Leveraging Modern Interconnects for Parallel System Software

Thesis submitted for the degree of

“Doctor of Philosophy”

by

Eitan Frachtenberg

Submitted to the Senate of the Hebrew University

December 2003

This work was carried out under the supervision of
Dr. Dror Feitelson

Abstract

The use of clusters of independent compute nodes as high capability and capacity computers is rapidly growing in industry, academia, and government. This growth is accompanied by fast-paced progress in cluster-aware hardware, and in particular in interconnection technology. Contemporary networks offer not only excellent performance as expressed by latency and bandwidth, but also advanced architectural features, such as programmable network interface cards, hardware support for collective communication operations, and support for modern communication protocols such as MPI and RDMA.

The rapid progress in cluster hardware and usage is unfortunately not matched by similar progress in system software. This software consists of the middleware: the operating system, user libraries, and utilities that interface between the hardware and the user applications, allowing them to make use of the machine's resources. In fact, most of these clusters use common workstation operating systems such as Linux running on each of the cluster's nodes, with a collection of loosely-related libraries, utilities, and scripts to access the cluster's resources. Such solutions are hardly adequate for large-scale clusters and/or high-performance computing applications. The problems they cause include (but are not limited to): (1) poor performance and scalability of applications and system software; (2) reduced utilization of the machine due to suboptimal resource allocation; (3) reliability problems caused by the multitude of independent software modules, and the redundancy in their operation, and (4) difficulty in operating and making full use of these machines.

The premise behind this dissertation is that system software can be dramatically improved in terms of performance, scalability, reliability, and simplicity by making use of the features offered by modern inter-

connects. Unlike single-node operating systems, most of a cluster's system software tasks involve efficient global synchronization of resources. As such, parallel system software can be designed to benefit from the novel hardware features offered by contemporary interconnection technology. This dissertation promotes the idea of treating a cluster's operating system as any other high-performance parallel application, and increasing its reliance on synchronization abilities while reducing its per-node complexity and redundancy.

This dissertation makes the following primary contributions. First, a set of necessary network mechanisms to support this system software model is described. A prototype implementation of system software based on these mechanisms is then discussed. This system currently tackles three main aspects of parallel computers: resource management, communication libraries, and job scheduling methods. This model was implemented on three different cluster architectures. Extensive performance and scalability evaluations with real clusters and applications show significant improvements over previous work in all three areas. In particular, this research focuses primarily on job scheduling strategies, and demonstrates that through advanced algorithms, the system's throughput and responsiveness can be improved over a wide spectrum of workloads.

Contents

List of Figures

List of Tables

List of Publications

1. Jose Carlos Sancho, Fabrizio Petrini, Greg Johnson, Juan Fernandez, Eitan Frachtenberg. On the Feasibility of Incremental Checkpointing for Scientific Computing. To appear in *Proceedings of the International Parallel and Distributed Processing Symposium* IPDPS'04, Santa Fe, NM, April 2004

2. Juan Fernandez, Eitan Frachtenberg, Fabrizio Petrini. **BCS MPI: A New Approach in the System Software Design for Large-Scale Parallel Computers**. In *Proceedings of the IEEE/ACM Conference on Supercomputing SC'03*, Phoenix, AZ, November 2003.

3. Fabrizio Petrini, Juan Fernandez, Eitan Frachtenberg, Salvador Coll. **Scalable Collective Communication on the ASCI Q Machine**. In *Proceedings of the 11th Hot Interconnects conference HOTi11*, Stanford University, Palo Alto, CA, August 2003.

4. Eitan Frachtenberg, Dror G. Feitelson, Juan Fernandez, Fabrizio Petrini. **Parallel Job Scheduling under Dynamic Workloads**. In *Proceedings of the 9th Workshop on Job Scheduling Strategies for Parallel Processing JSSPP'03*, In Conjunction with HPDC12 / GGF8, Seattle, WA, June 2003.

5. Fabrizio Petrini, Eitan Frachtenberg, Adolfy Hoisie, Salvador Coll. **Performance Evaluation of the Quadrics Interconnection Network**. In *Journal of Cluster Computing*, 6(2): 125–142, April 2003.

6. Salvador Coll, Eitan Frachtenberg, Fabrizio Petrini, Adolfy Hoisie, Leonid Gurvits. **Using Multi-rail Networks in High-Performance Clusters**. In *Concurrency and Computation: Practice and Experience*, 15(7-8): 625–651, April 2003.

7. Eitan Frachtenberg, Dror Feitelson, Fabrizio Petrini, Juan Fernandez. **Flexible CoScheduling: Dealing with Load Imbalance and Heterogeneous Resources**. In *Proceedings of the International Parallel and Distributed Processing Symposium* IPDPS'03, Nice, France, April 2003. Best Paper Award

8. Eitan Frachtenberg, Fabrizio Petrini, Juan Fernandez, Scott Pakin, Salvador Coll. **STORM: Lightning-Fast Resource Management**. In *Proceedings of the IEEE/ACM Conference on Supercomputing SC'02*, Baltimore, MD, November 2002.

9. Eitan Frachtenberg, Fabrizio Petrini, Juan Fernandez, Salvador Coll. **Scalable Resource Management in High-Performance Computers**. In *Proceedings of the IEEE International Conference on Cluster Computing CLUSTER'02*, Chicago, IL, September 2002.

10. Fabrizio Petrini, Wu-chun Feng, Adolfy Hoisie, Salvador Coll, Eitan Frachtenberg. **The Quadrics Network (QsNet): High-Performance Clustering Technology**. In *IEEE Micro*, 22(1): 46–57, February 2002.

11. Salvador Coll, Fabrizio Petrini, Eitan Frachtenberg, Adolfy Hoisie. **Performance Evaluation of I/O Traffic and Placement of I/O Nodes on a High Performance Network**. In *Proceedings of the Workshop on Communication Architecture for Clusters CAC'02*, In conjunction with the International Parallel and Distributed Processing Symposium IPDPS'02, Fort Lauderdale, FL, April 2002.

12. Fabrizio Petrini, Salvador Coll, Eitan Frachtenberg, Adolfy Hoisie. **Hardware- and Software-Based Collective Communication on the Quadrics Network**. In *Proceedings of the 1st IEEE International Symposium on Network Computing and Applications NCA'01*, Boston, MA, October 2001.

13. Salvador Coll, Eitan Frachtenberg, Fabrizio Petrini, Adolfy Hoisie, Leonid Gurvits. **Using Multirail Networks in High-Performance Clusters**. In *Proceedings of the IEEE International Conference on Cluster Computing CLUSTER'01*, Newport Beach, CA, October 2001.

14. Eitan Frachtenberg, Fabrizio Petrini, Salvador Coll, Wu-chun Feng. **Gang Scheduling with Lightweight User-Level Communication**. In *Proceedings of the Workshop on Scheduling and Resource Management for Cluster Computing SRMCC'01*, In conjunction with the *International Conference on Parallel Processing ICPP'01*, Valencia, Spain, September 2001.

15. Fabrizio Petrini, Wu-chun Feng, Adolfy Hoisie, Salvador Coll, Eitan Frachtenberg. **The Quadrics Network (QsNet): High-Performance Clustering Technology**. In *Proceedings of the 9th Hot Interconnects conference HOTi9*, Stanford University, Palo Alto, CA, August 2001.

16. Eitan Frachtenberg, Fabrizio Petrini, Juan Fernandez, Scott Pakin. **STORM: Scalable Resource Management for Large-Scale Parallel Computers**. Submitted to *ACM Transactions on Computer Systems*.

17. Juan Fernandez, Eitan Frachtenberg, Fabrizio Petrini, Jose Carlos Sancho. **An Abstract Interface for System Software on Large-Scale Clusters**. Submitted to *Journal of Parallel Computing*

1 Introduction

1.1 Problem Description

Large-scale clusters and loosely-coupled high-performance computing (HPC) systems are growing both in size and prevalence. These machines are used for a variety of complex computations and simulations, and many scientific, governmental, financial and commercial institutes are turning to clusters as a solution for their demanding computational needs. The hardware capabilities of these systems, as expressed by their processor and network performance, number of processors, etc., grows at an exponential rate [122].

Nevertheless, some of the main metrics used for evaluating HPC systems, such as sustained performance and utilization, show a comparatively slower growth. Furthermore, clusters become increasingly harder to manage than sequential systems. This growing gap stems partly from hardware constraints, such as the increasing imbalance between different components' performance. Still, a significant factor of this gap is the result of software deficiencies, and in particular, the system software, which comprises of all the software other than the user programsq. These deficiencies include uneven usage of resource (load-imbalance) or suboptimal usage of system resources such as the network, processors, I/O, etc., which leads to wasted idle time. While many applications can sometimes be optimized and tuned for specific architectures or load-balancing scenarios, often these optimizations are limited and/or prohibitively costly (see for example [61]). Moreover, since many workloads are quite complex, no single application can optimize the system performance as a whole.

Many of the large scale clusters use commodity system software, such as Linux and MPI [60]. Such software, although sometimes optimized for a given hardware platform or application, is typically too general to address the needs of large scale machines and applications. Many of these software packages were designed with a single node or small clusters in mind, and scale poorly to thousands of nodes. Many of the systems needs, such as efficient resource management and allocation, job scheduling, parallel file system, transparent fault tolerance, high performance communication libraries, and monitoring services are not adequately addressed by a collection of commodity software modules. Furthermore, the combination of separate, often ad-hoc, system programs to address these needs often involves inefficiency and redundancy, such as when different modules implement their own communication protocols to meet the same

requirements. Noise created by independent, node-based system dæmons that are not coscheduled to run together, often creates a domino effect that severely affects applications' performance [88]. The quality of the system software not only affects application performance, but also the cost of ownerships of these machines.

1.2 Scope of Work

Since system software for large-scale machines consists of many complex tasks, we set out to focus on some of the more important aspects of system software. In this work, we explore several aspects of system software, and how they can be viewed and implemented as a parallel application in their own sake. The main research emphasis of this work is in the field of job scheduling, and the efficient execution of multiple jobs under many, sometimes conflicting, constraints. Some related subjects are also discussed, such as resource management (the accounting and allocation tasks of resources in a large parallel machine), dealing with load-imbalance, workload analysis, user-level communication libraries, and fault tolerance. Much of the previous work in system software research for parallel and distributed systems focused on particular aspects of the problem domain. There is a wide body of work on the specific issues listed above, but to the best of our knowledge, no previous work has tried to generalize all aspects of parallel system software in a single, simplified view emerging from the capabilities offered by advanced interconnects. In the following chapters, we will survey significant related work that pertains to each respective subject.

This work offers several contributions to the areas of parallel system software. Most of the in-depth contributions are in the field of parallel job scheduling, and an improvement of system utilization and responsiveness through job scheduling techniques. We demonstrate the feasibility and benefits of novel job scheduling strategies for running realistic, complex workloads on large-scale machines. More generally, this thesis promotes the notion of a global, cohesive parallel operating system that is adequate for loosely-coupled large scale systems such as commodity clusters. It is based on the idea that system software is essentially a parallel, synchronized, high-performance application in itself, and thus should benefit from the advances in modern interconnect technology. We identify an abstract layer incorporating a set of advanced and yet portable network primitives, on which an entire foundation for system software can be erected. Using extensive modeling and performance evaluation, we also demonstrate the feasibility of relying on this layer to develop a system infrastructure, where various important tasks are simplified, converged, and yet offer unprecedented performance and scalability. Through implementation of experimental system software on several hardware platforms, the arguments laid above are translated into an actual proof of concept.

The machines discussed in this work include clusters of workstations and SMPs, constellations, and other loosely-coupled configurations characterized by relatively independent compute nodes connected by

an interconnection network[1]. Tightly-coupled systems, characterized by compute nodes that do not run independent OSs, are for the most part not covered in the scope of this work. By means of a unified, high-performance system software, we hope to bring the management and performance model of tightly-coupled systems closer to clusters.

1.3 Research Goals

The problems described in Section 1.1 above are made more challenging by the constant growth of clusters. One of the main goals in this work was to try to answer the following question: What hardware features, and thus which abstract interface, should the interconnection network provide to the system software designers? Once this layer is defined, we set out to use it in a system-centric approach that strives to solve application problems at operating-system level, so that the preexisting investment in application development can be kept. The vision behind this work of that of a global, cohesive OS, that like any other HPC application can leverage advanced interconnect technology, to offer an order of magnitude improvement in performance and scalability over the current predominant solutions, namely, the use of a collection of loosely-coupled single node operating systems.

To this end, we set out to investigate the following questions:

1. What are the properties of modern interconnects that are conducive to our vision?
2. How can a system infrastructure be based on a small set of network mechanisms?
3. Can we build upon such an infrastructure to address the various requirements of a global, parallel operating system, and in particular, for advanced job scheduling?

