


Figure 44. In the first layer the flow is equally split across two paths. Two of their links, marked by thick dashes, are the bottlenecks.

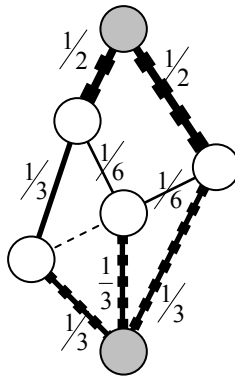


Figure 45. The second layer minimizes to $1/3$ the maximal load of the remaining seven links and identifies three bottlenecks.

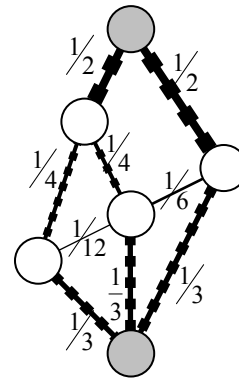


Figure 46. The third layer minimizes to $1/4$ the maximal load of the remaining four links and identifies two bottlenecks.

Figure 47 shows the 10th 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 10th layer. There could be tens of additional routing layers before the complete capillarization is achieved.

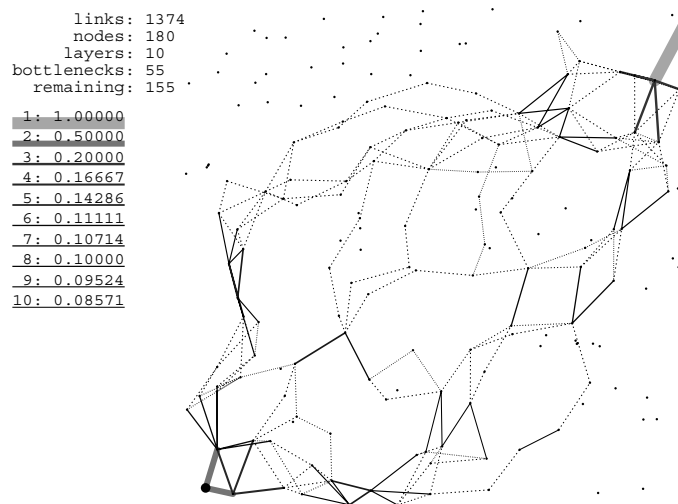


Figure 47. Routing pattern of layer 10 built by the capillary routing algorithm on a network sample with 180 nodes

By increasing the number of the underlying links, the overall rate of network failures increases. High overall failure rate increases the probability of overlapping failures, except when the link's ratio of failure time over operational time is sufficiently small. Since for computing

the ROR metric in Section 4.1 we assume a single link failure (more in Subsection 4.3.1), we consider that the single link's ratio of failure time over operational time is sufficiently small to ensure that our assumption holds. In Appendix G we present the limits for which our single link failure assumption holds. We also show how our theory can be further expanded to consider also simultaneous link failures.

4.2.2. Numerically stable version

Although the described LP process is perfectly valid, it is numerically unstable. 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 48 and Figure 49).

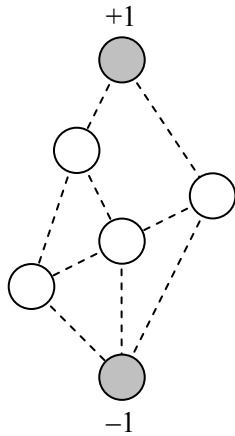


Figure 48. Initial problem with one source and one sink node

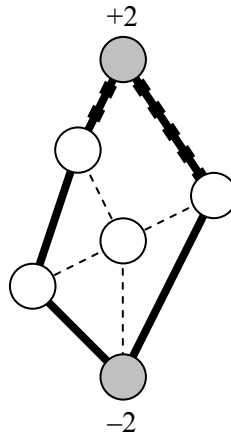


Figure 49. Maximize the flow, fix the new flow-out coefficients at the nodes and find the bottleneck links (layer 1, $F^1 = 2$)

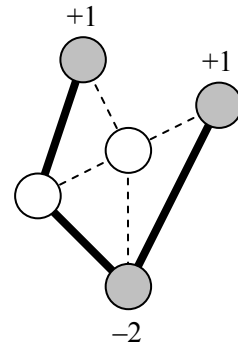


Figure 50. Remove the bottleneck links from the network and adjust the flow-out coefficients at the adjacent nodes

At each subsequent layer (Figure 50 to Figure 53) 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, in general, belong to the simple class of “network linear programs” (see [Fourer03], page 343).

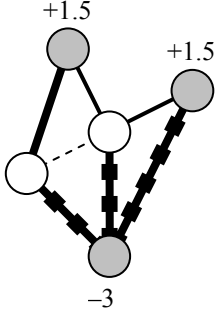


Figure 51. Maximize the flow in the new sub-problem, fix the new flow-out coefficients at the nodes and find the new bottlenecks (layer 2, $F^2 = 1.5$)

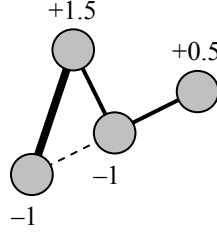


Figure 52. Again remove the bottleneck links from the network and adjust correspondingly the flow-out coefficients at the adjacent nodes

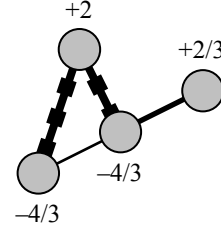


Figure 53. Maximize the flow in the obtained new problem, fixing the new resulting flow-out coefficients at the nodes and find the new bottlenecks (layer 3, $F^3 = 4/3$)

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:

$$N^{l+1} = N^l \quad (16)$$

the bottlenecks are removed from the network:

$$L^{l+1} = L^l - B^l \quad (17)$$

and the flow out coefficients are correspondingly adjusted:

$$f_j^{l+1} = f_j^l \cdot F^l + \sum_{(i,j) \in B^l} (+1) + \sum_{(j,k) \in B^l} (-1) \quad (18)$$

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 or 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 $1 \leq i \leq l$) of the present layer and all preceding layers:

$$r_{i,j} = \frac{1}{\prod_{i=1}^l F^i} \quad \text{where } l \text{ is the layer for which } (i, j) \in B^l \quad (19)$$

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 precision errors in the flow-out coefficients of the LP problem according to the flow conservation rules. The LP problems generated for each successive layer of capillary routing are freed from precision errors and therefore the errors cannot propagate and lead to numerical instabilities.

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 54 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 the four maximally loaded candidate links $\{a, b, d, e\}$, marked in Figure 55 by thick dashes.

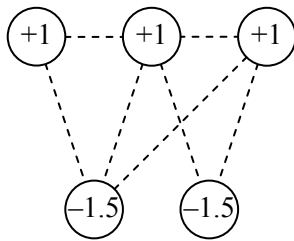


Figure 54. An example of a bounded multi-source/multi-sink problem (obtained during construction of the capillary routing from a network with one source and one destination node)

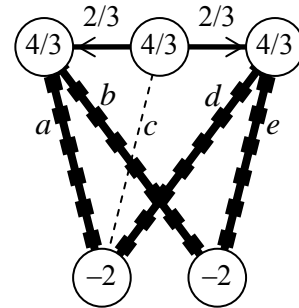


Figure 55. A max-flow solution with the flow increase factor of $4/3$, containing four maximally loaded candidate links $\{a, b, d, e\}$

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 55 the sum of loads of all four bottleneck candidate links can be minimized (by an LP objective) to 3 (see Figure 56). Now only three links $\{a, b, e\}$, marked by thick dashes, continue to maintain the maximal load. The sum of the loads of the three remaining bottleneck candidate links can be further reduced to 2 (see Figure 57). These two remaining links $\{b, e\}$, marked by thick dashes, maintain the maximal load at all times and are the true bottleneck links since the sum of their loads cannot be further reduced.

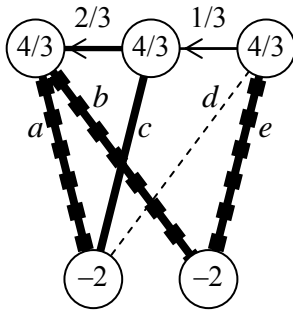


Figure 56. The cost reduction applied to the four fully loaded links of Figure 55 reduces the load of suspected link d , and the bottleneck candidate list is now $\{a, b, e\}$.

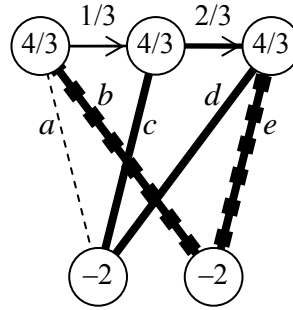


Figure 57. The cost reduction applied to the three fully loaded links of Figure 56 reduces the load of another suspected link a . The true bottleneck links are $\{b, e\}$.

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

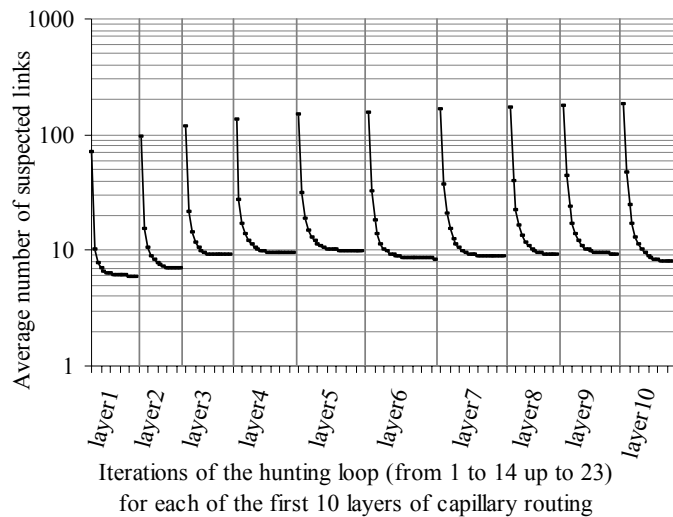


Figure 58. Decrease in the number of suspected links during the bottleneck hunting loop at each of the 10 capillary routing layers

For capillary routing layers built simultaneously on 200 independent network samples each with 300 nodes (on average 2,555.7 links per network), Figure 58 shows the decrease in the number of bottleneck candidate links during the bottleneck hunting loop of each capillary routing layer from 1 to 10.

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 the ROR metric are given in Subsection 4.3.1. The computation of the transmitted FEC block size as a function of the packet loss rate p is presented in Subsection 4.3.2. The equation of the ROR metric for the particular case of very large FEC blocks is presented in Subsection 4.3.3.

4.3.1. Definition of ROR

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

For a given routing pattern, the ROR metric 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:

$$ROR = \sum_{l \in L} \left(\frac{P_l}{P} - 1 \right) \quad (20)$$

Let us consider a long communication, and let D be the total failure time of a single network link during the entire 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 (20):

$$D \cdot P \cdot ROR = D \cdot P \cdot \sum_{l \in L} \left(\frac{P_l}{P} - 1 \right) \quad (21)$$

$$= \sum_{l \in L} (D \cdot P_l - D \cdot P) \quad (22)$$

Assuming one single link failure at a time (see Appendix G) and a uniform probability and duration of link failures, according to equation (22), $D \cdot P \cdot ROR$ 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's metric 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 an 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 t to weak losses, with $0 \leq t < 1$. 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 $0 \leq r(l) \leq 1$ according to the routing pattern, produces at the receiver a packet loss rate equal to the same relative traffic load $r(l)$.

Equation (20) for ROR can thus be re-written as follows:

$$ROR = \sum_{l \in L \mid t \leq r(l) < 1} \left(\frac{FEC_{r(l)}}{FEC_t} - 1 \right) \quad (23)$$

**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 (23), 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

Let us compute the FEC_p function (the number of packets in the FEC block as a function of the packet loss rate p) 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 transmitted FEC block.

In order to collect a mean of M packets at the receiver at a 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 $\delta \leq DER$, in order to compute the $FEC_p \geq M$ 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:

$$P_p(n|N) = \binom{N}{n} \cdot p^n \cdot q^{N-n} \quad (24)$$

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

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

$$\delta = \sum_{n=N-M+1}^N \binom{N}{n} \cdot p^n \cdot q^{N-n} \quad (25)$$

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 $M \rightarrow \infty$ (for packet loss rates much larger than a very small DER).

Regarding the upper bound, when M is equal to 1, the FEC block comprises copies of the single source packet (repeated FEC_p times). The probability that all packets will be lost is p^{FEC_p} , which is the probability that the source packets of the FEC block (in this case only one packet) cannot be recovered, i.e. p^{FEC_p} is the DER (equations (26) and (27)).

$$p^{FEC_p} = DER \text{ when } M = 1 \quad (26)$$

Therefore:

$$\frac{FEC_p}{M} = \log_p(DER) \text{ when } M = 1 \quad (27)$$

Regarding the lower bound of FEC_p / M , the larger the M (and thereby the number of the packets in the transmitted FEC block), the smaller the probability that the actual ratio of the received packets is significantly different from the expected mean ratio of received packets $1 - p$. Therefore in such an ideal case, the sender needs to transmit only $M / (1 - p)$ times more packets to ensure the delivery of M packets (equation (31)).

$$\lim_{M \rightarrow \infty} \frac{FEC_p}{M} = \frac{1}{1-p} \quad (28)$$

For M from 1 to 10 these transmission rate increase factors are plotted in Figure 59 (for $DER=10^{-5}$). Figure 59 shows that the higher the number of media packets in the block, the closer the transmission rate increase FEC_p / M approaches the lowest theoretical limit.