For the first question, we used the QsNet network as a case study, due to its high performance, advanced design, extensive programmability, and relative prevalence in high-end machines [122]. These features include not only raw performance but also hardware support for communication collectives and programmable NICs. Such novel features are also being introduced in other state of the art interconnects such as Myrinet, BlueGene's network and Infiniband, and serve an important role in the scheduling methods we propose.

To address the second question, we set out to build a complete resource management system prototype, based on a small set of network primitives. Many extensive performance and scalability tests were performed on this platform, and are described in Chapter 4.

Finally, the bulk of this research was dedicated to exploring various aspects of system software, and in particular those of job scheduling. To this end, we implemented several job scheduling algorithms, including various forms of batch scheduling, gang scheduling (GS), buffered coscheduling (BCS), flexible coscheduling (FCS), and implicit coscheduling (ICS).

[1] We refer collectively to these machines as *clusters*, for brevity.

Since an important difference between a parallel and a serial or distributed application is its communication requirements, we believe our operating system should be able to monitor and control the communication behavior of applications and even of its own management tasks. In our implementations of FCS and BCS, the system monitors the communication behavior and requirements of applications and uses this information to make informed scheduling decisions for improving resource utilization and load-balancing. Since global coordination can be beneficial for a cluster operating system and is required for FCS and BCS, the system should be able to support this need scalably and efficiently. HPC applications are often also more sensitive than serial application to the balance between architectural components such as CPU speed, memory bandwidth, I/O, etc. The subject of improved in-node utilization is also addressed in this thesis. The field of parallel job scheduling is still in its growth, and there are many factors and interactions that are still not well understood. One of the goals of this thesis is to expand on experimental methods in this field, and shed some light on some of the associated intricacies.

Realizing a full blown operating system at kernel level is a tedious task, and unnecessary to show the feasibility of our abstract layer and its usage. Instead, we chose to focus on user-level code with some additional code running on the NIC thread processor, more like a regular parallel application. This approach possibly comes at some performance loss, but offers significant gains in simplicity and portability.

The big picture

In a broad view, this thesis is part of a larger vision for cluster middleware. Typical clusters today use commodity operating systems that are optimized for a single node (e.g., Linux, AIX, Windows NT, etc). We believe that a cluster operating system should be regarded as a parallel application in itself, with its own communication infrastructure. As such, it should assume the traditional roles of an operating system — scheduling, resource management, security, etc. — with a cluster-wide view. In this work we focus mainly on the job scheduling aspects of such an OS, but we believe that the principles that are developed and explored, namely the use of advanced interconnects for global OS coordination, can be generalized to most of the other roles.

This work set out with the goal of advancing our knowledge and suggesting innovative answers to several research questions:

- Better understanding of modern interconnect features
- How such features can be used for scalable resource management
- The study of state-of-the-art job scheduling algorithms in a realistic hardware and software environment
- The development of useful benchmarks for job scheduling algorithms

- How job scheduling algorithms can be improved and extended by using advanced network features, and by incorporating active information about the applications' communication characteristics
- Advancing the state-of-the art in job scheduling in the areas of load balancing, resource utilization and achievable throughput
- Opening the door for novel applications of global resource coordination

1.4 Methodology

This study involves many different factors of high complexity, such as operating systems, hardware architectures, network protocols, parallel applications and dynamic workloads. Many of these factors and their interactions are not yet completely understood, making a theoretical model analysis largely impractical and unrealistic. The most common approach for research in this field is to implement detailed simulations of systems and workloads. However, we believe that such simulations cannot encapsulate all the complexities involved, and thus often make simplifying assumptions that fail to consider all the factors. Instead, the results presented in this work are based on experimental data garnered from actual parallel programs running on various cluster architectures. We tested our new concepts by developing a prototype platform for system software research, allowing the study of aspects ranging from user-level communication libraries to resource management and job scheduling. To this end, most of this study was conducted at the CCS-3 group of Los Alamos National Laboratory (LANL), where the facilities offer several state-of-the art hardware platforms. This study was mostly carried on three cluster architectures:

1. *Wolverine*: An Alpha EV6-processor based cluster with 256 PEs (4 per node) connected by two independent QsNet networks (rails). This cluster was initially ranked 83rd on the top500 list.
2. *Accelerando*: An Itanium-2 (McKinley) based cluster with 64 PEs (2 per node) connected by two QsNet rails.
3. *Crescendo*: A Pentium-III based cluster with 64 PEs (2 per node) connected by one QsNet network. This platform was dedicated to our experiments and was thus used to obtain most of our results.

Table 1.1 summarizes the main features of these clusters. We were also able to obtain a few performance numbers on the ASCI-Q cluster, ranking no. 2 at the top 500 list [122]. Chapter 2 describes in detail the Quadrics network architecture, that is shared by all these machines.

We developed a comprehensive software platform to study job scheduling and resource management algorithms called STORM, described in Chapter 4. STORM allows running arbitrary MPI programs on the cluster while making measurements on their communication patterns and performance. For this study, we mostly made use of two LANL applications, SAGE and SWEEP3D, as well as several applications from the NAS benchmark [4, 123].

Component	Feature	Crescendo	Accelerando	Wolverine
Node	Nodes×PEs	32×2	32×2	64×4
	Memory	$1GB$	$2GB$	$2GB$
	I/O buses	2×PCI	2×PCI-X	2×PCI
	Model	Dell 1550	HP Server rx2600	AlphaServer ES40
	OS	RH Linux 7.3	RH Linux 7.2	RH Linux 7.1
CPU	Type	Pentium-III	Itanium-II	Alpha EV68
	Speed	$1GHz$	$1GHz$	$833MHz$
I/O bus	Type	64bit×$66MHz$	64bit×$133MHz$	64bit×$33MHz$
Network	NICs	1×Elan3	2×Elan3	2×Elan3
Compiler	maker/version	Intel v.5.0.1	Intel v.7.1.17	Compaq

Table 1.1: Main experimental platforms

SWEEP3D [53] is a time-independent, Cartesian-grid, single-group, discrete ordinates, deterministic, particle transport code taken from the ASCI workload. SWEEP3D represents the core of a widely used method of solving the Boltzmann transport equation. Estimates are that deterministic particle transport accounts for $50 - 80\%$ of the execution time of many realistic simulations on current DOE systems. SWEEP3D is characterized by a fine computational granularity and a nearest-neighbor communication stencil.

SAGE (SAIC's Adaptive Grid Eulerian hydrocode) is a multidimensional (1D, 2D, and 3D), multimaterial, Eulerian hydrodynamics code with adaptive mesh refinement (AMR) [61]. The code uses 2nd order accurate numerical techniques. SAGE comes from the LANL Crestone project, whose goal is the investigation of continuous adaptive Eulerian techniques to stockpile stewardship problems. SAGE has also been applied to a variety of problems in many areas of science and engineering including water shock, stemming and containment, early time front design, and hydrodynamics instability problems.

To simplify some of the modeling and evaluation, we also use a synthetic application we developed to produce a wide spectrum of workloads and job mixes. Like SAGE and SWEEP3D, it assumes the bulk-synchronous parallel model (BSP), where a parallel applications basically consists of a loop or several loops involving a computation phase followed by a communication or synchronization phase. Our synthetic application uses MPI calls and CPU-intensive "computation" code. It allows the precise controlling of many parameters, such as the computation granularity, the communication pattern, and the amount of variability (heterogeneity and load-imbalance) to induce on different processes. The main difference between the synthetic application and real applications is that it exerts almost no cache or memory pressure, since it performs no useful computation.

1.5 Organization

The next chapters follow a bottom-up structure, and correspond to a large extent to the work done in chronological order. We start by reviewing the state of the art in modern interconnects, and in particular

we focus on the Quadrics QsNet network. This network became the primary platform for our research and was studied extensively. The following chapter describes in detail the various architectural properties and performance features that make novel networks such as QsNet suitable for system software advances.

The next natural step is the identification and analysis of a minimal set of network mechanisms that can support the needs of a global system software. To this end, we study various system needs in Chapter 3 and derive the required mechanisms. Based on these mechanisms, we developed a research prototype of an advanced resource management system, as the first step toward global, scalable system software. This system, called STORM (Scalable TOol for Resource Management) is described and studied extensively in Chapter 4. Other than advancing the state of the art in various aspects of resource management, it also provides a flexible infrastructure for evaluating different parallel job scheduling algorithms on a real cluster implementation.

Another important aspect of system software, the user-level communication library, is discussed in Chapter 5. We have implemented our own version of the MPI library based on the BCS model. While similar in performance to the Quadrics production-level MPI, it offers a much simpler design, and the potential for significant benefits that arise from the globally-deterministic nature of BCS.

The focal point of this thesis is the improvement of system performance using job scheduling techniques. Several previously-studied scheduling algorithms were implemented, as well as two new algorithm implementations, FCS and BCS. We compare these algorithms under various scenarios in Chapter 6. In particular, we were interested in the properties of different scheduling policies when running long, dynamic workloads with communicating programs, that are more representative of real world usage of HPC computing centers. This is a relatively new, complex area of study with a large number of parameters and feedback effects that are still not completely understood. Our work on parallel job scheduling sheds some new light on a few of these issues, and contains the bulk of the innovation in this dissertation.

One potential benefit from our deterministic communication approach is transparent fault tolerance, using automatic incremental checkpointing that remains consistent along the entire cluster. We intend to pursue this venue in future research, and discuss it briefly along with other future directions in Chapter 7. Finally, the last chapter offers a summary of the main contributions of this work.

2 Modern Interconnects

2.1 Overview

With the increased importance of scalable system-area networks for cluster computers, web-server farms, and network-attached storage, the interconnection network and its associated software libraries and hardware have become critical components in achieving high performance. Such components will greatly impact the design, architecture, and use of the aforementioned systems in the future.

Key players in high-speed interconnects include Gigabit Ethernet (GigE) [99], GigaNet [117], SCI [50], Myrinet [69], Quadrics' QsNet [87] and GSN (HiPPI-6400) [114]. These interconnects differ from one another with respect to their architecture, programmability, scalability, performance, and ease of integration into large-scale systems. While GigE resides at the low end of the performance spectrum, it provides a low-cost solution. GigaNet, GSN, QsNet, Myrinet, Infiniband and SCI add programmability and performance by providing communication processors on the network interface cards and implementing various types of user-level communication protocols. Infiniband [3] has been recently proposed as a standard for communication between processing nodes and I/O devices as well as for interprocessor communication, offering an integrated view of computing, networking and storage technologies. The Infiniband architecture is based on a switch interconnect technology with high speed point-to-point links and offers support for Quality of Service (QoS), fault-tolerance, remote direct memory access, etc.

In this dissertation we focus on the Quadrics network (QsNet) as a case study for advanced interconnects. QsNet provides a number of innovative design features, some of which are very similar to those defined by the Infiniband specification. Some of these salient aspects are the presence of a programmable processor in the network interface that allows the implementation of intelligent communication protocols, fault-tolerance, and RDMA. In addition, QsNet integrates the local virtual memory into a distributed virtual shared memory. At the time of writing, QsNet is used as the interconnect of 6 of the 10 most powerful supercomputers worldwide [122].

This chapter provides an overview of QsNet's advanced architecture and features, and in particular those that can be used for advancing the state of the art in parallel system software and job scheduling.

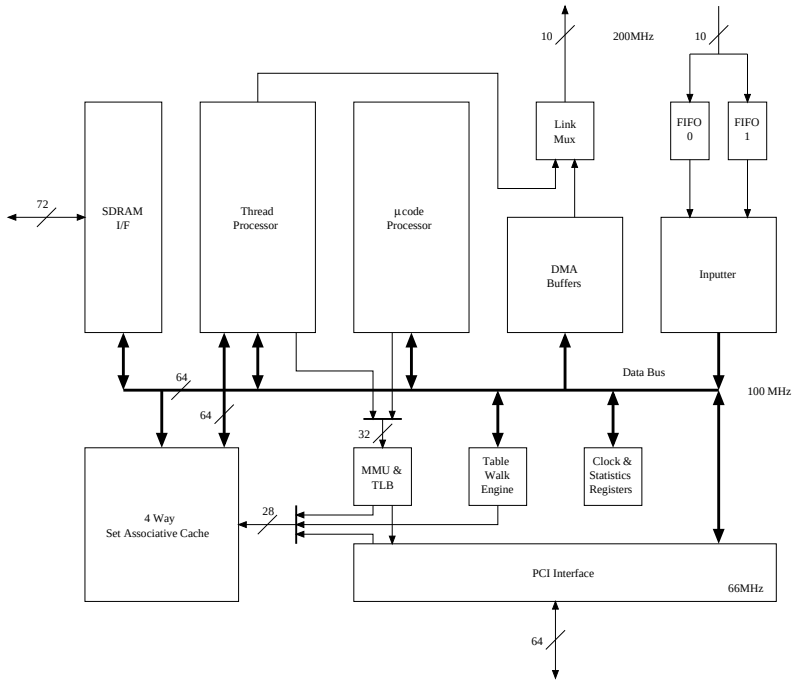


Figure 2.1: Elan Functional Units

2.2 Hardware Features

QsNet is based on two building blocks, a programmable network interface called Elan [95] and a low-latency high-bandwidth communication switch called Elite [96]. Elites can be interconnected in a fat-tree topology [67]. The network has several layers of communication libraries that provide trade-offs between performance and ease of use. Other important features are hardware support for collective communication and fault-tolerance.

2.2.1 Elan

The Elan[1] network interface links the high-performance, multi-stage Quadrics network to a processing node containing one or more CPUs. In addition to generating and accepting packets to and from the network, the Elan NIC also provides substantial local processing power to implement high-level message-passing protocols such as MPI. The internal functional structure of the Elan, shown in Figure 2.1, centers around two primary processing engines: the microcode processor and the thread processor.

The 32-bit microcode processor supports four separate threads of execution, where each thread can independently issue pipelined memory requests to the memory system. Up to eight requests can be outstanding at any given time. The scheduling for the microcode processor is lightweight, enabling a thread to wake up, schedule a new memory access on the result of a previous memory access, and then go back to

[1]This dissertation refers to the Elan3 version of the NIC. Elan and Elan3 will be therefore used interchangeably.

sleep in as few as two system-clock cycles.

The four microcode threads are described below:

1. *inputter thread:* Handles input transactions from the network.
2. *DMA thread:* Generates DMA packets to be written to the network, prioritizes outstanding DMAs, and time-slices large DMAs so that small DMAs are not adversely blocked.
3. *processor-scheduling thread:* Prioritizes and controls the scheduling and descheduling of the thread processor.
4. *command-processor thread:* Handles operations requested by the host processor at user level.

The thread processor is a 32-bit RISC processor used to aid the implementation of higher-level messaging libraries without explicit intervention from the main CPU. In order to better support such an implementation, the thread processor's instruction set was augmented with extra instructions that construct network packets, manipulate events, efficiently schedule threads, and block-save and restore a thread's state when scheduling.

The Elan contains routing tables that translate every virtual process number into a sequence of tags that determine the network route. Several routing tables can be loaded in order to have different routing strategies. The link logic transmits and receives data from the network and outputs 9 bits and a clock signal on each half of the clock cycle. The flit (flow-control digit) encoding scheme allows data and command tokens to be interleaved on the link and prevents a corrupted data word being interpreted as a token or a token being interpreted as another token. Each link provides buffer space for two virtual channels with a 128-entry, 16-bit FIFO RAM for flow control.

2.2.2 Elite

The other building block of the QsNet is the Elite switch. The Elite provides the following features: (1) 8 bidirectional links supporting two virtual channels in each direction, (2) an internal 16×8 full crossbar switch[2], (3) a nominal transmission bandwidth of $400MB/s$ on each link direction and a flow through latency of $35ns$, (4) packet error detection and recovery, with routing and data transactions CRC protected, (5) two priority levels combined with an aging mechanism to ensure a fair delivery of packets in the same priority level, (6) hardware support for broadcasts, (7) and adaptive routing.

The Elite switches are interconnected in a quaternary fat-tree topology, which belongs to the more general class of the k-ary n-trees [89, 90]. A quaternary fat-tree of dimension n is composed of 4^n processing nodes and $n * 4^{n-1}$ switches interconnected as a delta network, and can be recursively built by connecting 4 quaternary fat trees of dimension $n - 1$.

[2]The crossbar has two input ports for each input link, to accommodate the two virtual channels.

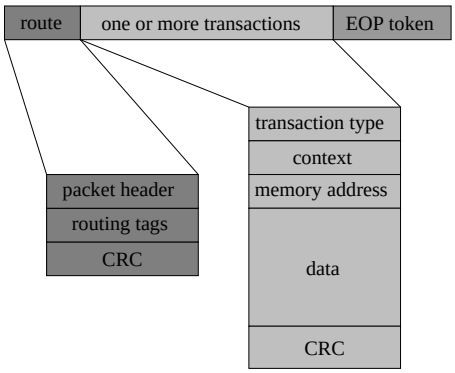


Figure 2.2: Packet Transaction Format

2.2.2.1 Packet Routing and Flow Control

Each user- and system-level message is chunked in a sequence of packets by the Elan. An Elan packet contains three main components. The packet starts with the (1) routing information, that determines how the packet will reach the destination. This information is followed by (2) one or more transactions consisting of some header information, a remote memory address, the context identifier and a chunk of data, which can be up to 64 bytes in the current implementation. The packet is terminated by (3) an end of packet (EOP) token, as shown in Figure 2.2.

Transactions fall into two categories: write block, and non-write block transactions. The purpose of the former is to write a block of data from the source node to the destination node, using the destination address contained in the transaction immediately before the data. A DMA operation is implemented as a sequence of write block transactions, partitioned into one or more packets (a packet normally contains 5 write block transactions of 64 bytes each, for a total of 320 bytes of data payload per packet).

The non-write block transactions implement a family of relatively low level communication and synchronization primitives. For example, non-write block transactions can atomically perform remote test-and-write or fetch-and-add and return the result of the remote operation to the source, and can be used as building blocks for more sophisticated distributed algorithms.

Elite networks are source-routed. The routing information is attached to the header before injecting the packet into the network and is composed by a sequence of Elite link tags. As the packet moves inside the network, each Elite removes the first routing tag from the header, and forwards the packet to the next Elite in the route or to the final destination. The routing tag can identify either a single output link or a group of adjacent links for adaptive routing purposes.

The transmission of each packet is pipelined into the network using wormhole flow control. At link level, each packet is partitioned in smaller units called flits [21] of 16 bits. The header flit opens a circuit between source and destination, and this path stays in place until the destination sends an acknowledgment to the source. At this point, the circuit is closed by sending and End Of Packet (EOP) token. It is worth

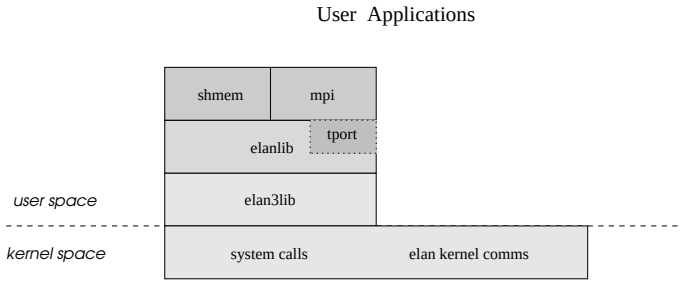


Figure 2.3: Elan3 programming library hierarchy

noting that both acknowledgment and EOP can be tagged to communicate control information. So, for example, the destination can notify the successful completion of a remote non-write block transaction without explicitly sending an extra packet.

Minimal routing between any pair nodes can be accomplished by sending the message to one of the nearest common ancestors and from there to the destination. That is, each packet experiences two routing phases, an adaptive ascending phase to get to a nearest common ancestor, followed by a deterministic descending phase. The Elite switches can adaptively route a packet picking the least loaded link.

2.3 Programmability and Libraries

The Elan network interface can be programmed using several programming libraries [94], as outlined in Figure 2.3. These libraries trade speed with machine independence and programmability. Starting from the bottom, Elan3lib is the lowest programming level available in user space which allows the access to the low level features of the Elan3 NIC. At this level, processes in a parallel job can communicate with each other through an abstraction of distributed virtual shared memory. Each process in a parallel job is allocated a virtual process id (VPID) and can map a portion of its address space into the Elan. These address spaces, taken in combination, constitute a distributed virtual shared memory. Remote memory (i.e., memory on another processing node) can be addressed by a combination of a VPID and a virtual address. Since the Elan has its own MMU, a process can select which part of its address space should be visible across the network, determine specific access rights (e.g. write- or read-only) and select the set of potential communication partners.

Elanlib is a higher level layer that frees the programmer from the revision-dependent details of the Elan, and extends Elan3lib with point-to-point, tagged message passing primitives (called Tagged Message Ports or Tports) and support for collective communication. Standard communication libraries as such MPI-2 [46] or Cray Shmem are implemented on top of Elanlib.

2.3.1 Elan3lib

The Elan3lib library supports a programming environment where groups of cooperating processes can transfer data directly, while protecting process groups from each other in hardware. The communication takes place at user level, with no data copying, bypassing the operating system. The main features of Elan3lib are: (1) event notification, (2) memory mapping and allocation, and (3) RDMA transfers.

2.3.1.1 Event Notification

Events provide a general purpose mechanism for processes to synchronize their actions. The mechanism can be used by threads running on the Elan and processes running on the main processor. Events can be accessed both locally and remotely. Thus, processes can be synchronized across the network, and events can be used to indicate the end of a communication operation, such as the completion of a remote DMA. Events are stored in Elan memory, in order to guarantee the atomic execution of the synchronization primitives[3]. Processes can wait for an event to be triggered by blocking or polling. In addition, an event can be tagged as being a block copy event. The block copy mechanism works as follows. A block of data in Elan memory is initialized to hold a predefined value. An equivalent sized block is located in main memory, and both are in the user's virtual address space. When the specified event is set, for example when a DMA transfer has completed, a block copy takes place. That is, the block in Elan memory is copied to the block in main memory. The user process polls the block in main memory to check its value, (for example, bringing a copy of the corresponding memory block into the L2 cache) without having to poll for this information across the PCI bus. When the value is the same as that initialized in the source block, the process knows that the specified event has ocurred.

2.3.1.2 Memory Mapping and Allocation

The MMU in the Elan can translate between virtual addresses written in the format of the main processor (for example, a 64-bit word, big Endian architecture as the AlphaServer) and virtual addresses written in the Elan format (a 32-bit word, little Endian architecture). For a processor with a 32-bit architecture, a one-to-one mapping is all that is required[4]. Using allocation functions provided by the Elan library, portions of virtual memory (1) can be allocated either from main or Elan memory, and (2) the MMUs of both main processor and Elan can be kept consistent. For efficiency reasons, some objects can be located on the Elan, for example communication buffers or DMA descriptors which the Elan can process independently of the main processor.

[3] The PCI bus implementations cannot guarantee atomic execution, so it is not possible to store events in main memory.
[4] Details on the mapping for a 64-bit processor can be found in [87].

2.3.1.3 Remote DMA

The Elan supports RDMA transfers across the network, without any copying, buffering or OS intervention. The process that initiates the DMA fills out a DMA descriptor, which is typically allocated on the Elan memory for efficiency. The DMA descriptor contains the VPIDs of both source and destination, the amount of data, the source and destination addresses, two event locations (one for the source and the other for the destination process) and other information used to enhance fault tolerance. A step-by-step description of RDMA operations can be found in [87].

2.3.2 Elanlib and Tports

Elanlib is a machine-independent library that integrates the main features of Elan3lib with Tports. Tports provide basic mechanisms for point-to-point message passing. Senders can label each message with a tag, the sender identity and the size of the message. This is known as the *envelope*. Receivers can receive their messages selectively, filtering them according to the identity of the sender and/or a tag on the envelope. The Tport layer handles communication via shared memory for processes on the same node. It is worth noting that the Tports programming interface is very similar to MPI [102].

Elanlib provides support for collective communication operations (those that involve a group of processes). The most important collective communication primitives implemented in Elanlib are: (1) the barrier synchronization and (2) the broadcast. We explore the special role these collectives can play in system software in the following chapters.

2.4 Performance Evaluation

The performance of interconnection networks and, in particular, switch-based wormhole networks has been extensively analyzed by simulation in the literature [14, 69, 77]. Since performance is strongly influenced by the load, it is important to evaluate QsNet under varied traffic patterns to get a complete view of the network behavior. The patterns considered here are representative of real scientific applications in use at LANL. One example of workload analysis is presented in [61] for SAGE (see Section 1.4). In this chapter, we focus on the most relevant point-to-point and collective communication patters for system software and LANL's applications. More extensive studies and results for other benchmarks and network aspects can be found in [16, 17, 41, 73, 82, 83, 84, 85, 86, 87]. Unless otherwise stated, the communication buffers are allocated in Elan memory in order to isolate I/O bus-related performance limitations, except for the ping tests, whose goal is to provide basic performance results that are a reference point for the following experiments. These measurements were taken on Wolverine.