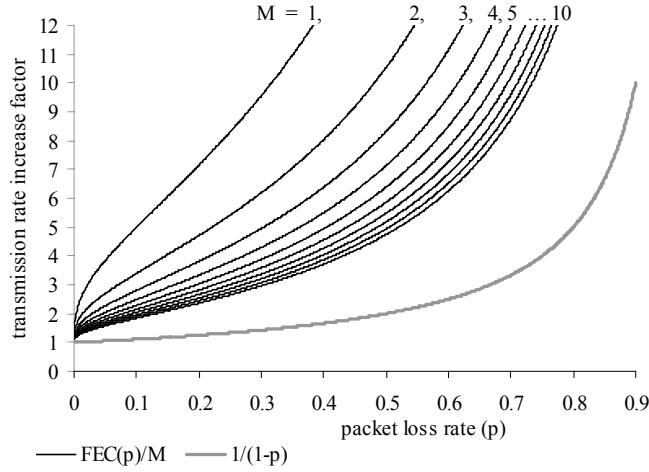


Figure 59. Transmission rate increase factor as a function of the packet loss rate ($DER = 10^{-5}$)

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, but the longer the buffering time at the receiver. For example, VOIP with a 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, then with thousands of source packets in an 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 low-density parity-check codes (LDPC) [MacKay96], [Richardson01] or fountain codes [MacKay05] can decode a large block of source packets, requiring only a very few excess packets.

In such a case, by replacing in equation (23) FEC_t with $1/(1-t)$, and $FEC_{r(l)}$ with $1/(1-r(l))$, the ROR metric of a multi-path routing pattern is computed according to the following equation:

$$ROR = \sum_{l \in L | t \leq r(l) < 1} \left(\frac{1-t}{1-r(l)} - 1 \right) \quad (29)$$

Path diversity also offers advantages for off-line large file downloads. For a typical Internet user, the connection bottleneck is usually the last mile (or last kilometer) link (for example a DSL connection). Therefore, the best estimation of the download time is related to the bottleneck's capacity. Losses or temporary failures occurring on a single communication path cause idle times. If the losses occur somewhere other than the bottleneck, the result is that the capacity of the last kilometer bottleneck is not used efficiently. Thanks to path diversity, the idle times of the last kilometer bottleneck occurring due to failures in the lossy Internet can be avoided. Relying on multi-path routing, a sender with an adaptive transmission rate can feed the last kilometer bottleneck link constantly at its maximal bandwidth so as to attain the minimal download time (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 (29). Note that according to equations (23) and (29), 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 also consider streaming media at 15 different tolerance values of static FEC codes which tolerate small packet loss rates from 3.6% to 7.8% (with an increment of 0.3%).

In Figure 60 we plot the average ROR coefficients for 300 different network instances of MANET having 115 nodes. The 300 timeframes are divided into seven nearly equal sets of consecutive timeframes. Each set contains about 43 successive network instances (i.e. network samples). For each set of samples and for each static FEC tolerance value we plot the average ROR coefficient (over all considered network samples) as the routing layer increases. Figure 60 shows that the ROR metric, i.e. the overall requirement in adaptive FEC packets decreases with capillarization. The ROR coefficients of the routing samples are computed according to equation (23) assuming a short playback buffering time. The FEC block size is computed as a function of the packet loss rate p according to equation (25). The number of media packets (M) per transmission block is 20 and the desired decoding failure rate (DER) is 10^{-5} .

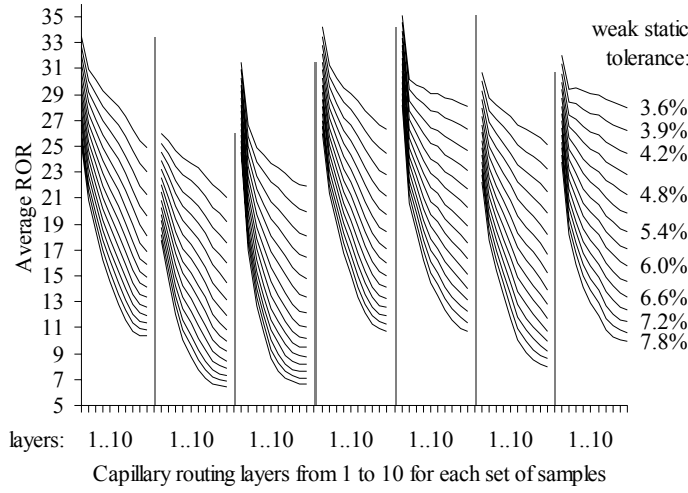


Figure 60. Average ROR metric as a function of the capillary routing layer

In Figure 61 we plot the average ROR coefficients for 150 different instances of MANET with 120 nodes. The 150 instances are divided into four sets of network samples. Each set of network samples comprises about 38 consecutive timeframes. The upper 15 curves are computed similarly to the curves of Figure 60 according to equations (23) and (25), where $M = 20$ and $DER = 10^{-5}$. However, the lower 15 curves of Figure 61 are computed according to equation (29) for streaming with large FEC blocks.

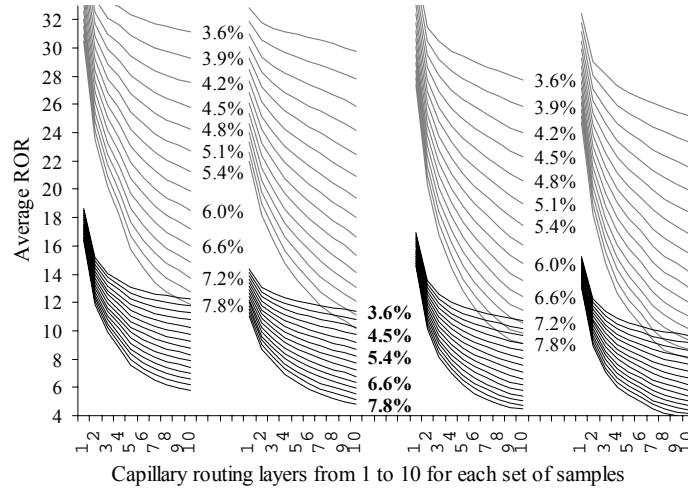


Figure 61. Average ROR metric computed assuming real-time streaming (the group of curves above) and off-line streaming (the group below)

When the application streams with large blocks, the ROR metric, representing the total redundancy effort, is twice as low as when the application streams maintaining restricted playback buffering time. The capillarization of routing is however beneficial in both cases.

Logically, the ROR curve of the media stream is shifted down as the statically added tolerance increases. The ROR represents the total amount (during the whole communication time) of redundant packets dynamically added as responses to temporary failures occurring in the network. Therefore if the static portion of the constantly maintained redundancy is increased a shifting of the ROR value must be expected.

Our simulations show however that the increase of the weak static tolerance emphasizes also the efficiency gain achieved by capillarization. A small increment of the static tolerance results in a small decrease of the ROR value if the diversity of the routing pattern is not strong. As the capillary routing layer increases and the path diversity develops, every small increment of the static tolerance results in a much more significant decrease of the ROR values.

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 60 and Figure 61 show that despite the larger communication footprint, with the routing patterns built by the capillary routing algorithm, the requirement in redundant packets decreases noticeably in most cases.

Section 4.5. Conclusions and perspectives

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. This is because in single path communications, even heavy FEC overheads cannot protect against failures lasting longer 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 the combination of channel coding and 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, thanks to the developed underlying routing patterns the streaming communication traverses the network more securely, using all parallel capacities available within the network. Unlike max-flow or shortest path solutions, for a given source and destination, by construction there exists only one solution of capillary routing.

We introduced the ROR coefficient, a metric for rating multi-path routing patterns with respect to the overall FEC effort 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 increased path diversity results in an 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 is 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].

In case of a typical Internet user connected to the network with a single link (usually also the bottleneck of the communication), path diversity cannot be achieved at that portion of the route (the last kilometer). Capillarization of the entire routing therefore can protect the streaming media only against the failures occurring in the Internet, and cannot prevent the failures occurring in the last kilometer link. However, in almost all cases, the failures and losses in the streaming communication (e.g. in VOIP) occurring in the last kilometer link are all due to congestion with bursty TCP traffic (e.g. HTTP). Unless there are physical failures, all congestion of the streaming media with the bursty traffic competing for the single last kilometer link are solved by proper QoS settings at the router. The router in this particular case is under control of the user (or at least of the immediate service supplier of the user). It is the rest of the network which has unpredictable and uncontrollable QoS policies, that needs the capillary routing. We hope that our investigation will provide some guidelines for future design of path diversity-based real-time streaming systems.

Conclusions

Parallel I/O

In Chapter 2 we presented the design and evaluation of a striped file I/O (SFIO) library which provides high performance parallel I/O within a Message Passing Interface (MPI) environment. We achieved a good load balance and equilibrated parallelism thanks to the fine granularity of a stripe unit size as low as a hundred bytes. We reduced communication and disk access overhead due to fine granularity by aggregating small data chunks into large messages. The optimizations are performed in the caches of compute nodes. We sort the remote I/O disk requests according to their offsets on the remote disks. Whenever possible we remove the overlapping segments and merge the small requests into continuous large requests. Network communication between any pair of nodes is also aggregated, even if the corresponding I/O requests cannot be further aggregated. In addition to a simple Unix-like interface SFIO, also supports a multi-block API. It allows the underlying I/O system to aggregate not only fragmentations arising from the striping of the global file but also the fragmentations present in the user memory layout. The gain from the multi-block interface is especially emphasized in scientific applications, e.g. multidimensional matrices.

The optimization subsystem is CPU efficient and requires very little memory. The I/O requests stored in the caches of compute nodes contain only pointers to the local memory and offsets in the global file. Aggregation operations in compute nodes are carried out at the level of pointers and offsets and no user data is actually copied.

The optimization subsystem converts the initial set of user requests into an optimized set of requests. Based on the optimized set of requests, we create on the fly an MPI derived datatype pointing to the fragmented layouts to be communicated to the remote I/O nodes. The communication between the fragmented memory layout and the network is carried out by a single MPI operation without memory copy.

SFIO exhibits high performance even for very small striping factors. It scales linearly as the number of I/O nodes increases. With the increase of compute nodes, the overall performance of the underlying I/O layer is not affected by concurrent accesses. For cluster computing, SFIO is a lightweight, portable parallel I/O solution for out-of-core MPI programs.

We also designed an isolated MPI-I/O layer (part of the MPI-2 specifications) which permits a user to interface with the SFIO subsystem through standard MPI-I/O operations. According to the specifications of MPI-2, in MPI-I/O all communications and disk accesses between the user processes and the I/O layer are carried out through derived datatypes. The user specifies desired fragmentations both in the memory and in the global file by creating corresponding derived datatypes. The derived datatypes are created recursively using dedicated MPI-1 operations. Once the opaque datatype is created, it cannot be decoded by a third party

library (e.g. SFIO) which uses standard MPI-1 operations. Therefore, for implementation of MPI-I/O it is assumed that one has access to the source code and to the internal structure of the particular MPI-1 implementation for which the MPI-I/O interface design is intended.

We developed a method permitting the library to decode opaque datatypes and recover their flattened layout relying only on the standard MPI-1 interface. This reverse engineering technique relies on analyzing the memory patterns after a virtual communication is carried out locally. The technique for flattening arbitrary datatype patterns permits us to provide a portable MPI-I/O interface, independent of a particular MPI-1 implementation.

Liquid schedules

High performance computing relies on networks with very low latencies. In such networks, large messages are copied from one processor to another across the network. The intermediate switches direct the content of the message without storing and forwarding the messages at each intermediate hop.

Simultaneous transmissions of large, indivisible messages across the network may result in congestion when the transmission paths intersect. When the number of parallel transmissions increases (e.g. in I/O) the rate of congestions increases rapidly. The throughput gain achieved by the data aggregation can be canceled by the high rate of congestions.

Optical networks are another example of coarse-grained circuit switching networks. Lightpaths sharing a common wavelength on a common link cannot be established during overlapping periods of time. Increasing the number of parallel transmissions may yield many blocked lightpaths and affect the throughput.

The theoretical upper limit of a network's capacity is its liquid throughput. The liquid throughput corresponds to the flow of a liquid in an equivalent network of pipes. The aggregate throughput of an arbitrarily scheduled collective communication may be several times lower than the maximal potential throughput of the network due to congestions between simultaneous transfers sharing a common communication resource.

We present a method for scheduling the transfers of a traffic so as to attain the liquid throughput of the network. This method, called liquid scheduling, relies on the knowledge of the underlying network topology and ensures an optimal utilization of all bottleneck links of the network. The liquid scheduling algorithm properly schedules the transmissions within a time as short as the utilization time of a bottleneck link. This guarantees that the liquid throughput is attained.