2.4.1 Point-to-Point Messages

2.4.1.1 Unidirectional Ping

We analyze the latency and bandwidth of the network by sending messages of increasing sizes. In order to identify bottlenecks, the communication buffers are placed either in main or in Elan memory, using the allocation mechanisms provided by Elan3lib described in Section 2.3.1.

At Elan3lib level the latency is measured as the elapsed time between the posting of the RDMA request and the notification of completion at the destination. The unidirectional ping tests for MPI are implemented using matching pairs of blocking sends and receives.

Figure 2.4(a) shows the bandwidth of the unidirectional ping. The peak bandwidth of $335\,MB/s$ is obtained when both source and destination buffers are placed in the Elan memory. The maximum amount of data payload that can be sent by the current Elan implementation in a packet is 320 bytes, partitioned in five low-level write-block transactions of 64 bytes. For this packet format, the overhead is 58 bytes, for the message header, CRCs, routing info, etc. This implies that the delivered peak bandwidth is approximately $396MB/s$, or 99% of the nominal bandwidth ($400MB/s$).

However, the asymptotic bandwidth for main memory to main memory communication is only $200MB/s$ for both Elan3lib and MPI. These results show that the PCI interface running at 33MHz on Wolverine is the bottleneck for this type of communication.

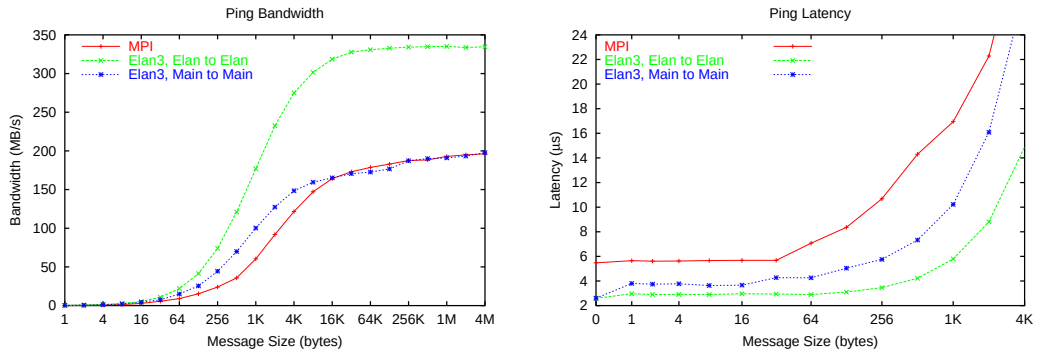
Figure 2.4(b) shows the latency for messages in the range $[0 \ldots 4KB]$. With Elan3lib the basic latency for a zero-byte message is only $2.2\mu s$ and remains nearly constant at $2.4\mu s$ for messages of up to 64 bytes. We note an increase in the latency at the MPI level, from $\approx 2\mu s$ to $5.5\mu s$, due to additional software layers tag matching overhead.

Bidirectional Ping analysis can be found in [87]. The main difference with unidirectional Ping is that bidirectional Ping is much more sensitive to the implementation of the PCI bus. QsNet is designed to handle bidirectional communication nearly as fast as unidirectional one. However, bandwidth for messages that reside in main memory (and thus require transfer through the PCI bus) suffer a significant performance degradation if the PCI bus cannot interleave bidirectional communication successfully.

2.4.1.2 Uniform Traffic

The uniform traffic is one the most frequently used traffic patterns for evaluating the network performance. With this pattern each node randomly selects its destination for each message. This distribution provides what is likely to be an upper bound on the mean inter-node distance because most computations exhibit some degree of communication locality [23, 61].

The results obtained for the uniform traffic pattern with $256KB$ messages are shown in Figures 2.5(a)-(c) for 16, 32 and 64 nodes configurations. The maximum sustained bandwidth versus the network size is depicted in Figure 2.5(d).



(a) (b)

Figure 2.4: Unidirectional Ping performance with QsNet

In the experiments we consider uniform and exponential distributions for both message size (identified by the tag S in the figures) and inter-arrival time (tag T). The results show that small network configurations are somewhat sensitive to these distributions, providing better results when both message size and inter-arrival times are uniformly distributed. This performance gap narrows with larger configurations and is much smaller with 64 nodes.

In Figure 2.5(d) we can see that the uniform traffic performance does not scale well with the network size. For example, with 64 nodes the asymptotic bandwidth is less than $120MB/s$, only 42% of the maximum bidirectional bandwidth, which is an upper bound for uniform traffic. This performance degradation is caused by the flow-control algorithms. Each packet keeps an open circuit between source and destination until the packet is successfully delivered, which increases the probability of having other packets blocked waiting for an available path. This problem is partially alleviated by the two virtual channels, that offer an escape path when one of the channels is busy, but they are not enough to guarantee scalability over a large number of nodes. Still, if we replace all the global communication with collectives operations instead of using the traditional point-to-point messages, significantly better scalability is obtained, as shown in the next section.

2.4.2 Collective Communication

We describe the mechanisms and performance for two basic collective communication operations which we deem as critical for system software: barrier synchronization and multicast/broadcast. For a realistic assessment of the network's scalability on very large scale clusters, these experiments were run on a 1,024-node (4,096 PE) segment of the ASCI-Q cluster at LANL.

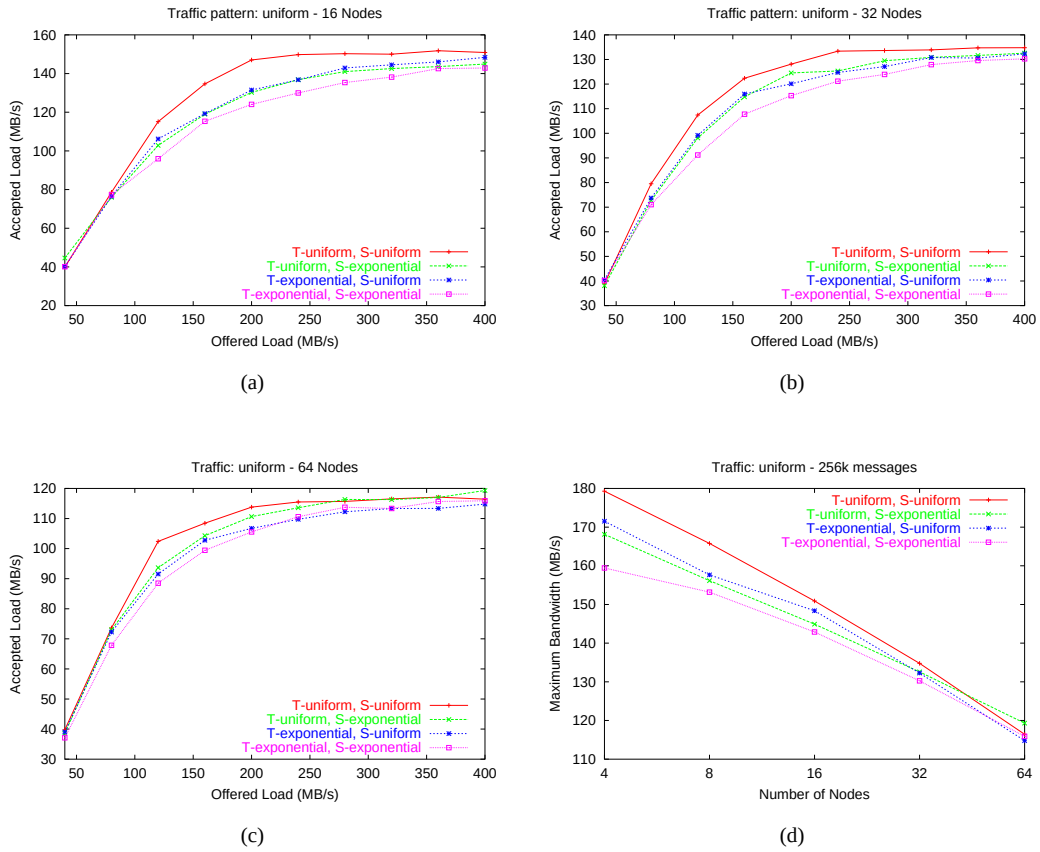
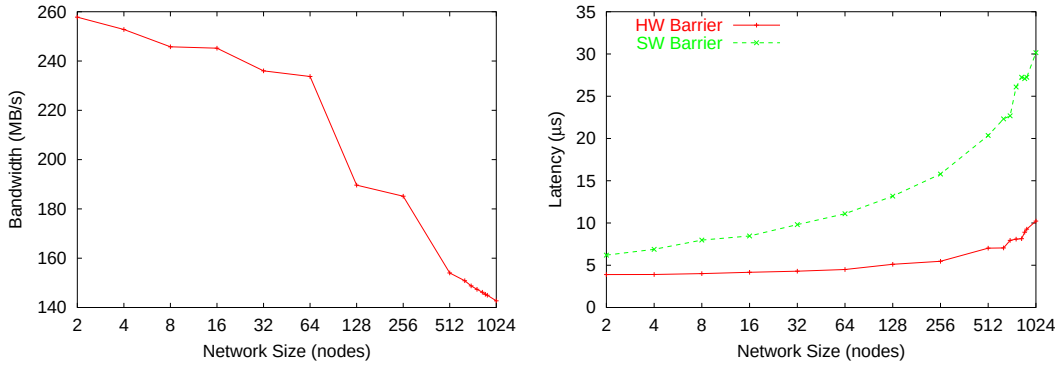


Figure 2.5: Uniform Traffic Scalability with QsNet

2.4.2.1 Broadcast

QsNet provides hardware support in both the NICs and the switches to implement scalable collective communication. Multicast packets can be sent to multiple destinations using either the *hardware* multicast capability of the network or a *software* tree implemented with point-to-point messages exchanged by the Elans without interrupting their processing nodes. These mechanisms constitute the basic blocks to implement collective communication patterns such as barrier synchronization, broadcast, reduce and allreduce.

The implementation of the hardware multicast is detailed in [85]. For a multicast packet to be successfully delivered, a positive acknowledgment must be received from all the recipients of the multicast group. The Elite switches combine the acknowledgments, as pioneered by the NYU Ultracomputer [91], returning a single one to the source. Acknowledgments are combined in a way that the 'worst' ACK wins (a network error wins over an unsuccessful transaction, which in turn wins over a successful one), returning a positive ACK only when all the partners in the collective communication complete the distributed transaction with success. This operation can be generalized so that a positive acknowledgment can only be received if all the nodes meet a certain condition, such as reaching a barrier point, or other binary queries on a variable.



(a) (b)

Figure 2.6: Broadcast bandwidth and barrier latency of QsNet on ASCI Q

The network hardware guarantees the atomic execution of the multicast: either all nodes successfully complete the operation or none. It is worth noting that the multicast packet opens a set of circuits from the source to the destination set, and that multiple transactions (up to 16 in the current implementation) can be pipelined within a single packet. For example, it is possible to conditionally issue a transaction based on the result of a previous transaction. This powerful mechanism allows the efficient implementation of sophisticated collective operations and high-level protocols [35, 44].

The broadcast bandwidth seen at the source node is shown in Fig. 2.6(a). To compute the aggregate bandwidth, we multiply this value by the number of nodes. In the largest configuration, the total bandwidth is $140MB/s \times 1024 = 140GB/s$. The performance degradation experienced when we increase the number of nodes is mostly due to the delays of the communication protocol, and reflects the number of levels in the fat tree. In fact, a multicast packet is blocked until all of its children send an acknowledgment, and is therefore very sensitive to even minor delays and noise created by various system activities [88].

2.4.2.2 Barrier Synchronization

A barrier synchronization is a logical point in the control flow of a parallel program at which all processes in the group must arrive before any of the them is allowed to proceed. Typically, a barrier synchronization involves a logical reduce operation followed by a broadcast.

If all the nodes in the barrier synchronization set are contiguous, it is possible to use the hardware multicast. When the barrier is performed, all processes in the group write a barrier sequence number in a memory location and wait for a global ‘go’ signal (e.g., polling on a memory location). The master process within the root node (the one with the lowest ID) uses an Elan thread to send a special test-and-set multicast packet. This packet spans all the processes and checks if the barrier sequence value in each process matches with its own sequence number (it does if the corresponding process reached the barrier).

Network	Global Comparison (μs)	Multicast (MB/s)
Gigabit Ethernet [101]	$46 \log n$	Not available
Myrinet [8, 12, 13]	$20 \log n$	$\sim 15n$
Infiniband [3, 69]	$20 \log n$	Not available
QsNet [83, 85, 87, 124]	< 10	$> 150n$
BlueGene/L [49]	< 2	$700n$

Table 2.1: Collective operations portability and performance for n nodes

All the replies are then combined by the Elite switches on the way back to the root node which receives a single acknowledgment. If all the nodes are ready, an end-of-packet token is sent to the group to set an event or write a word to wake up the processes waiting in the barrier. This mechanism is completely integrated into the network flow control and is expected to give the best performance when processes enter the barrier fairly close together, otherwise it backs off exponentially (to stop flooding the network with test-and-set multicast packets).

The software algorithm based on point-to-point messages uses a balanced tree to send the 'ready' signal to the lowest-id process. Each process waits for 'ready' signals from its children, and when it receives all of them sends its own signal up to the parent process. When the root process receives all its signals, it performs a broadcast using point-to-point messages to send the 'go' signal with the same tree structure.

The results for both the hardware- and software-based barrier synchronization are shown in Fig. 2.6(b). As expected, the hardware-based multicast is faster than the software mechanism that uses point-to-point messages. In both cases the latency for the largest configuration is remarkable: $10\mu s$ for the hardware barrier and $30\mu s$ for the software one. The network is capable of synchronizing 1,024 nodes with a latency comparable to that of a single point-to-point communication in most contemporary networks [9]. We were also able to run these measurements on the Pittsburgh Supercomputing Center's Alpha-based supercomputer, obtaining a similar barrier synchronization time of $\approx 6.5\mu s$ on 768 nodes [44].

2.4.2.3 Comparative Evaluation

As described in the next chapter, the premise in this dissertation is that most, if not all, of the traditional system software tasks for parallel machines can be expressed as collective communication operations. As such, they stand to benefit considerably from the advent of modern interconnects with hardware support for collective operations. In table 2.1 we discuss the known and expected performance of two such critical operations on several advanced interconnects: the latency of performing a global comparison/synchronization, and the bandwidth of a multicast. All values are given as a function of the number of nodes, to asses the network's scalability for the operation[5]. A detailed analysis of how these two mechanisms scale with QsNet in the context of resource management tasks is presented in Section 4.3.2.

[5]For GigE and Infiniband, reliable bandwidth numbers could not be obtained at the time of this writing.

3 Base Mechanisms

3.1 Motivation

Scalable management of distributed resources is one of the major challenges when building large-scale clusters. Among these tasks, we can find transparent fault-tolerance, efficient deployment of resources, and support for all the needs of parallel applications: parallel I/O, deterministic behavior and responsiveness. These challenges may seem daunting with commodity hardware and operating systems, since they were not designed to support a global, single management view of a large-scale system. An important premise in this thesis is that most of these tasks have many elements in common, and in particular they share the same needs of many HPC applications for high performance communication and synchronization. In the previous chapter we reviewed some of the main features of advanced interconnects that can be exploited to facilitate the implementation of a simple yet powerful global operating system. We now turn to identify a minimal set of network mechanisms that can offer enough functionality and performance to serve as a basis for such an operating system. In our model, the system software is a tightly-coupled parallel application that operates in coordination on all nodes. If the hardware support for this layer is both scalable and efficient, the system software inherits these properties. Such software is not only relatively simple to implement, but can also provide parallel programs with most of the services they require to make their development and usage efficient and more manageable.

3.2 Related Work

Interconnection network and system software designers of HPC clusters traditionally rely on a common abstract machine that clearly separates their territories. Moreover, many actual and research implementations seperated the communication layers for management and applications, and concentrated mostly on one of the two. The abstract machine sees the network as a medium that can move information from one processing node to another, with a given performance expressed by latency and bandwidth. This functional interface is simple and general enough to develop most system software, and can be implemented in several different ways, giving way to explore multiple hardware designs. The success of this interface also

relies on the implicit assumption that any performance improvement in both latency and bandwidth can be directly inherited by the system software. In [36] and summarized in this chapter, a new abstraction layer for large-scale clusters is proposed.

Abstract interfaces can slowly evolve over time, when new factors come into play. For example, in the last decade this basic abstract interface has been augmented with the presence of a distributed, shared memory. In such a global address space, a chunk of data is moved from a source to a destination address. This approach was initially pioneered by communication layers such as Active Messages [118], that emulated a virtual address space on top of physically addressed network interfaces. Active Messages proved that the availability of a global shared memory was able to greatly simplify the communication library and increase its performance. This successful experience was able to influence the design of the Cray T3D [19] and the Meiko CS-2 [72], that provided RDMA. A global, virtually addressed shared memory is nowadays a common feature in networks as QsNet [83] or Infiniband [69].

In this chapter, we review some of the important challenges faced by an operating system for large-scale clusters. We then suggest a small, scalable, and portable set of mechanisms (or primitives) to address these needs, and describe how these mechanisms can be used in each case. The following chapters offer a more detailed evaluation for three system tasks, namely, resource management, user level communication, and job scheduling.

3.3 Challenges in the Design of Parallel System Software

Many of today's fastest supercomputers are composed of COTS workstations connected by a fast interconnect [106, 122]. These nodes typically use commodity operating systems to provide a hardware abstraction layer to programmers and users. As discussed in the introduction, these operating systems are quite adequate for the development, debugging and running of applications on independent workstations and small clusters. However, such a solution is often insufficient for running demanding HPC applications in large clusters.

Common cluster solutions include middleware extensions on top of the workstation operating system, such as the MPI communication library [102] to provide some of the functionality required by these applications. These components tend to have many dependencies and their modular design may lead to redundancy of functionality. For example, both the communication library and the parallel file system used by the applications implement their own communication protocols. Even worse, some desired features such as multiprogramming, garbage collection or automatic checkpointing are not supported at all, or are very costly in terms of both development costs and performance hits.

Consequently, there is a growing gap between the services expected on a workstation, and those provided to HPC users, forcing many application developers to complement these services in their application. Table 3.1 overviews several of these gaps in terms of the basic functionality required to develop, debug,

Characteristic	Workstation	Cluster
Job launching	Operating system (OS)	Scripts, middleware on top of OS
Scheduling	Timeshared by OS	Batch queued or gang scheduled with large quanta (seconds to minutes) using middleware
Communication	OS-supported standard IPC calls and shared memory	Message Passing library (MPI) or Data-Parallel Programming (e.g. HPF)
Storage	Standard file system	Custom parallel file system
Debuggability	Standard tools (most software is deterministic)	Parallel debugging tools (non-determinism)
Fault tolerance	Little or none	Application / application-assisted checkpointing
Garbage collection	Run-time environment such as Java or Lisp	Global GC difficult due to nondeterminism of data's live state [58]

Table 3.1: System tasks in workstations and clusters

and effectively use parallel applications. Let us discuss some of these gaps in detail.

Job Launching

Virtually all modern workstations allow simple and quick launching of jobs, thus enabling interactive tasks such as debugging sessions or visual applications. In contrast, clusters offer no standard mechanism for launching parallel jobs, which entails the dissemination of programs and data to the compute nodes, and their subsequent coordinated execution. Typical cluster solutions rely on scripts or particular middleware modules. Job launching times can range from seconds to hours and are usually far from interactive [37]. Many solutions were suggested in the past to this problem, ranging from the use of generic tools such as rsh and NFS [107, 108], to sophisticated programs such as RMS [42], GLUnix [47], Cplant [11, 97], BProc [52], SLURM [57], and RDGM [59]. Some of these systems use tree-based communication algorithms to disseminate binary images and data to compute nodes, which can shorten job-launch times significantly. However, with larger clusters (of thousands of nodes), these systems are expected to take many seconds or minutes to launch parallel jobs, due to their reliance on software mechanisms.

Job Scheduling

In the workstation world, it is taken for granted that several applications can be run concurrently using time sharing, but this is rarely the case with clusters. Most middleware used for parallel job scheduling use simple versions of batch scheduling (or gang-scheduling at best). This affects both the user's experience of the machine, which is less responsive and interactive, and the system's utilization of available resources. Even systems that support gang scheduling typically revert to relatively high time quanta, in order to hide the high overhead costs associated with context switching a parallel job in software.

The SCore-D and ParPar schedulers uses a combination of software and hardware to perform the global context switch relatively efficiently [24, 54, 55]. A software multicast is used to synchronize the nodes and

force them to flush the network state, to allow each job the exclusive use of the network for the duration of its time slice. The flushing of the network context and the use of software multicast can have a detrimental effect on the time quanta when using a cluster of more than a few hundreds of nodes. In the SHARE gang scheduler of the IBM SP2 [45], network context is switched by the software, where messages that reach the wrong process are simply discarded. This incurs significant overhead, as processes need to recover lost messages. The CM-5 had a gang-scheduling operating system (CMOST) and a hardware support mechanism for network preemption called All-Fall-Down [113]. In this system, all pending messages at the time of a context switch fall down to the nearest node regardless of destination. This creates noticeable delays when the messages need to be re-injected to the system. Even more significantly, this implies that message order and arrival time are completely unpredictable, making the system hard to debug and control. Other machines such as the Makbilan with its common processor bus also had some hardware support for context-switching [22]. However, these specialized machines cost more and did not scale as well as contemporary COTS clusters.

Communication Libraries

User processes running in a workstation communicate with each other using standard interprocess communication mechanisms provided by the OS. While these may be rudimentary mechanisms that provide no high-level abstraction, they are adequate for serial and coarse-grained distributed jobs, due to their low synchronization requirements. Unlike these jobs, HPC applications require a more expressive set of communication tools to keep the software development at manageable levels.

The prevailing communication model for modern HPC applications is message passing, where processes use a communication library to send synchronous and asynchronous messages. Of these libraries, the most commonly used are MPI [102] and PVM [110]. These libraries offer standard interfaces that facilitate portability across various cluster and MPP architectures. On the other hand, much effort is required for the optimization and tuning of the libraries to different platforms, in order to improve the latency and bandwidth for single messages. Another problem is that these libraries offer low-level mechanisms that force the software developer to focus on implementation details, and makes modeling application performance difficult. In order to simplify and abstract the communication performance of applications, several models have been suggested.

The well-known LogP model [20], developed by Culler et al., focuses on latency and bandwidth in asynchronous message passing systems. A higher-level abstraction is the Bulk-Synchronous Parallel (BSP) model introduced by Valiant. in [115]. Computation is divided into *supersteps* so that all messages sent in one superstep are delivered to the destination process at the beginning of the the next superstep. All the processes synchronize between two consecutive supersteps. This model constitutes the first attempt to optimize the communication pattern as a whole rather than optimizing single-message latency and bandwidth.

In our view, optimizing latency, bandwidth, or synchronization speed should not be the focal point of

the underlying communication layer and model. The cost of developing, maintaining, and operating HPC software in a large-scale environment is more significant than another slight improvement in latency. Rather than modeling, we should strive to reduce the "noise" in the global state of the machine (without hurting performance), in order to reduce complexity, facilitate debuggability and reduce non-determinism. The Buffered Coscheduling (BCS) model, presented in detail in Chapter 5, is a methodology that tries to do just that by controlling the communication in the system and coordinating the state of network traffic globally [81]. Communication is only performed at given intervals, after exchanging information on communication requests, scheduling messages accordingly, and sending them in a controlled manner. While this can in some cases hurt the latencies of messages, it converts the on-line problem of managing traffic to an offline, planned solution. As we show in Chapter 5, in most cases this does not hinder application performance in any significant way.

Determinism

Serial applications are much easier to debug compared to their parallel counterparts. This is due mainly to their inherent determinism, making most problems easy to reproduce. Parallel programs are often virtually impossible to trace repeatedly: the independent nature of the system's components – nodes, operating systems, processes and network components – add up to an inherent non-deterministic behavior.

With the approach we present in Chapter 5, the global synchronization of all system activities facilitates the global scheduling of computation and communication resources. Thus, if the scheduling algorithm generates the same sequence of decisions, the behavior of the system will become significantly more predictable allowing for simpler debugging and tracing of parallel programs.

Fault Tolerance

The same non-determinism also makes fault tolerance using checkpointing so challenging, since the application is rarely in a known steady state where all processes and in-transit messages are synchronized. Fault tolerance on workstations is not considered a major problem, and thus rarely addressed by the OS. On large clusters however, where the high number of components results in a low mean time between failures and the amount of computation cycles invested in the program is considerable, fault tolerance becomes one of the most critical issues. Still, there is no standard solution available, and many of the existing solutions rely on some application modifications.

Bosilca et al. introduced a system called MPICH-V to address some of these problems [10]. Their implementation of MPI uses uncoordinated checkpoint/rollback and distributed message logging to convalesce in case of a network fault. MPICH-V requires a rather complex runtime environment, partly due to messages in transit that need to be accounted for. The performance of MPICH-V varies with the application characteristics, sustaining a slowdown of up to 200% or more in some cases. To amortize some of this overhead, the authors use a checkpoint interval of $130s$.