To construct a liquid schedule, we must choose time frames utilizing all bottleneck links, and perform as many transfers as possible within each timeframe. We therefore partition the traffic into time frames comprising mutually non-congesting transfers, keeping all bottleneck links busy during all time frames. The saturated subsets of non-congesting transfers using all

bottleneck links are called full teams. An efficient construction of liquid schedules relies on the fast retrieval of full teams. We obtained a significant speed up in the construction algorithm by carrying out optimizations both in the retrieval of full teams and in their assembly into a schedule.

Measurements on the traffic carried out on various sub-topologies of the Swiss-T1 cluster supercomputer have shown that for most sub-topologies, we are able to increase the overall communication throughput by a factor of between 1.5 and 2.

The liquid schedules can be found in a fraction of a second for traffic patterns consisting of several thousand transfers across networks of up to a hundred nodes.

Liquid scheduling can be applied to High Performance Computing (HPC) networks. It can also be applied to optical networks, for example in Optical Burst Switching (OBS) where the edge IP routers perform liquid scheduling in order to ensure an efficient utilization of the capacities of the interconnecting optical cloud.

Capillary routing

We presented a method for achieving fault-tolerance for real-time packetized communications. This method relies on using parallel paths and erasure resilient codes.

In real-time streaming, the extremely short playback buffering time makes erasure resilient codes inefficient for protecting a single path communication against long link failures. A combination of erasure resilient codes with path diversity makes Forward Error Correction (FEC) codes a very efficient method for protection of real-time communications.

Applicability of FEC when streaming only through dual path routes was already studied. We show that additional path diversity can significantly reduce the overall effort of the sender even if the number of links in the communication footprint and therefore also the overall failure rate increases.

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, thanks to the developed underlying routing patterns, the streaming communication traverses the network more securely. By using all parallel capacities available in the network, the damage caused to the media stream by single link failures is minimized. Unlike max-flow or shortest path solutions, for a given source and destination, by construction there exists only one solution of capillary routing.

We introduced a scalar coefficient for rating a multi-path routing topology of any complexity in respect to the overall FEC effort of the sending node. It is called Redundancy Overall Requirement (ROR) and is proportional to the total number of adaptive FEC packets required for protecting communications from link failures arbitrarily occurring in the network. With the capillary routing algorithm, we built thousands of multi-path routing patterns. By

computing their ROR coefficients, we showed that the overall requirement in FEC codes is reduced when increasing the path diversity according to a new capillary routing algorithm. Overall requirement in FEC codes is reduced despite the fact that the increased path diversity results in an increase of the overall failure rate of underlying links.

Capillary routing can be applied to multi-hop mobile wireless networks, to corporate IP networks or to the networks of ISPs. It can also be applicable to the public Internet, assuming an overlay network. The demand for streaming applications is growing rapidly. A typical residential Internet user is connected to the network with a last kilometer link. Although the last kilometer link offers no possibility for path diversity, it is connected to the router, which is under the control of the user (or at least the immediate ISP of the user) and the streaming media can be protected at the QoS level. Therefore no congestion-provoked failures can occur on the last kilometer link. The congestions and failures arbitrarily occurring in the lossy Internet can be solved thanks to end to end erasure resilient coding and path diversity relying on an overlay network.

Further work

With respect to liquid scheduling, we may in the future study dynamic models where the edge nodes of an optical cloud continuously receive communication flows which evolve over time. There is a need for investigating queuing strategies of the edge nodes for optimal application of liquid scheduling.

With respect to capillary routing, we may extend the equations of ROR to also consider simultaneous link failures. For simple network samples, we should compare the theoretically optimal multi-path routing patterns according to the ROR metric [060509] with the patterns obtained by our capillary routing algorithm in order to further evaluate its efficiency. Furthermore, the current study does not take into account the overall network utilization. Strategies permitting the simultaneous optimization of the overall network utilization and minimization of the ROR coefficient should also be considered.

We may also extend the method to consider coding inside the network, and not only at the edge nodes. We should investigate applying the extended method in wireless Mobile Ad-hoc Networks, aiming not only at fault-tolerance but also at saving energy [Lun06], [Pakzad05], [Tuninetti05], [060724].

Appendix A. 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.

Section A.1. File management operations

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

```
MFIL* mopen(char *name, int stripeUnitSz);
void mclose(MFIL *f);
void mchsize(MFIL *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 global file name format is a simple semi-colon separated concatenation of local subfile names (including their hostnames) in the right order. The format is as follows:

```
"<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.

Section A.2. Data access operations

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

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

```

unsigned numberOfBlocks,
long offsets[],
char *buffers[],
unsigned sizes[]);

```

The data access requests are blocking and non-collective. The functions *mreadc* and *mwritelc* are the optimized versions of the *mread* and *mwritel* functions. The multiple block data access operations *mreadb* and *mwritelb* are optimized. The *numberOfBlocks* argument in the *mreadb* and *mwritelb* operations specifies the number of blocks to be accessed by the operation in the logical file. The information about each block must 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.

Section A.3. Error management operations

Error management is provided by the *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 the *open*, *close*, *creat*, *unlink*, *ftruncate*, *lseek*, *write*, and *read* functions of the local OS. The function *prioerrora* is a collective operation which prints the error statistics to the standard output of the application.

Appendix B. Congestion graph coloring heuristic approach

The search for a liquid schedule requires the partitioning of the traffic into sets of mutually non-congesting transfers. This problem can also be represented as a conflict graph coloring problem [Beauquier97]. Vertices of the conflict (or congestion) graph represent the transfers. Edges between vertices represent congestions between the transfers.

Figure 62 shows a congestion graph that corresponds to the all-to-all traffic pattern across the network of Figure 28, which consists of 25 transfers. These transfers are shown in Figure 29 as pictograms and in Figure 30 as sets of communication links. The vertices of the congestion graph are labeled with two indexes (i, j) . Vertex (i, j) represents the transfer from the sending node i to the receiving node j . Vertex $(4,1)$, for example, represents the transfer from node t_4 to node r_1 , denoted as $\text{---} \text{---} \text{---}$ in Figure 29 and as $\{l_{t_4}, l_{ba}, l_{r_1}\}$ in Figure 30.

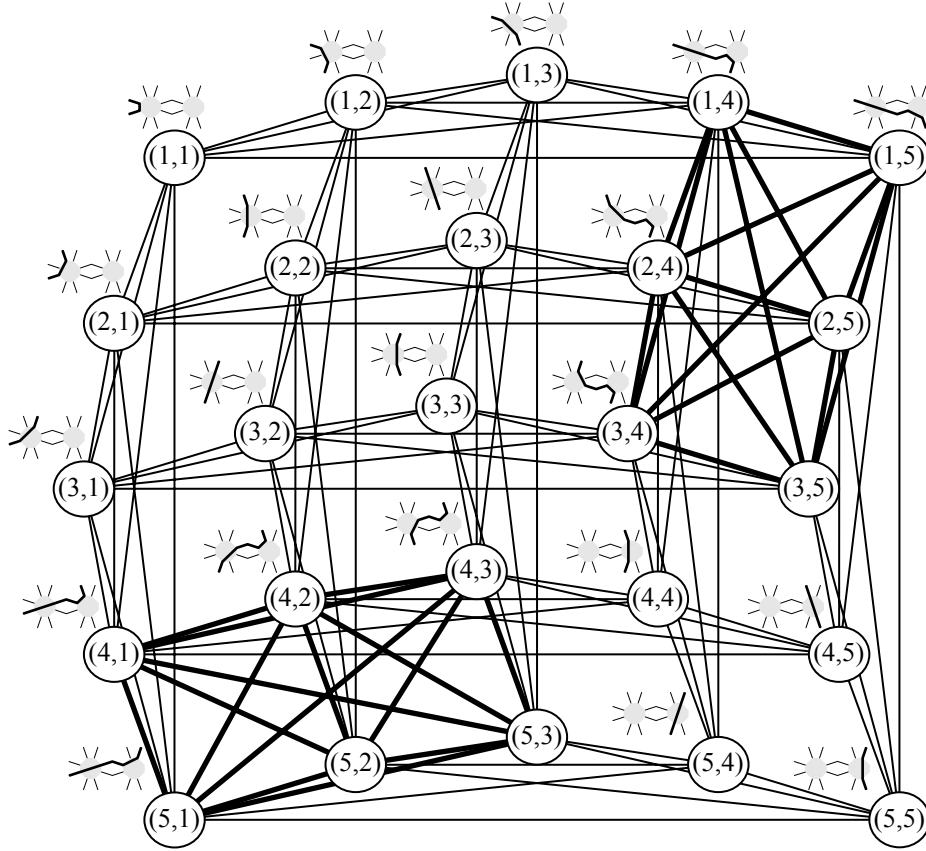


Figure 62. Congestion graph corresponding to the traffic pattern of Figure 29 across the network of Figure 28; the vertices of the graph represent the 25 transfers; the edges represent congestions between the transfers

An edge between two vertices is present if one or more links are shared between the two corresponding transfers. Therefore each edge of the congestion graph can be labeled by the

link(s) causing the congestion. In Figure 62 we marked in bold the edges which occurred due to the bottleneck links l_{ab} and l_{ba} (see the network diagram in Figure 28). The 15 bold edges between any two of the following vertices (1,4), (1,5), (2,4), (2,5), (3,4), (3,5) represent the congestions due to the bottleneck link l_{ab} . The other 15 bold edges between the vertices (4,1), (4,2), (4,3), (5,1), (5,2), (5,3) represent the congestions due to the bottleneck link l_{ba} .

According to the graph coloring problem, the vertices of the graph must be colored such that no two vertices have the same color if they are connected. The objective of the graph coloring problem is to properly color the graph using the minimum number of colors. The graph coloring problem has a complexity of NP-complete, but various heuristic algorithms exist.

Once the graph is properly colored, vertices having the same color can represent a time frame of the liquid schedule, since the corresponding transfers can be carried out simultaneously without congestions. Whenever a liquid schedule exists, an optimal solution of the graph coloring problem corresponds to a liquid schedule and the chromatic number of the graph's optimal coloring is therefore the length of the liquid schedule. A heuristic graph coloring algorithm, however, may find solutions requiring more colors than the optimal solution, therefore reducing therefore the throughput of the corresponding schedule.

The congestion graphs corresponding to the traffic patterns across the network of the Swiss-T1 cluster supercomputer have a relatively low density of edges (see Figure 63). For example, an all-to-all data exchange on the Swiss T1 cluster with 32 transmitting and 32 receiving processors results in a graph with $32 \times 32 = 1024$ vertices and 48704 edges (the corresponding complete graph K_{1024} has 523776 edges that is eleven times denser).

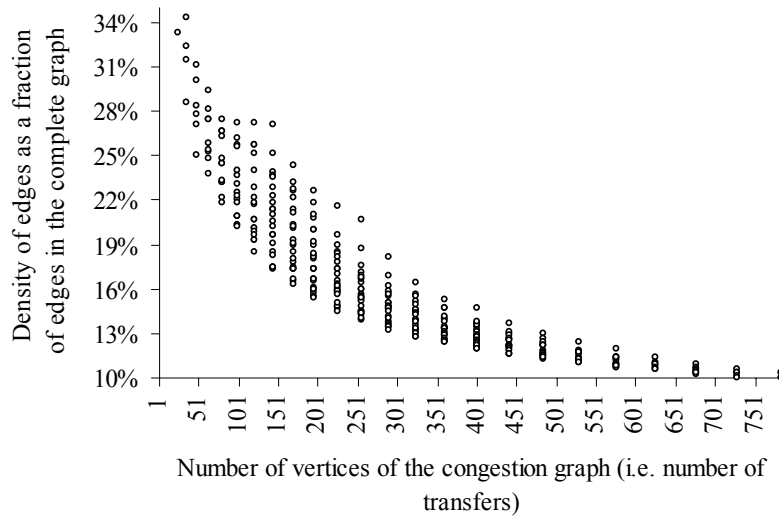


Figure 63. Number of edges in the 362 congestion graphs corresponding to the traffic patterns of Figure 40 and Figure 41

We compared our method of finding a liquid schedule with the results obtained by applying the heuristic fast graph coloring algorithm DSatur [Brelaz79], [Culberson97], [Rolland-Balzon02], [Trick94], which carries out the steps shown in Table 3.

Table 3. DSatur graph coloring heuristic algorithm

1.	Arrange the vertices by decreasing order of degrees.
2.	Color a vertex of maximal degree with color 1.
3.	Choose a vertex with a maximal saturation degree (defined as the number of different colors to which it is adjacent). If there is an equality, priority is given to the vertex having the maximal degree in the uncolored sub-graph.
4.	Color the chosen vertex with the least possible (lowest numbered) color.
5.	If all the vertices are colored, stop. Otherwise, return to step 3.