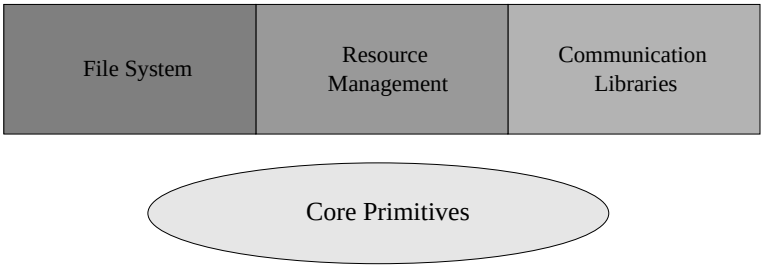


Figure 3.1: Simplified system software model

We believe that with some minimal support from the hardware, a relatively simple fault-tolerant system software can be implemented with much smaller overhead and checkpoint interval. To achieve this, we rely on global synchronization and scheduling of all system activities. In that case, there are points along the execution of a parallel program in which all the allocated resources are in a steady state. Therefore, it is relatively straightforward to implement an algorithm to checkpoint the job in a safe way.

Toward a Global Operating System

The design, implementation, debugging, and optimization of system middleware for large-scale clusters is far from trivial, and potentially very time and resource consuming [63]. System software is required to deal with parallel jobs comprising of thousands of processes each. Furthermore, each process may have several threads, open files, and outstanding messages at any given time. All these elements result in a large and complicated global machine state which in turn increases the complexity of the system software. The lack of global coordination is a major cause of the non-deterministic nature of parallel systems. This behavior makes both system software and user-level applications much harder to debug and maintain. The lack of synchronization also hampers application performance, e.g. when non-synchronized system dæmons introduce computational holes that can severely skew and impact fine-grained applications [88].

To address these issues, we promote the idea of a simple, global cluster OS that makes use of advanced network resources, like any other HPC application. Our vision is that a cluster OS should behave like a SIMD (single-instruction-multiple-data) application, performing resource coordination in lockstep. The basic model is that of a unified set of services, all based on a small set of core network primitives (Fig. 3.1). We argue that performing this task scalably and at sub-millisecond granularity requires hardware support, represented in a small set of network mechanisms. Our goal in this study is to identify and describe these mechanisms. Using a prototype system on a network that supports most of these features, we present experimental results that indicate that a cluster OS can be scalable, powerful, and relatively simple to implement. We also discuss the gaps between our proposed mechanisms and the available hardware, suggesting ways to overcome these limitations.

3.4 Core Primitives

We can now characterize precisely the primitives and mechanisms that we consider essential in the development of system software for large-scale clusters.

Our set of support mechanisms consists of just three hardware-supported network primitives:

XFER-AND-SIGNAL Transfer (PUT) a block of data from local memory to the global memory of a set of nodes (possibly a single node). Optionally signal a local and/or a remote event upon completion. By global memory we refer to data at the same virtual address on all nodes. Depending on implementation, global data may reside in main or network-interface memory.

TEST-EVENT Poll a local event to see if it has been signaled. Optionally, block until it is.

COMPARE-AND-WRITE Compare (using ¿ , ¡ , =, or $\neq$) a global variable on a node set to a local value. If the condition is true on *all* nodes, then (optionally) assign a new value to a (possibly different) global variable.

Note that XFER-AND-SIGNAL and COMPARE-AND-WRITE are both atomic operations. That is, XFER-AND-SIGNAL either PUTs data to *all* nodes in the destination set (which could be a single node) or (in case of a network error) *no* nodes. The same condition holds for COMPARE-AND-WRITE when it writes a value to a global variable. Furthermore, if multiple nodes simultaneously initiate COMPARE-AND-WRITEs with identical parameters except for the value to write, then when all of the COMPARE-AND-WRITEs have completed, all nodes will see the same value in the global variable. In other words, XFER-AND-SIGNAL and COMPARE-AND-WRITE are sequentially consistent operations [65]. TEST-EVENT and COMPARE-AND-WRITE are traditional, blocking operations, while XFER-AND-SIGNAL is non-blocking. The only way to check for completion is to TEST-EVENT on a local event that XFER-AND-SIGNAL signals. These semantics do not dictate whether the mechanisms are implemented by the host CPU or by a network co-processor. Nor do they require that TEST-EVENT yield the CPU (although not yielding the CPU may adversely affect system throughput).

Implementation and Portability

The three primitives presented above assume that the network hardware provides global, virtually addressable shared memory and RDMA. These features are present in several state-of-the-art networks like QsNet and Infiniband and their convenience has been extensively studied [69, 83]. In QsNet however there is a simple 1:1 mapping between these mechansims and hardware primitives, exposed by the Elan3Lib API. Some or all of these mechanisms have already been implemented in several other interconnects as well, as shown in Section 2.4.2.3. Their design was originally meant to improve the communication performance of user applications. To the best of our knowledge, their usage as an infrastructure for system software was not explored before this work.

Hardware support for multicast messages sent with XFER-AND-SIGNAL is convenient in order to guarantee scalability for large-scale systems. Software approaches, while feasible for small clusters, do not scale to thousands of nodes. In our case, QsNet provides hardware-supported PUT/GET operations and events so that the implementation of XFER-AND-SIGNAL is straightforward.

COMPARE-AND-WRITE assumes that the network is able to return a single value to the calling process regardless of the number of queried nodes. Again, QsNet includes a hardware-supported global query operation that allowed us to implement COMPARE-AND-WRITE.

Table 2.1 demonstrates the expected performance of the mechanisms that are already implemented by several interconnect technologies. This table shows that our small set of mechanisms is actually not that far-conceived and is portable to many clusters. While several networks already support at least some of these mechanisms (which attests to their portability), we argue that they should become a standard part of every large-scale interconnect. We also stress that their implementation must emphasize scalability and performance (in terms of bandwidth and latency) for them to be useful to the system software.

3.5 Matching with System Software

We now explain how these mechanisms can be used to tackle the main problems discussed above.

Job Launching

The traditional approach to job launching, including the dissemination of executable and data files to cluster nodes, is a simple extension of single-node job launching: data is disseminated using a network file system such as NFS, and jobs are launched with scripts or simple utilities such as rsh or mpirun. These methods obviously do not scale to large machines, where the load on the network file system and the time it would take to serially execute a binary on many nodes make it impractical. To perform this task scalably, a smarter mechanism must be used to disseminate the data efficiently. Several solutions have been proposed for this problem, all of them focusing on software tricks to reduce the dissemination time. For example, Cplant and BProc both use their own tree-based algorithm to disseminate data with latencies that are logarithmic in the number of nodes [11, 52]. While more portable than relying on hardware support, these solutions are significantly slower and are not always simple to implement [44].

If we break down job launching to simple sub-tasks, we can find that little is required to make it work effectively and scalably:

- Binary and data dissemination are simply a multicast of packets from a file server to a set of nodes that can be implemented using XFER-AND-SIGNAL. We also use COMPARE-AND-WRITE for flow control purposes in order to prevent the multicast packets from overrunning the available buffers.
- Actual launching of a job can again be achieved simply and efficiently by multicasting a control

message to all the nodes that are allocated to the job by using XFER-AND-SIGNAL. The system software on each node would fork the new processes upon receipt of this message, and wait for their termination.

- The reporting of job termination can incur much overhead if each node sends a single message for every process that terminates. This problem can be solved by ensuring that all the processes of a job reach a common synchronization point upon termination (using COMPARE-AND-WRITE) before delivering a single message to the resource manager (using XFER-AND-SIGNAL).

Job Scheduling

Interactive response times from a time-sharing scheduler are required to make a parallel machine as usable as a workstation. This in turn implies that the system should be able to perform preemptive context switching with the same latencies we have come to expect from single nodes, in the order of magnitude of a few milliseconds. Such latencies however are virtually impossible to achieve without hardware support: the time required to coordinate a context switch over thousands of nodes can be prohibitively large in a software-only solution. A good example for this is shown in the work on the SCore-D software-only gang scheduler. In [55], Hori et al. report that the time for switching the network context on a relatively small Myrinet cluster is more than two thirds of the total context switch time. Furthermore, the context switch message is propagated to the nodes using a software-based multicast tree, increasing in latency as the cluster grows. SCore-D has four separate, synchronized phases for each context switch, requiring about $200ms$ context-switch granularity to hide most of the overhead in a 64-node cluster. Finally, even though the system is able to efficiently context switch between different jobs, the coexistence of application traffic and synchronization messages in the network at the same time could eventually make the latter go unattended for some time. If this happens even on a single node and even for a few milliseconds, it will have a detrimental effect on the responsiveness of the entire system [88].

To overcome these problems, the network should offer some capabilities to the software scheduler to elide these delays. The ability to maintain multiple communication contexts alive in the network securely and reliably, without kernel intervention, is already implemented in some state-of-the-art networks like QsNet. Job context switching can be easily achieved by simply multicasting a control message or *heartbeat* to all the nodes in the cluster using XFER-AND-SIGNAL. One obvious solution to guarantee quality of service between application and synchronization messages is prioritized messages. The current generation of many networks, including QsNet, does not yet support prioritized messages in hardware, so a workaround must be found to keep the system messages' latencies low.[1] In our case, we exploit the fact that some of our clusters have dual networks (two rails), and used one rail exclusively for system messages, so that they do not have to compete with application-induced traffic on the same network.

[1] The latest version, QsNet II, already supports prioritized messages

Determinism and fault tolerance

Hori et al. (and the CM-5 before that) proposed a mechanism called network preemption to facilitate tasks such as maintaining a known state of the cluster and context switching [56]. We believe this mechanism is certainly necessary for an efficient solution to this problem, but not sufficient. Even when a single application is running on the system (so there is only one network context, and no preemption), messages can still be en-route at different times, and the system's state as a whole is not deterministic.

When the system globally coordinates all the application processes, parallel jobs can be led to evolve in a controlled manner. Global coordination can be easily implemented with XFER-AND-SIGNAL, and can be used to perform global scheduling of all the system resources. Determinism can be significantly increased by taking the same scheduling decisions between different executions, as is discussed in Chapter 5. At the same time, the global coordination of all the system activities may help to identify the steady states along the program execution in which it is safe to checkpoint the status.

Communication

Most of MPI's, TCP/IP's, and other communication protocols' services can be reduced to a rather basic set of communication primitives, e.g. point-to-point asynchronous messages and multicasts. If the underlying primitives and the protocol reductions are implemented efficiently, scalably, and reliably by the hardware and cluster OS, the higher level protocol can also inherit the same benefits of scalability, performance and reliability. In many cases, this reduction is very simple and can eliminate the need for many of the implementation quirks of protocols that need to run on a variety of network hardware.

To illustrate this strategy, we have implemented a small subset of the MPI library, called BCS-MPI [35], which is sufficiently large to support real applications. As is shown in Chapter 5, these applications have similar performance with our and the production-level version of MPI, but have the potential to benefit from the increased determinism and resource-overlapping advantages of BCS-MPI.

A summary of these mechanisms' usage for the system software needs is shown in Table 3.2. The next chapters delve into a detailed evaluation of some of the more important aspects of system software, and their implementation using these mechanisms.

Characteristic	Requirement	Solution
Job Launching	Data dissemination	XFER-AND-SIGNAL
	Flow control	COMPARE-AND-WRITE
	Termination detection	COMPARE-AND-WRITE
Job Scheduling	Heartbeat	XFER-AND-SIGNAL
	Context switch responsiveness	Prioritized message or multiple rails
Communication	PUT	XFER-AND-SIGNAL
	GET	XFER-AND-SIGNAL
	Barrier	COMPARE-AND-WRITE
	Broadcast	XFER-AND-SIGNAL and COMPARE-AND-WRITE
Storage	Metadata / file data transfer	XFER-AND-SIGNAL
Debuggability	Debug data transfer	XFER-AND-SIGNAL
	Debug synchronization	COMPARE-AND-WRITE
Fault Tolerance	Fault detection	COMPARE-AND-WRITE
	Checkpoint synchronization	COMPARE-AND-WRITE
	Checkpoint data transfer	XFER-AND-SIGNAL
Garbage Collection	Live state synchronization	Determinism and COMPARE-AND-WRITE

Table 3.2: Base mechanisms' usage for system tasks

4 Resource Management

4.1 Background

4.1.1 Problem Description

As discussed in the introduction, cluster hardware is increasingly improving in terms of price and performance, while cluster usability remains poor. The primary cause of this gap is the system software, which tends to consist of a commodity operating system (frequently Linux [125]) running on each node, a batch scheduler (e.g. PBS [51, 126]), and a collection of handcrafted scripts for most of the remaining cluster management. Ideally, a large cluster should be as easy to develop for, use, and manage as a desktop computer. In practice, this is not the case. Table 3.1 contrasts desktops and clusters in terms of a few usability characteristics. In addition to these differences, desktop nodes are also characterized by faster response times, quicker job launching, and seamless multiprogramming, making desktop systems more comfortable to use than clusters. None of the cluster's shortcomings, however, are inherent; they are an artifact of limited resource management (RM) software.