Although the heuristic algorithm is fast, it often induces additional colors. For 26% of all test traffic patterns (shown in Figure 40 and Figure 41) across the network of the Swiss-T1 cluster supercomputer (Figure 39 and Table 2), the heuristic graph coloring algorithm induces a loss in the overall communication throughput. For the 94 traffic patterns (out of 362) affected by the heuristic algorithm, Figure 64 shows the reduction in throughput. The losses occur due to the additional, unnecessary colors introduced by the heuristic graph coloring algorithm.

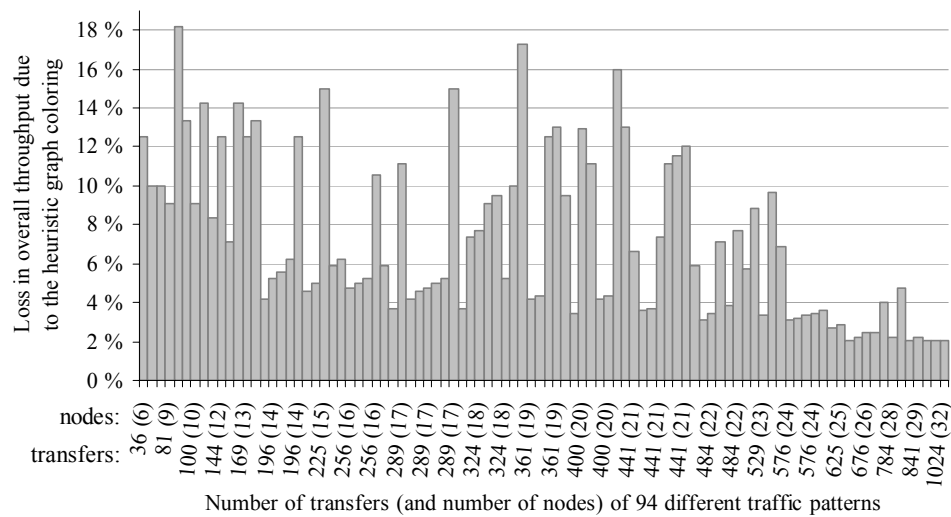


Figure 64. Loss in throughput induced by schedules computed with the DSatur heuristic algorithm

For 66% of the considered topologies (or traffic patterns), the performance loss is between 2% and 9%. For the remaining 34% of topologies, the loss of performance is between 9% and 19%.

The computation time of the heuristic algorithm is polynomial and is therefore faster than the algorithm searching for the liquid schedule. However, for massive data exchanges, the cost of the liquid scheduling algorithm, which usually does not exceed 1/10 of a second (see Appendix C), is negligible compared with the gain in communication time attained by liquid schedules.

The liquid scheduling algorithm can efficiently color a congestion graph if additional information about the routes of transfers (represented by a vertex in the congestion graph) is also provided. The algorithm does not rely only on information about the conflicts between all possible pairs of transfers. The liquid scheduling relies also on the fact that the transfers are sets consisting of communication links.

For example, the fast algorithm for retrieving the full teams of a traffic first retrieves all full teams of the traffic's skeleton (see Subsection 3.6.2). The traffic skeleton, in turn, comprises the transfers using the bottleneck links of the network. It is an example in which the algorithm uses not only information on the conflicts between the transfers (i.e. the congestion graph), but also information on the content of the transfers.

Therefore, in the case of the liquid scheduling algorithm, the transfers cannot be abstracted into vertices of a graph. A graph provides only information about the presence or the absence of a conflict between any given pair of vertices. By limiting the input of the problem to only a conflict graph, the liquid scheduling algorithm would not receive information about the bottleneck links and therefore could not operate. Therefore the liquid scheduling is not a general graph coloring algorithm.

Appendix C. Comparison of the liquid scheduling algorithm with Mixed Integer Linear Programming

The problem of liquid scheduling can be formulated and solved with Mixed Integer Linear Programming (MILP); see [CPLEX02], [Fourer03]. The problem of minimizing the number of timeframes (and/or wavelengths) can be represented as an MILP objective.

We represent the network as a directed graph $G = ((V(G), E(G)))$. The routing is represented by a parameter $R_e^{s,d}$, indexed above by the source and destination nodes ($s \in V(G)$, $d \in V(G)$) and below by the network link $e \in E(G)$. This parameter indicates whether the transmission (flit stream flow for wormhole switching or lightpaths for optical networks) from the source s to the destination d traverses the link e . It is set to 1 if the transmission (s, d) uses link e and to 0 otherwise.

$$R_e^{s,d} \in \{0, 1\} \quad (30)$$

Also given is the traffic pattern X comprising pairs of communication nodes (s, d) . The transmissions $(s, d) \in X$ of the traffic pattern are allocated to timeframes $t \in \{1 \dots T\}$ according to the variable $A_t^{s,d}$. The variable $A_t^{s,d}$ is 1 if the transmission $(s, d) \in X$ is allocated to the timeframe t and is 0 otherwise.

$$A_t^{s,d} \in \{0, 1\} \quad (31)$$

The objective is to allocate the transfers such that the number T is minimized. We may formulate this as follows:

Minimize: T

subject to:

$$0 \leq \sum_{(s,d) \in X} A_t^{s,d} \cdot R_e^{s,d} \leq 1 \quad \forall e \in E(G), \forall t \in \{1 \dots T\} \quad (32)$$

and

$$\sum_{t=1}^T A_t^{s,d} = 1 \quad \forall (s, d) \in X \quad (33)$$

Relation (32) represents the simultaneity constraint: number of the transfers in a timeframe t using a given network link e can be either 0 or 1 (for all links $e \in E(G)$ and timeframes $t \in \{1 \dots T\}$). Equation (33) represents the partitioning constraint. The traffic X is partitioned into time frames of a schedule, therefore each transfer (s, d) of the traffic must be assigned to one and only one time slot.

The present problem is hard to solve with MILP. For the 362 test bed topologies introduced in Subsection 3.8.1 (see Figure 40 and Figure 41), we compared the Mixed Integer

Linear Programming (MILP) method with the liquid scheduling algorithm. The computation speed of MILP is far below that of our liquid scheduling algorithm (Figure 65). Our algorithm is on average about 4000 times faster than MILP.

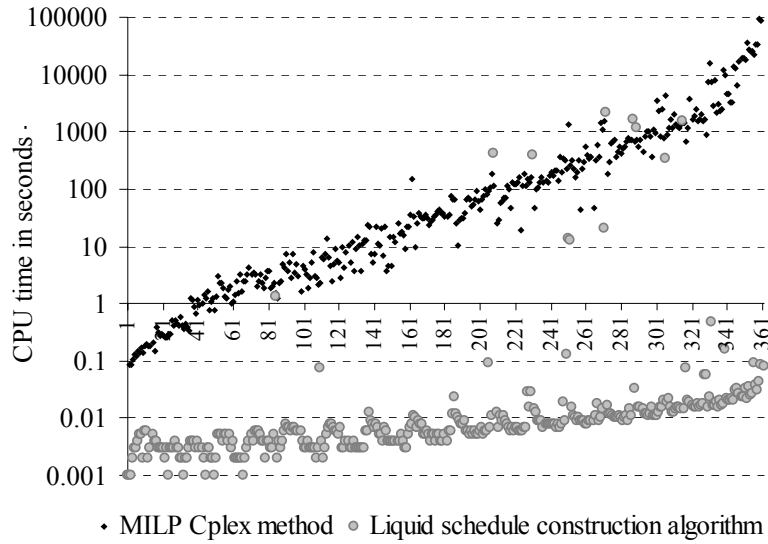


Figure 65. Running times for computing liquid schedules with the MILP Cplex method and with the liquid schedule construction algorithm

Appendix D. Assembling a liquid schedule: Considering teams of the reduced traffic instead of the teams of the original traffic

The basic algorithm for constructing liquid schedules (see Subsection 3.7.2) assumes that a liquid schedule can be assembled by considering various combinations of teams of the original traffic. For example if a certain combination of teams of X is already selected (from the set $\mathfrak{T}'(X)$ of all teams of X) and there still remains a subtraffic X_{sub} of not yet carried out (scheduled) transfers, then, according to the basic algorithm, the teams of the original traffic $\{A \in \mathfrak{T}'(X) \mid A \subset X_{sub}\}$ must be considered in the choice of the next timeframe (Subsection 3.7.2, equation (11) and Figure 38).

The following two theorems prove that we can restrict our choice of possibilities when selecting successive time frames without affecting the solvability, meaning that if the solution space is not empty then at least one solution will be found.

Theorem 1 shows that by removing a time frame (i.e. a team) from a liquid schedule, we form a new liquid schedule on the remaining traffic. The remaining traffic may have additional bottlenecks. For example, in Figure 35, from time frame 3 on, links l_{i3} and l_{r3} appear as additional bottlenecks and from time frame 5 on, the links l_{i4} and l_{r5} also appear as additional bottlenecks (making the total number of bottlenecks equal to 6).

Newly emerged bottlenecks allow us to limit our choice from a large set of teams of the original traffic to a smaller set of teams of the reduced traffic. According to theorem 2, this does not affect the solvability. The statement appears logically clear (in terms of the remaining transmissions to be carried out). The exercise of giving a formal proof is provided for the sake of keeping the mathematical model complete.

THEOREM 1. Let α be a liquid schedule on X and A be a *time frame* of α . Then $\alpha - \{A\}$ is a liquid schedule on $X - A$.

PROOF. By definition, a schedule is liquid if its length is equal to the duration of the traffic (equation (9) of Subsection 3.7.1). Clearly A is a team of X . Remove the team A from X so as to form a new traffic $X - A$. The duration of the new traffic $X - A$ is the load of the bottlenecks in $X - A$.

The load of bottlenecks of X in X is the highest and therefore is more than the load of all other links by at least 1. By removing a team of X the load of all bottleneck links is reduced by 1. Therefore, a link which is a bottleneck in X is still a bottleneck in $X - A$. Thus the bottlenecks of $X - A$ include the bottlenecks of X .

The load of a bottleneck of X is decreased by one in the new traffic $X - A$ and therefore the duration of $X - A$ is the duration of X decreased by one, i.e. $\Lambda(X - A) = \Lambda(X) - 1$. The

schedule α without the element A is a schedule for $X - A$ by the definition of a schedule given in Subsection 3.7.1 (a schedule is a collection of simultaneities partitioning the traffic). Obviously $\#(\alpha - \{A\}) = \#(\alpha) - 1$. Therefore the new schedule $\alpha - \{A\}$ has as many time frames as the duration of the new traffic $X - A$ is. Hence $\alpha - \{A\}$ is a liquid schedule on $X - A$. ■

In other words, if the traffic has a liquid schedule, then a schedule reduced by one team is a liquid schedule on the reduced traffic. The repeated application of Theorem 1 implies that any non-empty subset of a liquid schedule is a liquid schedule on the correspondingly reduced traffic.

THEOREM 2. If, by traversing each team A of a traffic X none of the sub-traffics $X - A$ has a liquid schedule, then the traffic X does not have a liquid schedule either.

PROOF. Let us prove the theorem by contradiction and suppose that X has a liquid schedule α . Then, a time frame A of α shall be a team of X . Furthermore, according to Theorem 1, the schedule $\alpha - \{A\}$ shall be a liquid schedule for $X - A$. Therefore, for at least one team A of X , the sub-traffic $X - A$ has a liquid schedule. This proves the theorem. ■

Theorem 2 implies that if X has a liquid schedule, at least one team A of X will be found such that the sub-traffic $X - A$ has a liquid schedule β . Obviously $\beta \cup \{A\}$ will be a liquid schedule for X .

Instead of considering for the set of possible time frames all teams of the original traffic included in the current sub-traffic X_{sub} , i.e. $\{A \in \mathfrak{T}'(X) \mid A \subset X_{sub}\}$, we propose to consider for the set of possible time frames (at the current node of the construction tree) all teams of the current sub-traffic, i.e. $\mathfrak{T}'(X_{sub})$.

By induction, theorem 2 implies that if a solution for X (i.e. a liquid schedule on X) exists, then this algorithm will necessarily find it.

Since the teams of the current sub-traffic X_{sub} together with the bottlenecks of the original traffic X must also use the additional bottlenecks of X_{sub} , the number of teams of the current subtraffic $\mathfrak{T}'(X_{sub})$ is smaller or equal to the number of teams of the original traffic whose transfers belong to the current subtraffic:

$$\#(\mathfrak{T}'(X_{sub})) \leq \#(\{A \in \mathfrak{T}'(X) \mid A \subset X_{sub}\}) \quad (34)$$

Therefore fewer possible teams need to be considered when building the schedule. The solution space is not affected, since theorem 2 is valid at any level of the search tree.

The construction algorithm traverses the tree in depth-wise order (Figure 38). A solution is found when the current node (sub-traffic) forms a single team. The path from the root to that leaf node forms the set of teams yielding the liquid schedule. The example of a liquid schedule of Figure 35 shows that each timeframe incorporates additionally also the bottlenecks (marked in bold) of the remaining reduced traffic. Therefore each timeframe is also a team of the reduced

traffic. A node in the construction tree is a dead end if the corresponding sub-traffic does not have a team (see Figure 36 and Figure 37 for example). In that case the algorithm backtracks and evaluates other choices. Evaluation of all choices ultimately leads to a solution if it exists.