The reason that RM software tends to perform inefficiently is that it has not previously been important to make it efficient. Small clusters have statistically few user-visible failures per unit time, and the global operations needed to launch and schedule applications can be performed quickly even with linear-time algorithms. As a result, more emphasis has been given to making applications run fast than to making RM tools more efficient. However, with cluster sizes reaching almost 10,000 processors [1, 84], resource management can no longer be ignored. Even a small amount of wasted wall-clock time translates to a massive amount of wasted total CPU time. Furthermore, slow or non-scalable RM functions must be amortized by calling them as infrequently as possible. This degrades response time and hinders the usage of interactive jobs.

In this chapter, we describe a scalable solution based on the mechanisms described in Chapter 3 that addresses some of the more important RM issues.

4.1.2 Research Aims

The underlying goal in this chapter is the design and implementation of a scalable, high performing, and simple RM infrastructure. This system, called STORM (Scalable TOol for Resource Management) was also designed to be flexible enough so that it can later be used as a testbed for job scheduling research.

Although many functions lie beneath the umbrella of "resource management," we believe that the small set of mechanisms that was described in Chapter 3 can be used as building blocks to construct a wide variety of RM functions. The intention is that an efficient implementation of these few mechanisms should automatically translate into an efficient implementation of all RM functions implemented on top of them. The RM functions that we considered important and chose to implement include the following:

1. Efficient dissemination of global data to nodes, including binary and data files for parallel programs
2. Responsive (in desktop terms) job launching and termination
3. Responsive (in desktop terms) and low-overhead global context switching
4. Quick detection and isolation of faults
5. Seamless incorporation of various time- and space-sharing scheduling algorithms.

For all these goals, scalability to many thousands of processors was explicitly added as a required feature. Another prominent issue that was considered for STORM was portability, and indeed the system was successfully ported and implemented on several clusters and three different microchip architectures (x86, IA64, and Alpha). STORM was implemented over two interchangeable network layers, QsNet's Elanlib, and generic MPI (which can run on any MPI architecture). For obvious performance reasons, all the tests were run on the lower-level Elanlib implementation.

The results presented below aim to demonstrate that by implementing RM in a manner that is both fast and scalable to large numbers of nodes, clusters can approach the usability of a desktop machine.

4.1.3 The STORM Approach

STORM was developed to provide RM mechanisms that are scalable, high-performance, and lightweight, and to support the implementation of most current and future job scheduling algorithms. We used a common structure for similar RM systems, that is based on a central system dæmon on a management node and helper dæmons on each compute node. Unlike most similar systems, the emphasis in STORM is moved to the network level, stemming from the premise that RM tasks are essentially global communication problems. Thus, all the RM functionality in STORM is reduced to our set of core mechanisms, allowing them to inherit the scalability and performance of these mechanisms' implementation by the network layer. STORM's dæmons communicate with extremely fast messages and collective operations. Coordination of

Dæmon	Number	Location
MM (Machine Manager)	One per cluster	Mngmnt. node
NM (Node Manager)	One per compute node	Compute nodes
PL (Program Launcher)	One per potential process (# of compute nodes × # of processors per node × desired level of multiprogramming	Compute nodes

Table 4.1: STORM dæmons

the dæmons is done through scalable strobes (heartbeat messages) that are implemented with an efficient hardware multicast.

4.2 STORM architecture

As discussed above, STORM is made up of different types of dæmons running in a client-server configuration. Each of these dæmons is a user-level process in charge of specific RM tasks. Connecting these processes is a thin software layer in charge of *all* the communication between the server and the clients. This layer abstracts RM messages such as "send this binary to all nodes" and "notify server of termination of user process". This abstract layer is simple enough that it can be implemented in few hundreds of lines. Underlying this level, we have implemented two interchangeable communication modules. The first uses generic MPI calls, and is useful to port STORM to new network architectures that support MPI, but performs relatively poorly. The second uses our three network mechanisms as described in Chapter 3, which in turn are implemented on top of the QsNet mechanisms. As is shown below, The direct reliance on scalable network mechanisms for RM tasks, using a relatively simple reduction, translates to scalable RM performance. The detailed implementation and algorithms for these tasks can be found in [43].

4.2.1 Process Structure

STORM consists of three types of dæmons that handle job launching, scheduling, and monitoring: the Machine Manager (MM), the Node Manager (NM), and the Program Launcher (PL). Table 4.1 lists the number and location of each of these dæmons.

The MM is in charge of resource allocation for jobs, including both space and time resources. Whenever a new job arrives, the MM enqueues it and attempts to allocate processors to it using a DHC algorithm [27, 30, 32, 127][1]. If the scheduling policy allows for multiprogramming (e.g. GS), the processors are allocated in a time slot that has enough resources available. After a successful allocation, the MM broadcasts a job-launch message to all the NMs, and those NMs on nodes that are allocated to the job launch it when its

[1]The allocation algorithm is implemented in an independent and easily-replaceable module.

time slot arrives.[2] The MM mode of operation is clock-based, operating only at regular ticks, and sleeping or busy-waiting throughout the period of time between ticks, called a *timeslice*. During its wakeup time, the MM proceeds through its chores, including checking for incoming messages and jobs to execute.

When launching a parallel application, the MM first transfers the binary image of the program to each node's local file system (using each node's NM) and then instructs the PLs (again, via the NMs) to launch the application from the local file system. This procedure exploits XFER-AND-SIGNAL in both cases, which can disseminate a file of several megabytes in a fraction of a second to all the nodes, instead of using a slower and less-scalable shared file system, such as NFS [100]. When the job terminates, the MM notes the re-availability of the time/space resources occupied by that job. More details about the termination notification algorithm can be found in [43]. Note that even though we used a centralized approach, which is susceptible to scalability problems, in reality no bottleneck situation is created. This is a direct result of using the scalable network mechanisms described in Chapter 3 for all the global management operations, while the local operations – accepting a new process, allocating resources to it, and receiving process-termination notifications—are rare and lightweight. In fact, if the MM's node is also configured as compute node, the MM sleeps between timeslice intervals to maximize CPU availability and only performs its operations when a new time slot is due.

NMs are responsible for managing resources on a single node (which is many times an SMP). NMs work asynchronously by responding to the following types of events:

Job launch If the job pertains to the NM's node, the NM finds available PLs and sends them the job information.

Job caching The binary image is read from the communication layer and stored in a file, preferably in a RAM-disk file to avoid unnecessary I/O.

Heartbeat/strobe The NM checks in its local data structures, for every PE, whether another process occupies the next time slot. If so, it deschedules the current process (using UNIX's SIGSTOP [107]) and resumes the next one.

Process termination The NM keeps track of whether all of its PLs have terminated. When they have, the NM sets a flag to notify the MM.

At all other times, the NMs block, leaving the CPU to the application processes. Note that some scheduling algorithms require that the NMs make their own local scheduling decisions. As a trivial example, in local scheduling, the NM ignores context-switch messages, as the UNIX scheduler handles all scheduling decisions. In algorithms such as FCS [39], BCS [81] or ICS [2], the NM might deschedule a process that is blocked for communication before the expiration of the time slot and schedule another process instead, in order to increase resource utilization.

[2] On QsNet, it is simpler—and just as fast—to broadcast to all NMs using XFER-AND-SIGNAL, and discard unnecessary messages, than to define a multicast group for every combination of NMs.

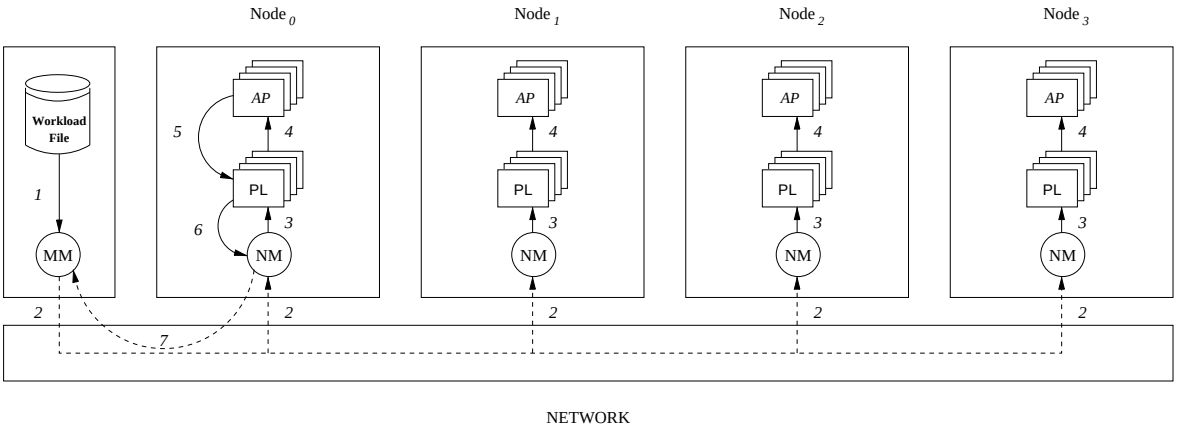


Figure 4.1: Running a job in STORM

Each PL has the relatively simple task of handling an individual application process for the NM. A PL sleeps until it receives a program execution message from the NM. It then proceeds to fork a new process, set up the network initialization procedures for the application process (AP), redirect standard output and error to the console that launched STORM, and execute the AP. It then blocks with the `wait()` system call until the AP terminates. Finally, it notifies the NM when this happens and returns to sleep.

The PL may seem redundant, since all of its tasks can be performed by the NM. In fact, we plan to eliminate the PL in a future version of STORM, but it is currently required due to a limitation in the QsNet capability creation process.

4.2.2 Running a Job

Figure 4.1 illustrates the process of running a job with STORM. This example shows a management node and four SMP nodes with two PEs each. The arrows represent information flow (with dashed lines representing network messages and solid lines representing shared-memory communication), and the numbers on the arrows indicate the order of the events, based on the following key:

1. The MM receives the job information and queues it according to its given arrival time and resource availability. STORM can receive the job interactively or via a workload file, to facilitate complex evaluations (as shown in Fig. 4.1).

2. When the job's time to run has come, and resources have been allocated, the MM broadcasts the job information (possibly with the binary image if not locally available) to all of the NMs. It is worth noting that this multicast (using XFER-AND-SIGNAL) is performed by a thread in the Elan NIC so as not to interrupt the CPU. Furthermore, the multicast uses an I/O bypass mechanism, as described in Section 4.2.3 below.

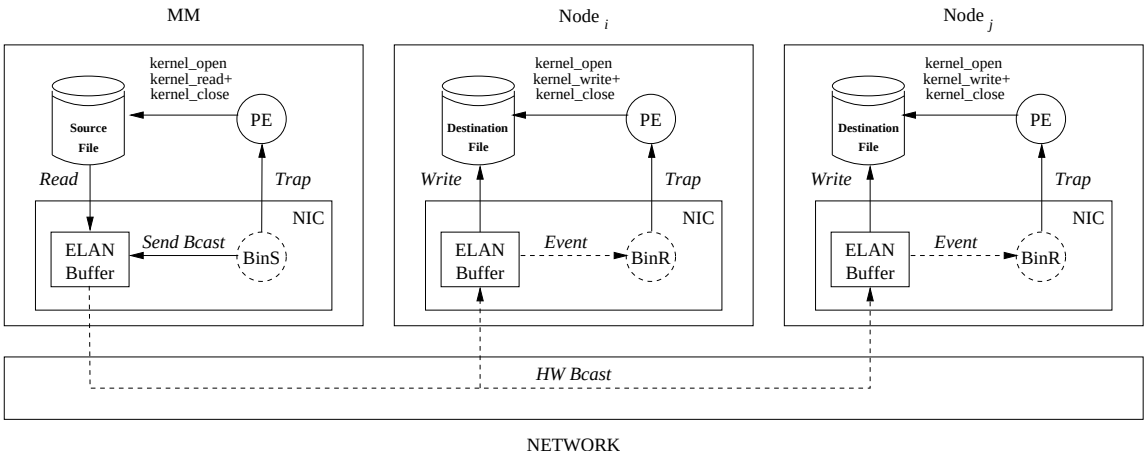


Figure 4.2: I/O bypass mechanism

3. NMs on participating nodes identify the appropriate PLs (according to the job's allocation). The NMs then communicate the job information to the PL using shared memory.