Appendix E. Assembling a liquid schedule: Considering full teams of the reduced traffic instead of all its teams

Assuming the liquid schedule construction algorithm of Subsection 3.7.3, we can build a liquid schedule by further limiting the choice of teams of the reduced subtraffic to its full teams.

Let us modify a given liquid schedule so as to convert one of its teams into a full team. Let a traffic X have a liquid schedule α . Let A be a time frame of α . If A is not a full team of X , then by moving the necessary transfers from other time frames of α , we can convert the team A into a full team. Clearly, by doing so, the properties of liquidity (partitioning, simultaneousness and length) of α are not affected. Therefore if X has a solution then it has also a solution for which any one of its selected time frames is full.

Therefore, if it is possible to built a liquid schedule, then it can be built by a choice of a full team A of the current reduced traffic X_{sub} . Thus, the choice of the teams in the construction tree of Figure 38 may be narrowed from the set of all teams to the set of full teams only, i.e. $\aleph(X_{sub}) = \mathfrak{F}(X_{sub})$. This yields the optimization of Subsection 3.7.4 (equations (13), (14) and (15)). An efficient algorithm for retrieving the set of all full teams $\mathfrak{F}(X_{sub})$ is presented in Table 1.

Figure 35 shows a liquid schedule constructed with full teams. It can be easily verified that, for any given timeframe, all transfers of the following timeframes congest with at least one transfer of that timeframe.

Appendix F. Overall overview of all liquid schedule construction optimizations

Liquid scheduling permits optimum partitioning of a traffic into subsets of non-congesting transfers. Its construction relies on the fast retrieval of full teams (saturated collection of non-congesting transfers using all bottleneck links of the network) presented in Section 3.6, and on their assembly into a schedule as presented in Section 3.7. The overall liquid schedule construction algorithm is briefly outlined in Table 4.

Table 4. Overall overview of liquid schedule construction algorithm and its all relevant optimizations

1.	Full teams are enumerated by recursively partitioning the solution space using inclusion and exclusion constraints:
1.1.	The blank optimization identifies empty partitions at early stages of the search tree;
1.2.	The idle optimization identifies partitions containing no full teams at early stages of the search tree;
1.3.	The skeleton optimization speeds up the retrieval of full teams, first by considering only the transfers necessary to keep all bottleneck links busy and then by adding up other non-congesting transfers.
2.	We construct liquid schedules by partitioning the traffic into teams:
2.1.	The construction of the liquid schedule is accelerated by limiting the choice at each time frame to the teams, which must incorporate in addition also the newly emerging bottleneck links (i.e. teams of the reduced traffic);
2.2.	By additionally limiting the choice to only full teams of the reduced traffic we further speed up the construction of the liquid schedule.

Measurements on real traffic carried out on 362 different network configurations of the Swiss-T1 cluster supercomputer (Section 3.8) have shown that by applying the liquid scheduling algorithm of Table 4, we are able to increase the overall communication throughput by a factor between 1.5 and 2 (see Figure 66).

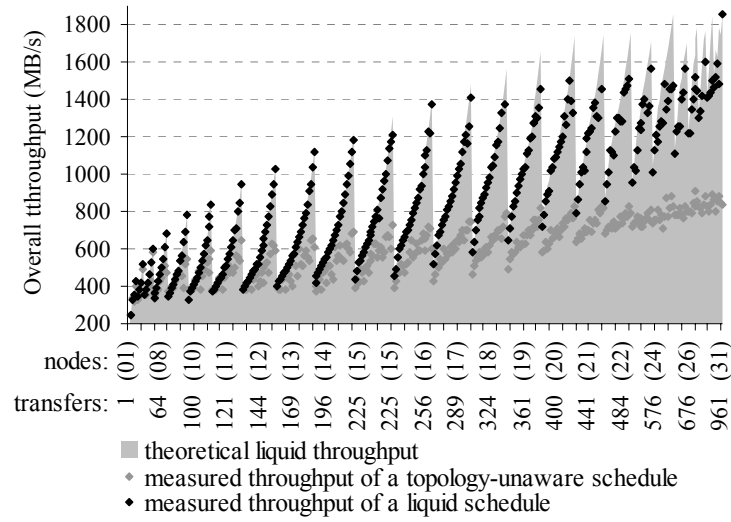


Figure 66. The overall measured throughputs of hundreds of different traffic patterns carried out according to both a liquid schedule and a topology unaware schedule

Appendix G. Probability of simultaneous link failures in multi-path routing patterns

When introducing in Subsection 4.3.1 equation (23) for the Redundancy Overall Requirement (ROR), we rely on the assumption of a single link failure. Only if this assumption holds, ROR is the proportionality coefficient for the number of redundant packets that the sender needs to transmit during a given communication time, in order to protect the communication against randomly occurring link failures.

In this appendix, we delimit the conditions under which the single link failure assumption holds. We also analyze how our theory should be extended in order to consider also the probability of multiple simultaneous link failures.

Section G.1. Limitations of the single link failure assumption

The probability that the single link failure assumption does not hold depends on the failure rate of one single link, the duration of a single failure and the number of links in the network (which is proportionally increasing the overall failure rate in the network). We assume that all links have an equal failure probability and duration.

We will consider a Poisson process for evaluating the probability of overlapping failures of two different links. This is the probability that our assumption of a single link failure does not hold.

The events in the Poisson process occur randomly in time.

Let X be the interarrival time between two events.

Let G denote the right-tail distribution function of X :

$$G(t) = P(X > t) \quad t \geq 0 \quad (35)$$

where $P(X > t)$ is the probability that the interarrival time X between two failures is longer than t

It is known [[Wiki-Poisson06](#)], [[Siegrist01](#)] that the right-tail distribution function G of equation (35) is an exponential function and is expressed as follows:

$$G(t) = P(X > t) = e^{-r \cdot t} \quad t \geq 0 \quad (36)$$

where r is the rate parameter

In our model r is the overall rate of link failures in the network.

Let $F(t)$ denote the probability that the interarrival time is below or equal to t . Then according to equation (36):

$$F(t) = P(X \leq t) = 1 - G(t) = 1 - e^{-r \cdot t} \quad t \geq 0 \quad (37)$$

Let for the sake of simplicity assume that all link failures last a fixed period of time equal to t . Let N be the number of links in the network. Let d be the average time between two link failures (e.g. between the two points at which begin the two respective faulty states each lasting a duration t). The overall mean rate r of network link failures is then computed as follows:

$$r = N \cdot \frac{1}{d} \quad (38)$$

Two consecutive link failures will overlap if the interarrival time between these two failures is smaller than the failure duration t . The probability of this is $F(t)$. Therefore, according to equations (37) and (38):

$$F(t) = 1 - e^{-N \cdot \frac{t}{d}} \quad (39)$$

The chart of Figure 67 shows F as a function of t , i.e. the probability that the interarrival time between two consecutive failures is less than t . If t is considered as a failure duration, then the chart represents the probability of overlapping of two consecutive link failures as a function of the failure duration. In this example the average time between failures of one single link is one hour and there are 50 links in the network. For example if the duration of a single link failure is 1 second, then the probability of overlapping of two consecutive failures is 2.74%.

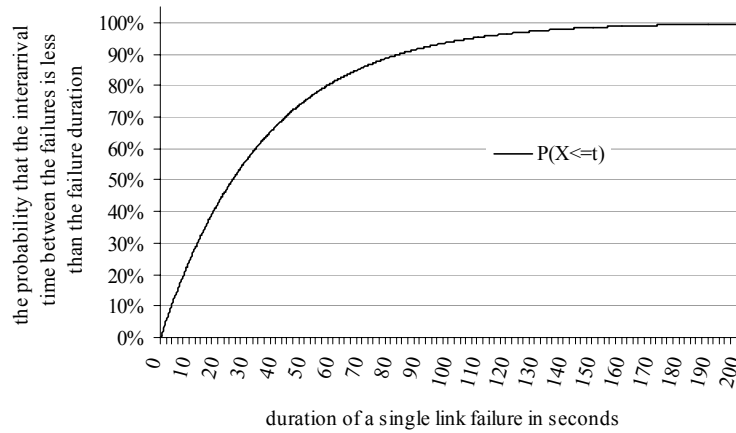


Figure 67. The probability that the interarrival time between two consecutive failures in a Poisson process is less than a given time, $r = 1/3600$, $N = 50$

Let the Failures Overlapping Probability (FOP) be the acceptable probability that two consecutive link failures overlap in time. We then say that if the probability of overlapping of two consecutive failures is below FOP, then our assumption of single link failure (Subsection 4.3.1) holds and our theory of Chapter 4 is valid. Let us compute the maximal number of links in the network ensuring that the probability of overlapping of two successive failures does not

exceed the acceptable FOP. For a given average time d between two failures of the same link, and for a given failure time t of a single link failure, according to equation (39), we have:

$$FOP = 1 - e^{-N \cdot \frac{t}{d}} \quad (40)$$

$$\left(e^{-\frac{t}{d}} \right)^N = 1 - FOP \quad (41)$$

$$N = \frac{\ln(1 - FOP)}{\left(-\frac{t}{d} \right)} \quad (42)$$

and therefore:

$$N = -\frac{d}{t} \cdot \ln(1 - FOP) \quad (43)$$

With equation (43) one can verify that if the acceptable FOP is 5%, and that in average each network link fails once a day for a period of 5 seconds, the communication flow can follow a routing footprint consisting of nearly 900 links and the probability of even partial overlapping of failures will stay below the acceptable upper limit. Or in other words with 900 underlying network links the probability of partial or full overlapping of two consecutive link failures does not exceed 5%.

Section G.2. *Extension of ROR for considering also the overlapping failures*

For frequent and long failures the equation (23) of ROR based on the assumption of a single link failure at a time may not be valid anymore (see Subsection 4.3.1). In this case the equation of ROR must be extended.

Assuming that the probability of overlapping of three simultaneous failures is essentially zero, let us denote by k_1 the sum of fractions of time during which only single link failures occur and by k_2 the sum of fractions of time during which two links are in a faulty state. The coefficients k_1 and k_2 are fractions relative to the total failure time (i.e. the total sum of times during which at least one link is in a faulty state). Therefore:

$$k_1 + k_2 = 1 \quad (44)$$

The ROR coefficient, which considers also the possibility of overlapping of two faulty states, must be therefore computed according the following equation:

$$ROR = k_1 \cdot ROR_1 + k_2 \cdot ROR_2 \quad (45)$$

In equation (45) ROR_1 is an ROR metric assuming only non-overlapping failures. Therefore ROR_1 can be computed according to equation (23) as follows:

$$ROR_1 = \sum_{l \in L | t \leq r(l) < 1} \left(\frac{FEC_{r(l)}}{FEC_t} - 1 \right) \quad (46)$$

Do not confront t and r with the temporary notations used in Section G.1. In equation (46) and in the further equations (as introduced in Section 4.3), t represents the tolerable limit of packet losses (the constant tolerance of the streaming media) and r represents the routing function, i.e. the fraction of flow traversing through the given link under the given routing.

R_2 in equation (45) is an ROR metric assuming exclusively overlapping failures of two links, i.e. all states during which two link failures are taking place simultaneously. The coefficients k_1 and k_2 are the two respective weights.

Assume now that two links l_1 and l_2 are in a faulty state at the same time. According to the routing function r , when all network links were operational, the first link carried out the $r(l_1)$ portion of the traffic and the second link the $r(l_2)$ portion respectively. Clearly:

$$0 \leq r(l_1) \leq 1 \quad (47)$$

$$0 \leq r(l_2) \leq 1 \quad (48)$$

If the links are completely parallel and independent, meaning that no part of flow passing through link l_1 passes after also through link l_2 and vice versa, then the loss rate observed at the receiver during the time of the simultaneous failure of these two links will be the sum of the two fractions:

$$loss = r(l_1) + r(l_2) \quad (49)$$

If the links are completely sequential, meaning that the flow of one link completely passes through the other link then:

$$loss = \max(r(l_1), r(l_2)) \quad (50)$$

The loss rate $loss(l_1, l_2)$ as a function of two links l_1 and l_2 , observed at the receiver during the time of simultaneous failures of these two links respects therefore the following relations:

$$\max(r(l_1), r(l_2)) \leq loss(l_1, l_2) \leq r(l_1) + r(l_2) \quad (51)$$

The ROR_2 coefficient is the sum of the transmission rate increment factors across all possible pairs of simultaneously failing network links (l_1, l_2) . The transmission rate increment factor (TRIF) for a simultaneous failure of a single pair of two links l_1 and l_2 is expressed as follows:

$$TRIF(l_1, l_2) = \frac{FEC_{loss(l_1, l_2)}}{FEC_t} - 1 \quad (52)$$

The ROR_2 coefficient therefore can be written as follows, as a sum across all possible pairs of (l_1, l_2) :

$$ROR_2 = \frac{1}{2} \cdot \sum_{l_1 \in L | t \leq r(l_1) < 1} \sum_{\substack{l_2 \in L | l_1 \neq l_2 \\ t \leq r(l_2) < 1}} \left(\frac{FEC_{loss(l_1, l_2)}}{FEC_t} - 1 \right) \quad (53)$$

Because the nested sums of equation (53) counting each pair twice, we include a compensating coefficient $1/2$.

Then, finally the global ROR coefficient which considers also simultaneous failures of two links can, according to equations (46) and (53), be rewritten as follows:

$$ROR_2 = k_1 \cdot \sum_{l \in L | t \leq r(l) < 1} \left(\frac{FEC_{r(l)}}{FEC_t} - 1 \right) + \frac{k_2}{2} \cdot \sum_{l_1 \in L | t \leq r(l_1) < 1} \sum_{\substack{l_2 \in L | l_1 \neq l_2 \\ t \leq r(l_2) < 1}} \left(\frac{FEC_{loss(l_1, l_2)}}{FEC_t} - 1 \right) \quad (54)$$

In the future, we intend to search for an analytical expression for the coefficients k_1 and k_2 (equations (44) and (45)). We also need a method for computing the function $loss(l_1, l_2)$ (equation (51)).

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, "SEMPAR: high-performance remote parallel I/O over SRB Cluster Computing and the Grid", International Symposium on CCGrid, 9-12 May 2005, vol. 1, pp. 366-373
- [Altman01] Eitan Altman, Chadi Barakat, Victor M. Ramos, "Queueing analysis of simple FEC schemes for IP telephony", INFOCOM 2001, Apr 22-26, vol. 2, pp. 796-804
- [Ayad97] N.M.A. Ayad, F.A. Mohamed, "Performance analysis of a cut-through vs. packet-switching techniques", 2nd IEEE Symposium on Computers and Communications, 1-3 July 1997, pp. 230-234
- [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
- [Beauquier97] B. Beauquier, J.C. Bermond, L. Gargano, P. Hell, S. Pérennes, U. Vaccaro, "Graph Problems Arising from Wavelength-Routing in All-Optical Networks", IPPS'97: WOCS'97 - 2nd IEEE Workshop on Optics and Computer Science, April 1997
- [Bermond96] J.-C. Bermond, L. Gargano, S. Perennes, A. A. Rescigno, and U. Vaccaro, "Efficient collective communication in optical networks", ICALP'96 - Lecture Notes in Computer Science 1099, Springer Verlag, Berlin 1996, pp. 574-585
- [Bermudez06] Aurelio Bermúdez, Rafael Casado, Francisco J. Quiles, José Duato, "Fast routing computation on InfiniBand networks", IEEE Transactions on Parallel and Distributed Systems, March 2006, vol. 17, issue 3, pp. 215-226

- [Boden95] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, Wen-King Su, "Myrinet: a gigabit per second local area network," IEEE Micro, February 1995, vol. 15, issue 1, pp. 29-36
- [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 1999, pp. 12-15, <http://sawwww.epfl.ch/SIC/SA/publications/SCR99/scr11-page12.html>
- [Brelaz79] Daniel Brelaz, "New Methods to Color the Vertices of a Graph", Communication of the ACM, April 1979, Vol. 22, Issue 4, pp. 251-256
- [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
- [Byun00] Chansup Byun, Christopher Duncan, "A Comparison of Job Management Systems in Supporting HPC ClusterTools", SUPeRG, Vancouver, Fall 2000, <http://www.indiana.edu/~uits/rac/mgmt.pdf>
- [Caragiannis02] I. Caragiannis, Ch. Kaklamanis, P. Persiano, "Wavelength Routing in All-Optical Tree Networks: A Survey", Bulletin of the European Association for Theoretical Computer Science, 2002, Vol. 76, pp. 104-112
- [CERN04] Large Hadron Collider, Computer Grid project, CERN, 2004, <http://lcg.web.cern.ch/LCG/>
- [Chan01] S.-H. Gary Chan, "Operation and cost optimization of a distributed server architecture for on-demand video services", IEEE Communications Letters, September 2001, Vol. 5, Issue 9, pp. 384-386
- [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
- [Chiu89] Dah-Ming Chiu, Raj Jain, "Analysis of the increase and decrease algorithms for congestion avoidance in computer networks", Computer Networks and ISDN Systems, 1989, Vol. 17, pp. 1-14
- [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

- [Colajanni99] M. Colajanni, B. Ciciani, F. Quaglia, "Performance Analysis of Wormhole Switching with Adaptive Routing in a Two-Dimensional Torus", Euro-Par'99, Toulouse, France, Springer-Verlag, August – September, 1999
- [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
- [CPLEX02] ILOG CPLEX 8.0, User's Manual, ILOG SA, Gentilly, France, 2002
- [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
- [Culberson97] Joseph Culberson, "Graph Coloring Programs Manual", University of Alberta, Canada, 1997,
<http://www.cs.ualberta.ca/~joe/Coloring/Colorsrc/manual.html>
- [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
- [Duato99] J. Duato, A. Robles, F. Silla, R. Beivide, "A comparison of router architectures for virtual cut-through and wormhole switching in a NOW environment", SPDP'99 - IEEE Symposium on Parallel and Distributed Processing, 12-16 April 1999, pp. 240-247
- [Dvorak05] Vaclav Dvorak, "Scheduling Collective Communications on Wormhole Fat Cubes", 17th International Symposium on Computer Architecture and High Performance Computing, 24-27 Oct 2005, pp. 27-34
- [EWSD04] Siemens Carrier Networks, EWSD Digital Switching System, April 2004,
<http://www.icn.siemens.com/carrier/products/switching/ewsdsw.html>
- [Fourer03] Robert Fourer, David M. Gay, Brian W. Kernighan, "AMPL: A Modeling Language for Mathematical Programming", 2nd edition, Thomson Learning Brooks/Cole, 2003
- [Fujita03] Naoyuki Fujita, Hirofumi Ookawa, "Storage devices, local file system and crossbar network file system characteristics, and 1 terabyte file I/O benchmark on the Numerical Simulator III", MSST'03 - 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies, 7-10 April 2003, pp. 72-76,
http://storageconference.org/2003/papers/10_Fujita-Storage.pdf
- [Gabrielyan00] Emin Gabrielyan, "[Parallel I/O for SwissTx](#)", Swiss-Tx Progress Meeting, EPFL, Lausanne, Switzerland, February 21, 2000
- [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
- [Gabrielyan03] Emin Gabrielyan, Roger D. Hersch, "Network Topology Aware Scheduling of Collective Communications", ICT'03 - 10th International Conference on Telecommunications, 2003, pp. 1051-1058

- [Gabrielyan04A] Emin Gabrielyan, Roger D. Hersch, “Liquid Schedule Searching Strategies for the Optimization of Collective Network Communications”, 18th International Multi-Conference in Computer Science & Computer Engineering: PCC’04 - Pervasive Computing and Communications, Las Vegas, USA, 21-24 June 2004, CSREA Press, vol. 2, pp. 834-848
- [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://sawwww.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
- [Gruber01] Ralf Gruber, Pieter Volgers, Alessandro De Vita, Massimiliano Stengel, “Commodity computing results from the Swiss-Tx project”, Electronic Notes in Future Generation Computer Systems, 2001, Vol. 1
- [Gruber02] Ralf Gruber, Alessandro de Vita, Massimiliano Stengel, Trach-Minh Tran, “Application Dedicated Clustering”, EPFL Supercomputing Review, May 2002, pp. 37-40, http://sawwww.epfl.ch/SIC/SA/SPIP/Publications/IMG/pdf/scr13_page37.pdf
- [Gruber05] Ralf Gruber, “High Performance Computing Methods”, Swiss-Tx and Swiss Grid, 2005, http://pleiades.epfl.ch/~rgruber/cours/C5_6part1.0.ppt
- [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
- [H323] H.323 Standards, <http://www.openh323.org/standards.html>
- [Halmos74] Paul R. Halmos, Naive Set Theory, Springer-Verlag New York Inc, 1974, pp. 26-29
- [Hassaine02] Omar Hassaine, “HPC Administration Tips and Techniques”, CPR Engineering-HPC, Sun BluePrints OnLine, October 2002, <http://www.sun.com/blueprints/1002/817-0079-10.pdf>
- [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, February 1995, Vol. 15, Issue 1, pp. 37-45
- [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
- [InfiniBand] InfiniBand Trade Association, <http://www.infinibandta.org/>
- [ITU-D-VOIP03] "The Essential Report on IP Telephony", by the Group of Experts on IP Telephony / ITU-D, 2003, http://www.itu.int/ITU-D/e-strategy/publications-articles/pdf/IP-tel_report.pdf
- [Jagannathan02] S. Jagannathan, A. Tohmaz, A Chronopoulos, H.G. Cheung, "Adaptive admission control of multimedia traffic in high-speed networks", IEEE International Symposium on Intelligent Control, 27-30 Oct 2002, pp. 728-733
- [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
- [Kartalopoulos00] Stamatios V. Kartalopoulos, "What is WDM technology", Technology and Trends for International Optical Engineering Community, November 2000, <http://www.spie.org/web/oer/november/nov00/wdm.html>
- [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 1999, pp. 3-11, <http://sawwww.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", HPC'99 - High-Performance Computing Conference, 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>
- [Liu01] Pangfeng Liu, Jan-Jan Wu, Yi-Fang Lin, Shih-Hsien Yeh, "A simple incremental network topology for wormhole switch-based networks", 15th International Parallel and Distributed Processing Symposium, 23-27 April 2001, pp. 6-12
- [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
- [Loh96] P.K.K. Loh, Wen Jing Hsu, Cai Wentong, N. Sriskanthan, "How network topology affects dynamic loading balancing", Parallel & Distributed Technology: Systems & Applications, Fall 1996, Vol. 4, Issue 3, pp. 25-35
- [Luby02] Michael Luby, "LT codes", FOCS'02, November 16-19, pp. 271-280
- [Lun06] D. S. Lun, P. Pakzad, C. Fragouli, M. Medard, R. Koetter, An Analysis of Finite-Memory Random Linear Coding on Packet Streams, 2nd Network Coding Workshop, 2006, <http://www.mit.edu/~medard/papersnew/netcod2006.pdf>
- [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

- [Maach04] Abdelilah Maach, Gregor v. Bochmann, Hussein Mouftah, "Contention avoidance in optical burst switching", ICN'04 - International Conference on Networking, 2004, pp. 1-7
- [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
- [Mandjes02] M. Mandjes, D. Mitra, W. Scheinhardt, "Simple models of network access, with applications to the design of joint rate and admission control", INFOCOM'02, 23-27 June 2002, Vol. 1, pp. 3-12
- [McCulloch43] W. S. McCulloch, W. Pitts, "A Logical Calculus of the Ideas Immanent in Nervous Activity", Bulletin of Mathematical Biophysics, 1943, vol. 5, pp. 115-133
- [McEneaney02] John E. McEneaney, "What's in the brain that ink may character?", National Reading Conference, December 7, 2002
- [Melamed00] Benjamin Melamed, Khosrow Sohraby, Yorai Wardi, "Measurement-Based Hybrid Fluid-Flow Models for Fast Multi-Scale Simulation", DARPA/NMS Project, Sep 2000, <http://204.194.72.101/pub/nms2000sep/UMissouri-KC.pdf>
- [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
- [Naghshineh93] M. Naghshineh, R. Guerin, "Fixed versus variable packet sizes in fast packet-switched networks", INFOCOM'93, March 28 - April 1, 1993, vol. 1, pp. 217-226
- [Nguyen02] Thinh Nguyen, Avidesh Zakhori, "Protocols for distributed video streaming", Image Processing 2002, Vol. 3, Jun 24-28, pp. 185-188
- [Nguyen03] Thinh Nguyen, P. Mehra, Avidesh Zakhori, "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
- [Pakzad05] P. Pakzad, C. Fragouli, A. Shokrollahi, Coding Schemes for Line Networks, ISIT 2005, http://arxiv.org/PS_cache/cs/pdf/0508/0508124.pdf
- [Petrini01] Fabrizio Petrini, Adolfo Hoesie, Wu-chun Fengy, Richard Grahamy, “Performance Evaluation of the Quadrics Interconnection Network”, 15th International Parallel and Distributed Processing Symposium, 23-27 April 2001, pp. 1698-1706
- [Petrini03] Fabrizio Petrini, Adolfo Hoesie, Wu-chun Fengy, Richard Grahamy, Salvador Coll, Eitan Frachtenberg, “Performance Evaluation of the Quadrics Interconnection Network”, Cluster Computing 6, 2003, pp. 125-142
- [Ping06] Yuan Ping, Bai Yu, Wang Hao, “A Multipath Energy-Efficient Routing Protocol for Ad hoc Networks”, International Conference on Communications, Circuits and Systems - ICCAS’06, 25-29 June 2006, pp. 14662-1466 Vol. 3
- [Pitts47] W. Pitts, W. S. McCulloch, “How We Know Universals: The Perception of Auditory and Visual Forms”, Bulletin of Mathematical Biophysics, 1947, vol. 9, pp. 127-147
- [Qiao99] Chunming Qiao, Myungsik Yoo, “Optical burst switching (OBS) - A New Paradigm for an Optical Internet”, Journal of High Speed Networks, 1999, vol. 8, no. 1, pp. 69-84
- [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
- [Quadrics] www.quadrics.com
- [Ramaswami97] R. Ramaswami, G. Sasaki, “Multiwavelength optical networks with limited wavelength conversion”, INFOCOM’97, 7-11 April 1997, vol. 2, pp. 489-498
- [Reinemo06] Sven-Arne Reinemo, Tor Skeie, Thomas Sødning, and Olav Lysne, Ola Tørudbakken, “An overview of QoS capabilities in infiniband, advanced switching interconnect, and ethernet”, IEEE Communications Magazine, July 2006, vol. 44, issue 7, pp. 32-38
- [Rexford96] Jennifer Rexford, Kang G. Shin, “Analytical Modeling of Routing Algorithms in Virtual Cut-Through Networks”, University of Michigan, 1996
- [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
- [Rolland-Balzon02] Philippe Rolland-Balzon, “Color by DSATUR (Brelaz, 1979) a Dimacs graph”, April 2002, <http://prolland.free.fr/works/research/dsatphp/dsat.html>

- [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
- [Shin96] K.G. Shin, S.W. Daniel, "Analysis and implementation of hybrid switching", IEEE Transactions on Computers, June 1996, Vol. 45, Issue 6, pp. 684-692
- [Shokrollahi04] Amin Shokrollahi, "Raptor codes", ISIT'04, June 27 – July 2, page 36
- [Siegrist01] Kyle Siegrist et al, "The Poisson Process", Virtual Laboratories in Probability and Statistics, 2001, http://www.ds.unifi.it/VL/VL_EN/poisson/
- [SIP] SIP Forum, <http://www.sipforum.org/>
- [Sitaram00] Dinkar Sitaram, Asit Dan, "Multimedia Servers", Morgan Kaufmann Publishers, San Francisco California, 2000, pp. 69-73
- [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
- [Steen05] Aad J. van der Steen, Jack J. Dongarra, "Infiniband" from the "Overview of Recent Supercomputers", <http://www.phys.uu.nl/~steen/web05a/infiniband.html>
- [Stern99] Thomas E. Stern, Krishna Bala, "Multiwavelength Optical Networks: A Layered Approach", Addison-Wesley, May 1999
- [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
- [Trick94] Michael Trick, "Network Resources for Coloring a Graph", October 26, 1994, <http://mat.gsia.cmu.edu/COLOR/color.html>
- [Tuninetti05] D. Tuninetti, C. Fragouli, On the Throughput Improvement due to Limited Complexity Processing at Relay Nodes, ISIT 2005, <http://ieeexplore.ieee.org/iel5/10215/32581/01523506.pdf?arnumber=1523506>
- [Turner02] Jonathan Turner, "Terabit Burst Switching Progress Report", Washington University in St. Louis, 14 May 2002
- [Turner99] Jonathan Turner, "Terabit Burst Switching", Journal of High Speed Networks, 1999, vol. 8, no. 1, pp. 3-16
- [Wiki-Poisson06] Poisson distribution, http://en.wikipedia.org/wiki/Poisson_distribution
- [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
- [Yocum97] K.G. Yocum, J.S. Chase, A.J. Gallatin, A.R. Lebeck, "Cut-through delivery in Trapeze: An Exercise in Low-Latency Messaging", 6th International Symposium on High Performance Distributed Computing, 5-8 August 1997, pp. 243-252

Biography

Emin Gabrielyan was born in 2 January 1972 in Yerevan, Armenia.

In 1993 he received his diploma in physics from Yerevan State University (YSU). During his studies, he worked at the Yerevan Physics Institute (YerPhI) on the interconnection of DEC VAX/VMS computer cluster and PC clusters with equipments (CAMAC and FASTBUS at that time) of particle experiments running on the electron accelerator facility. He also participated in installing the satellite link with CERN, thereby establishing the first Internet link in Armenia.

In 1994 he joined Infocom, the national data network carrier operating international X.25 satellite links and providing nationwide X.25, X.28 and X.400 services. He participated in the development and management of a telegraphy message switching center, fax-over-data services and voice message exchange services.

In 1996, he convinced the company of the necessity to provide Internet at the national scale. He organized the installation of international satellite IP links and established an Internet department. He headed the department until 1998. Simultaneously, he founded his own company, with about 20 employees. He started providing Internet services by operating his own international circuits. He carried out negotiations with telecommunication authorities so as to adapt the local regulations according to the ITU recommendations. The VOIP domain was liberated from the telephony monopoly. His company became the first one to provide large scale VOIP terminations and originations in Armenia.

Later, he founded companies in Switzerland and in USA and extended the business to international wholesale telephony. Business in Armenia was soon abandoned, since under the pressure of the monopoly of the phone company, the VOIP domain did not remain free. By establishing parallel redundant communications with different wholesale voice termination markets, high quality telephony services were provided to major phone carriers, relying exclusively on the at that time unreliable packetized VOIP technology.

In 1999 he joined the Peripheral Systems Laboratory at EPFL to start his PhD project. He contributed to the Swiss-Tx project, carried out by the leading Swiss technology institutions and by Compaq Computer Corporation, in cooperation with the Sandia National Laboratory (SNL) and the Oak Ridge National Laboratory (ORNL). The goal of the project was the development of a teraflop supercomputer. Emin Gabrielyan was in charge of the design of a parallel I/O library for the Swiss-Tx cluster supercomputer.

Encountering frequent network congestions occurring during collective I/O transmissions, he started working on optimal scheduling of parallel network transmissions for the Swiss-Tx interconnection network. He then started a research on parallel transmissions for achieving fault-tolerant packetized communications.

From 2005 on, his Swiss company started providing to residential and business customers subscriptions with standalone VOIP phones. The US company now provides long distance and international phone services for permanent line subscribers.

Emin Gabrielyan is married and has 3 children.

Personal Bibliography

Publications related to parallel I/O

- [Gennart99] Benoit A. Gennart, Emin Gabrielyan, Roger D. Hersch, “Parallel File Striping on the Swiss-Tx Architecture”, [EPFL Supercomputing Review 11](#), November 1999, pp. 15-22
- [Gabrielyan00G] Emin Gabrielyan, “SFIO, Parallel File Striping for MPI-I/O”, [EPFL Supercomputing Review 12](#), November 2000, pp. 17-21
- [Gabrielyan01B] Emin Gabrielyan, Roger D. Hersch, “SFIO a striped file I/O library for MPI”, [Large Scale Storage in the Web](#), 18th IEEE Symposium on Mass Storage Systems and Technologies, 17-20 April 2001, pp. 135-144
- [Gabrielyan01C] 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

Papers related to parallel I/O are available at:

<http://switzernet.com/people/emin-gabrielyan/060524-SFIO-papers-workshops/>

Conference papers on liquid scheduling problem

- [Gabrielyan03] Emin Gabrielyan, Roger D. Hersch, “Network Topology Aware Scheduling of Collective Communications”, [ICT’03 - 10th International Conference on Telecommunications](#), Tahiti, French Polynesia, 23 February - 1 March 2003, pp. 1051-1058
- [Gabrielyan04A] Emin Gabrielyan, Roger D. Hersch, “Liquid Schedule Searching Strategies for the Optimization of Collective Network Communications”, [18th International Multi-Conference in Computer Science & Computer Engineering](#), Las Vegas, USA, 21-24 June 2004, CSREA Press, vol. 2, pp. 834-848
- [Gabrielyan04B] Emin Gabrielyan, Roger D. Hersch, “Efficient Liquid Schedule Search Strategies for Collective Communications”, [ICON’04 - 12th IEEE International Conference on Networks](#), Hilton, Singapore, 16-19 November 2004, vol. 2, pp 760-766

Presentations and papers on liquid scheduling problem are available at:

<http://switzernet.com/people/emin-gabrielyan/060729-papers-liquid-sched/>

Papers related to capillary routing

- [Gabrielyan06A] Emin Gabrielyan, “Fault-tolerant multi-path routing for real-time streaming with erasure resilient codes”, ICWN’06 - International Conference on Wireless Networks, Monte Carlo Resort, Las Vegas, Nevada, USA, 26-29 June 2006, pp. 341-346

- [Gabrielyan06B] Emin Gabrielyan, Roger D. Hersch, “Rating of Routing by Redundancy Overall Need”, ITST’06 - 6th International Conference on Telecommunications, June 21-23, 2006, Chengdu, China, pp. 786-789

- [Gabrielyan06C] Emin Gabrielyan, “Fault-Tolerant Streaming with FEC through Capillary Multi-Path Routing”, ICCAS’06 - International Conference on Communications, Circuits and Systems, Guilin, China, 25-28 June 2006, vol. 3, pp. 1497-1501

- [Gabrielyan06D] Emin Gabrielyan, Roger D. Hersch, “Reducing the Requirement in FEC Codes via Capillary Routing”, ICIS-COMSAR’06 - 5th IEEE/ACIS International Conference on Computer and Information Science, 10-12 July 2006, pp. 75-82

Notes, drafts, reports, talks and papers on capillary routing and ROR are available at:
<http://switzernet.com/people/emin-gabrielyan/060129-capillary-documentation/>

Glossary

3G	3rd Generation mobile communication
3GPP	3rd Generation Partnership Project
ADIO	Abstract Device Interface for Portable Parallel I/O
ADSL	Asynchronous Digital Subscriber Line
AMPL	A Mathematical Programming Language
AMR	Adaptive Multi-Rate voice codec 4.75 - 12.2 kbps
ANL	Argonne National Laboratory, http://www.anl.gov/
AODV	Ad-hoc On-demand Distance Vector routing
API	Application Program Interface
ARPANET	Advanced Research Projects Agency Network
ARQ	<u>A</u> utomatic <u>R</u> epet request
ATM	Asynchronous Transfer Mode, a telecommunication protocol
BDT	Bureau de Développement des Télécommunications, currently ITU-D
BER	Bit Error Rate
CAMAC	Computer Automated Measurement And Control, is a modular data handling system used at almost every nuclear physics research laboratory and many industrial sites all over the world
CCITT	Comité Consultatif International Téléphonique et Télégraphique, International Telegraph and Telephone Consultative Committee, based in Geneva, Switzerland, after 1992 is called ITU
CERN	Conseil Européen pour la Recherche Nucléaire, European Organization for Nuclear Research
CODINE / GRD	<u>C</u> omputing in <u>D</u> istributed <u>N</u> etworked <u>E</u> nvironment / <u>G</u> lobal <u>R</u> esource <u>D</u> irector
CPLEX	A high-performance linear programming solver
CPU	Central Processing Unit
CTI	Swiss Commission for Technology and Innovation
DCE	<u>D</u> ata <u>C</u> ommunications <u>E</u> quipment or <u>D</u> ata <u>C</u> ircuit-terminating <u>E</u> quipment, a device that establishes, maintains and terminates a session on a network; it is typically the modem, contrast with DTE
DCL	Digital Command Language, for Open VMS operating system
DEC	Digital Equipment Corporation, purchased by Compaq, which in turn was purchased by Hewlett-Packard
DER	Decoding Error Rate
DMA	Direct Memory Access
DoD	The U.S. Department of Defense
DoS	Deny of Service
DSatur	<u>S</u> aturation <u>D</u> egree, a graph coloring algorithm by D. Brelaz

DSL	Digital Subscriber Line
DTE	Data Terminating Equipment, a communications device that is the source or destination of signals on a network; it is typically a terminal or computer, contrast with DCE
DWDM	Dense Wavelength Division Multiplexing
E/O	Electro/Optical conversion
E/S	Entrées/Sorties
EIGRP	Enhanced Interior Gateway Routing Protocol
EPFL	École Polytechnique Fédérale de Lausanne, Swiss Federal Institute of Technology Lausanne, http://www.epfl.ch/
ETHZ	Eidgenössische Technische Hochschule Zürich, Swiss Federal Institute of Technology Zurich
FASTBUS	IEEE 960 standard for a Modular High-Speed Data Acquisition and Control System
FCI	Fast Communication Interface
FEC	Forward Error Correction
FedEx	Federal Express
FIFO	First In, First Out
flit	<u>Flow unit</u> , in wormhole and cut-through switching
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
GPS	Global Positioning System
gsmfr	High complexity voice codec GSMFR 13200 bps
HCA	Host Channel Adapter, in InfiniBand Architecture
HPC	High Performance Computing
HTTP	HyperText Transfer Protocol
I/O	Input-Output
IBA	InfiniBand Architecture
IBTA	InfiniBand Trade Association
IEEE	Institute of Electrical and Electronics Engineers, http://ieee.org/
IFF	<u>If</u> and only <u>if</u>
ILOG	Developer and distributor of linear programming solutions, http://www.ilog.com
IMP	Interface Message Processor
IOS	Internet Operating System
IP	Internet Protocol
ISO	International Organization for Standardization, http://www.iso.org/
ISP	Internet Service Provider
ITSP	Internet Telephony Service Provider
ITU	International Telecommunication Union, prior to 1992, it was known as CCITT

ITU-D	ITU-D, Telecom Development, is responsible for creating policies, regulation and providing training programs and financial strategies in developing countries. It was created in 1992 from the Bureau de Développement des Télécommunications (BDT)
ITU-T	International Telecommunication Union-Telecommunication Standardization Sector
LAN	Local Area Network
LDPC	Low-Density Parity-Check code
LP	Linear Programming
LSF	Load Sharing Facility, a scheduling system in HPC
LSP	Laboratoire de Systèmes Périphériques, Peripheral Systems Laboratory of EPFL, http://lsp.epfl.ch
LT	Luby Transform Code
MANET	Mobile Ad-hoc Network
MBMS	Multimedia Broadcast/Multicast Service
MDS	Maximum Distance Separable
MEMS	Micro-Electro-Mechanical Systems
MILP	Mixed Integer Linear Programming
MPEG	Moving Picture Experts Group
MPI	Message Passing Interface
MPICH	“CH” in MPICH stands for “Chameleon”, symbol of adaptability to one’s environment and thus of portability
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
NAT	Network Address Translation
NP-complete	Non-deterministic Polynomial time
NYSE	New York Stock Exchange, http://www.nyse.com/
O/E	Optical/Electrical conversion
O/E/O	Optical/Electrical/Optical conversion
OADM	Optical Add/Drop Multiplexer
OBS	Optical Burst Switching
OLT	Optical Line Terminal
ORNL	Oak Ridge National Laboratory, http://www.ornl.gov/
OS	Operating System
OSI	Open System Interconnection Protocols, is an ITU-T standard, comprises numerous standard protocols that are based on the OSI reference model
OSPF	Open Shortest Path First
OXC	Optical Cross-Connect
PAD	Packet Assembler/Disassembler, a communications device that formats outgoing data into packets of the required length for transmission in an X.25 packet switching network; it also strips the data out of incoming packets

PBS	Portable Batch System, a scheduling system in HPC
QoS	Quality of Service
ROR	Redundancy Overall Requirement
RS	Reed-Solomon
RTP	Real-time Transport Protocol
RTT	Round Trip Time
SAN	Storage Area Networks
SCS	Supercomputing Systems
SFIO	Striped File I/O
SIP	Service Initiating Protocol
SNL	Sandia National Laboratories, http://www.sandia.gov/
SONET	<u>S</u> ynchronous <u>O</u> ptical <u>N</u> etwork
Sprint Intl	Sprint International (NYSE:S), a leading US based telecommunication carrier, http://www.sprintlabs.com/
SRI	Stanford Research Institute
TCA	Target Channel Adapter, in InfiniBand Architecture
TCP	Transmission Control Protocol
TDM	Time-Division Multiplexing, a technology in circuit-switched digital telephony
Tele globe	a leading US/Canadian telecommunication carrier acquired by VSNL in 2005, http://www.vsnlinternational.com/
TNET	High-performance switch-based communication network aiming at low-latency and high-bandwidth
UA	User Agent
UCLA	University of California, Los Angeles
UDP	User Datagram Protocol
UNIX	<u>U</u> niplexed <u>I</u> nformation and <u>C</u> omputing <u>S</u> ystem (it was originally spelled “Unics”)
VAX	<u>V</u> irtual <u>A</u> ddress <u>E</u> xtension, a computing architecture that supports an orthogonal instruction set (machine language) and virtual addressing developed by DEC
VCT	Virtual Cut-Through
VMS	Virtual Memory System or Open VMS, is the name of a high-end computer server operating system that runs on the VAX and Alpha family of computers developed by Digital Equipment Corporation, and more recently on Hewlett-Packard systems built around Intel Itanium CPU
VOIP	Voice Over IP
VPN	Virtual Private Network
VSNL	Videsh Sanchar Nigam Limited (NYSE:VSL), India’s leading international telecommunications service provider, http://www.vsnl.in/
WAN	Wide Area Network
WAP	Wavelength Assignment Problem
WDM	Wavelength Division Multiplexing
WIXC	Wavelength-Interchanging Cross-Connect

WSXC	Wavelength-Selective Cross-Connect
X.25	an ITU-T protocol standard for WAN communications that defines how connections between user devices and network devices are established and maintained
X.28	An ITU standard (1977) for exchange of information between a DTE and a PAD; commonly known as PAD commands
X.400	an OSI standard developed by the ITU-T (at the time the CCITT) in cooperation with ISO for the exchange of messages
XOR	<u>E</u> xclusive <u>O</u> R
YerPhI	Yerevan Physics Institute
YSU	Yerevan State University

List of Figures

Figure 1.	Loading the transatlantic cable into the ‘Great Eastern’ in 1865.....	1
Figure 2.	Diagrams from the 51-page report of Paul Baran to the U.S. Air Force, 1964.....	2
Figure 3.	Kidney blood filtering in the human organism.....	3
Figure 4.	Pulmonary circuit of the human organism.....	4
Figure 5.	One of the first Interface Message Processor (IMP) of ARPANET connecting UCLA with SRI in August 1969	5
Figure 6.	Packet switching network: packets are entirely stored at each intermediate switch and only then forwarded to the next switch.....	5
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	6
Figure 8.	Swiss-Tx supercomputer in June 2001	13
Figure 9.	File Striping	14
Figure 10.	SFIO integration into MPI-I/O	16
Figure 11.	Distribution of a striped file across subfiles	18
Figure 12.	Disk access optimization	19
Figure 13.	Comparison of the optimized write access with a non-optimized write access as a function of the file striping granularity (3 I/O nodes, 1 compute node, global file size is 660 Mbytes).....	20
Figure 14.	Comparison of the optimized multi-block write access with corresponding separate non-optimized single block accesses (Fast Ethernet, stripe unit size is 1005 bytes, 7 I/O nodes).....	20
Figure 15.	SFIO functional architecture.....	21
Figure 16.	Aggregate throughput of Fast Ethernet as a function of the number of contributing nodes	24
Figure 17.	SFIO architecture on Swiss-T1.....	24
Figure 18.	SFIO/MPICH all-to-all I/O performance for a 200 byte stripe size, Fast Ethernet	25
Figure 19.	Aggregate throughput of TNET as a function of the number of the contributing nodes.....	25
Figure 20.	The Swiss-T1 network interconnection topology	26
Figure 21.	SFIO all-to-all I/O performance on TNET	27
Figure 22.	The use of derived datatypes in MPI-I/O interface.....	29
Figure 23.	The recursive construction of derived datatypes in MPI (“Contiguous” is a derived datatype obtained by repeatedly joining another datatype which in turn can be fragmented)	29
Figure 24.	The MPI-I/O implementation requires a method for retrieving the fragmentation patterns of opaque MPI derived datatypes.....	30

Figure 25.	A reverse engineering method for discovery the fragmentation pattern of an opaque datatype built by the user	31
Figure 26.	Isolated implementation of a portable MPI-I/O interface functional on any MPI-1 implementation	32
Figure 27.	Wavelength routing in the optical layer	40
Figure 28.	Example of a simple network	42
Figure 29.	The pictograms representing the 25 transfers from all sending nodes to all receiving nodes of the network of Figure 28	43
Figure 30.	Example of a traffic comprising 25 transfers of Figure 29 (over the network of Figure 28) each represented as set of links	45
Figure 31.	An initial category before fission, where symbol Ξ , represents any transfer that is in congestion with x and symbol Θ represents any transfer which is simultaneous with x	48
Figure 32.	Fission of the category of Figure 31 into its positive and negative sub categories.	48
Figure 33.	Fraction of transfers within a skeleton of a traffic, compared with the total number of transfers in the traffic	50
Figure 34.	Search space reduction obtained by idle+skeleton+blank optimization steps	52
Figure 35.	Time frames of a liquid schedule of the collective traffic shown in Figure 30	53
Figure 36.	A traffic of three transmissions (shown in Figure 37) across this network has no team and therefore no liquid schedule	54
Figure 37.	A traffic consisting of three transmissions to be carried across the network shown in Figure 36	54
Figure 38.	Liquid schedule construction tree: $X_{i_1 i_2 \dots i_n}$ denotes a reduced subtraffic at the layer $n+1$ of the tree and $A_{i_1 i_2 \dots i_n i_{n+1}}$ denotes a candidate for the time frame $n+1$; the operator \aleph applied to a subtraffic X_{sub} yields the set of all possible candidates for a time frame	55
Figure 39.	Architecture of the Swiss-T1 cluster supercomputer interconnected by a high performance wormhole switch fabric	58
Figure 40.	For a given number of contributing nodes, all possible allocations of nodes yielding different liquid throughputs	60
Figure 41.	The 362 topologies of Figure 40 yielding different liquid throughput values placed along one axis, sorted first by the number of contributing nodes and then by their liquid throughputs	60
Figure 42.	Theoretical liquid throughput and measured round-robin schedule throughput for 362 network sub topologies	61
Figure 43.	Predicted liquid throughput and measured throughput according to the computed liquid schedule	62
Figure 44.	In the first layer the flow is equally split across two paths. Two of their links, marked by thick dashes, are the bottlenecks	68
Figure 45.	The second layer minimizes to 1/3 the maximal load of the remaining seven links and identifies three bottlenecks.	68

Figure 46.	The third layer minimizes to 1/4 the maximal load of the remaining four links and identifies two bottlenecks.	68
Figure 47.	Routing pattern of layer 10 built by the capillary routing algorithm on a network sample with 180 nodes.....	68
Figure 48.	Initial problem with one source and one sink node	69
Figure 49.	Maximize the flow, fix the new flow-out coefficients at the nodes and find the bottleneck links (layer 1, $F^1 = 2$).....	69
Figure 50.	Remove the bottleneck links from the network and adjust the flow-out coefficients at the adjacent nodes	69
Figure 51.	Maximize the flow in the new sub-problem, fix the new flow-out coefficients at the nodes and find the new bottlenecks (layer 2, $F^2 = 1.5$).....	70
Figure 52.	Again remove the bottleneck links from the network and adjust correspondingly the flow-out coefficients at the adjacent nodes.....	70
Figure 53.	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, $F^3 = 4/3$).....	70
Figure 54.	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).....	71
Figure 55.	A max-flow solution with the flow increase factor of 4/3, containing four maximally loaded candidate links $\{a, b, d, e\}$	71
Figure 56.	The cost reduction applied to the four fully loaded links of Figure 55 reduces the load of suspected link d , and the bottleneck candidate list is now $\{a, b, e\}$	72
Figure 57.	The cost reduction applied to the three fully loaded links of Figure 56 reduces the load of another suspected link a . The true bottleneck links are $\{b, e\}$	72
Figure 58.	Decrease in the number of suspected links during the bottleneck hunting loop at each of the 10 capillary routing layers.....	72
Figure 59.	Transmission rate increase factor as a function of the packet loss rate ($DER = 10^{-5}$)	76
Figure 60.	Average ROR metric as a function of the capillary routing layer	78
Figure 61.	Average ROR metric computed assuming real-time streaming (the group of curves above) and off-line streaming (the group below)	78
Figure 62.	Congestion graph corresponding to the traffic pattern of Figure 29 across the network of Figure 28; the vertices of the graph represent the 25 transfers; the edges represent congestions between the transfers	87
Figure 63.	Number of edges in the 362 congestion graphs corresponding to the traffic patterns of Figure 40 and Figure 41	88
Figure 64.	Loss in throughput induced by schedules computed with the DSatur heuristic algorithm.....	89
Figure 65.	Running times for computing liquid schedules with the MILP Cplex method and with the liquid schedule construction algorithm.....	92
Figure 66.	The overall measured throughputs of hundreds of different traffic patterns carried out according to both a liquid schedule and a topology unaware schedule	100

Figure 67. The probability that the interarrival time between two consecutive failures in a Poisson process is less than a given time, $r = 1/3600$, $N = 50$ 102

List of Tables

Table 1.	Optimized algorithm for retrieving all full teams of a traffic	51
Table 2.	The routing table of the Swiss-Tx supercomputer shown in Figure 39	59
Table 3.	DSatur graph coloring heuristic algorithm	89
Table 4.	Overall overview of liquid schedule construction algorithm and its all relevant optimizations	99

Links

- [060921] The document for the private defense, <http://switzernet.com/people/emin-gabrielyan/060921-thesis-for-experts>, <http://4z.com/people/emin-gabrielyan/public/060921-thesis-for-experts>
- [051003] Capillary routing releases, source codes and downloads, <http://switzernet.com/people/emin-gabrielyan/051003-capillary-releases/>, <http://4z.com/people/emin-gabrielyan/public/051003-capillary-releases/>
- [060509] Least FEC routing, multi-path patterns for small scale networks, <http://switzernet.com/people/emin-gabrielyan/060509-least-FEC-routing/>, <http://4z.com/people/emin-gabrielyan/public/060509-least-FEC-routing/>
- [060724] Energy saving in a wireless network using network coding, <http://switzernet.com/people/emin-gabrielyan/060724-netcod-flooding/>, <http://4z.com/people/emin-gabrielyan/public/060724-netcod-flooding/>