4. The PLs execute the APs, as described in Section 4.2.1.

5. When an AP terminates, the PL receives a notification from the operating system.

6. The PLs then proceed to notify the NM of the termination of the AP.

7. Finally, the NMs send an asynchronous point-to-point message to the MM, which inspects these messages before issuing the next strobe and deallocating resources. This can be a collection of point-to-point messages (as depicted) or a global check using COMPARE-AND-WRITE.

4.2.3 I/O Bypass

One of the major bottlenecks associated with program launching is the interaction with the I/O subsystem. To alleviate this bottleneck, we implemented a mechanism for streamlining the I/O associated with program and data dissemination. We specifically make use of the threads in the Elan NIC, which can issue system calls (using a kernel helper thread) that operate on the file system, such as opening, reading, writing, and closing files. The relevant phases of the I/O bypass protocol during the launch of a job are listed below and shown in Figure 4.2. Note that kernel_read+ and kernel_write+ indicate sequences of kernel reads and writes. BinS and BinR are the binary sender and binary receiver threads running on the Elan NIC.

1. The MM sends a DMA message to a thread in the local Elan NIC with the source file name and a remote destination path.

2. The sender thread uses kernel traps to open and read the source file. These traps go through the kernel but require very little CPU intervention. Compute-bound processes running on the host are

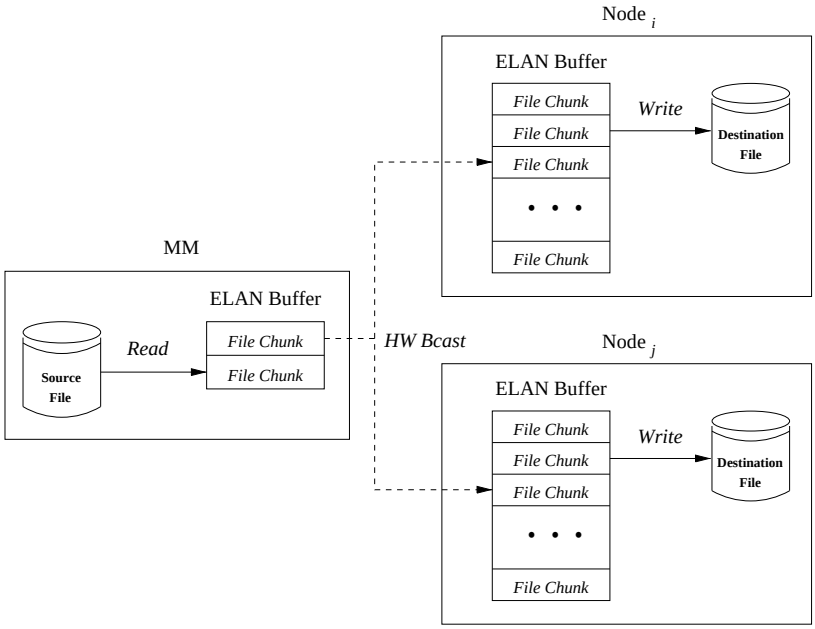


Figure 4.3: Pipelining of I/O read, hardware multicast, and I/O writes

not noticeably affected.

3. The file is read in chunks directly into a communication buffer that can be efficiently accessed by the Elan DMA engine. The file is then sent to a peer thread on all of the compute nodes using QsNet's hardware multicast.

4. The sender thread uses a double-buffering scheme to pipeline the reading and multicast operations, so that while one buffer is being read, the other is concurrently being sent, as shown in Figure 4.3.

5. The destination threads on the compute nodes queue the incoming chunks and write them to the destination path using a flow control protocol to avoid buffer overflows. File system writes and incoming multicasts can proceed in parallel.

6. When all of the chunks have been sent and written to their respective local files (or conversely, if an error occurred), the MM is notified.

7. When the MM decides to launch the job (after successfully sending the binary and allocating resources to it), it uses the new remote path name in the job-launch message.

4.3 Analysis

In this section, we analyze STORM's performance. In particular, we:

1. Measure the costs of launching jobs in STORM, and

2. Test various aspects of the gang scheduler (effect of the timeslice quantum, node scalability and multiprogramming level).

the job launching experiments were carried out on the Wolverine cluster, which at the time of the measurements was ranked no. 83 at the top500 list [122]. Since we could not access it for the other series of experiments, they were performed on the other two clusters.

4.3.1 Job Launching Time

In this set of experiments, we study the overhead associated with launching jobs with STORM and analyze its scalability with the size of the binary and the number of PEs. We use the approach taken by Brightwell et al. in their study of job launching on Cplant [11], viz. we measure the time it takes to run a do-nothing program of sizes 4MB, 8MB, or 12MB that terminates immediately.[3]

4.3.1.1 Launch times in STORM

STORM logically divides the job-launching task into two separate operations: The transferal (reading+broadcasting+writing+notifying the MM) of the binary image, and the actual execution which includes sending a job-launch command, forking the job, waiting for its termination, and reporting back to the MM. In order to reduce non-determinism, the MM can issue commands and receive the notification of events only at the beginning of a timeslice. Therefore, both the binary transfer and the actual execution take at least one timeslice. To minimize the MM overhead and expose maximal protocol performance, we use a relatively short timeslice value of $1ms$ in the following job-launching experiments.

Figure 4.4 shows the time needed to transfer and execute a do-nothing application of sizes $4MB$, $8MB$, and $12MB$ on 1–256 processors. Observe that the send times are proportional to the binary size but grow only slowly with the number of nodes. This can be explained by the highly scalable algorithms and hardware broadcast that are used for the send operation. On the other hand, the execution times are mostly independent of the binary size but grow more rapidly with the number of nodes. The reason for this growth is the skew that is accumulated by the processes in the job. The main cause of this skew is the overhead of the operating system and its asynchronous dæmons [88]. In the largest configuration tested, a $12MB$ file can be launched in $110ms$, a remarkably low latency. In this case, the average transfer time is $96ms$ (a protocol bandwidth of $125MB/s$ per node, with an aggregate bandwidth of $7.87GB/s$ on 63 nodes[4]), The average job execution time is $14ms$. In Section 4.3.2.2, we analyze in depth the specific effects and scalability of each of these two components: transfer and launch.

[3] The program contains a static array, which pads the binary image to the desired size. Note that this is equivalent to sending a fixed binary file with different data sets.

[4] The binary transfer does not include the source node.

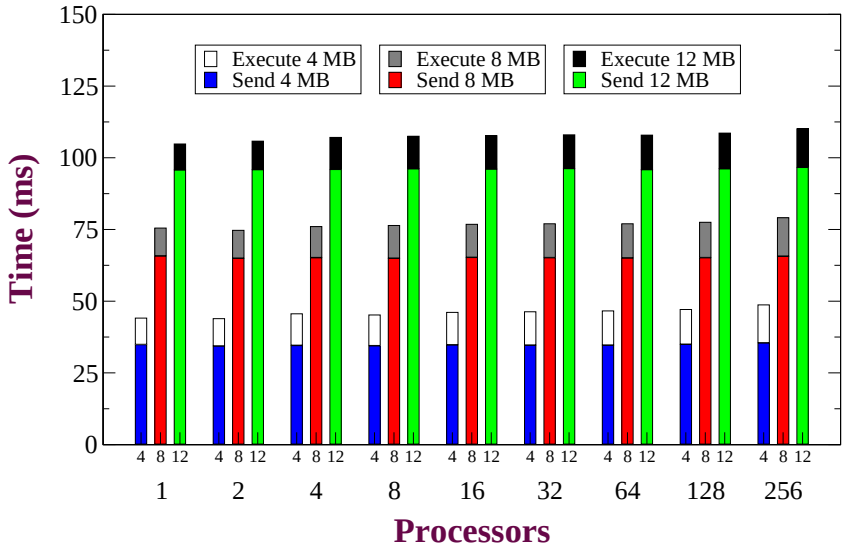


Figure 4.4: Send and execute times for a $4MB$, $8MB$, and $12MB$ file on an unloaded system

4.3.1.2 Launching on a loaded system

To test how a heavily-loaded system affects the launch times of jobs, we wrote two programs that artificially load the system in a controlled manner. The first program performs a tight spin-loop, which introduces CPU contention. The second program repeatedly issues point-to-point messages between pairs of processes, which introduces network contention. Both programs are run on all 256 processors. We measured the same experiments as those used in Section 4.3.1.1, but with either the CPU-consuming program or the network-bandwidth-consuming program simultaneously running on all nodes of the cluster.

Figure 4.5 shows the results of launching the same three do-nothing binaries while the CPU-consuming program is running in the background. Note the different scale on the y axis from that in Figure 4.4. In this scenario, STORM is required to deschedule the CPU-consuming program, run the application, repeatedly reschedule the CPU-consuming program or application on every timeslice, and finally notify the MM when the benchmark application terminates. This experiment shows that a CPU load exacts a significant price in both send time and execution time. The combined launch and execution time is now close to $1s$ in the largest configuration tested and with a $12MB$ binary. This large increase in time is due to the interference of the computation with the I/O activities (reads and writes). Elan-induced I/O is implemented via a user-level lightweight process running on the host. This lightweight process is subject to the same local scheduling policies as any other UNIX process. Since STORM's dæmons, the application, and the CPU-consuming program all utilize all four CPUs, there are no CPUs remaining to service QsNet's lightweight I/O-handling process. The result is an increased sensitivity to CPU load. We suspect we may decrease this sensitivity by moving some of STORM's functionality into the kernel.

The second load-inducing program is designed to stress the network by pairing all of the processors

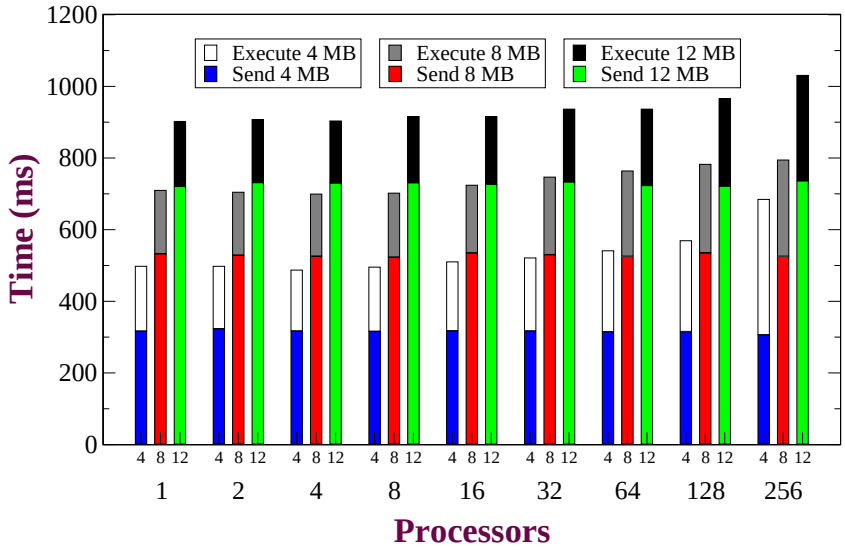


Figure 4.5: Send and execute times on a CPU-loaded system

and continuously sending long messages back and forth between them. This test is particularly relevant to STORM in this implementation, since we have seen in Section 2.4.2 that a heavily-loaded network has an adverse effect on collective communications in the QsNet interconnect (see also [84, 85]). In Figure 4.6, we can see how running the network-loading program in the background affects the launch time of the test binaries (again, note the different scale used on the y axis.) The execution part does increase to $160ms$ in the worst case, due mostly to the increased delays in the collection of the termination info. However, it increases less than in the previous experiment. In contrast, the send operation is considerably slower than on a CPU-loaded or unloaded system. This agrees with our previous results, as the send operation is implemented using XFER-AND-SIGNAL, on top of a QsNet collective.

Figure 4.7 summarizes the difference among the launch times on loaded and unloaded systems. In this figure, the send and execute times are shown under the three loading scenarios (U – unloaded, C – CPU loaded, and N – network loaded), for the $12MB$ file only. Note that even in the worst scenario, with a network-loaded system, it still takes only $1\frac{1}{2}$ seconds to launch a $12MB$ program on 256 processors.

4.3.2 Scalability and Job Launch Analysis

We now turn to list the different components involved in STORM's job launching, and present an analytical model showing how its performance is expected to scale to cluster configurations with thousands of nodes.

4.3.2.1 Performance Analysis

The time needed to launch a parallel job can be broken down into the following components:

Read time This is the time taken by the management node to read the application binary from the file sys-

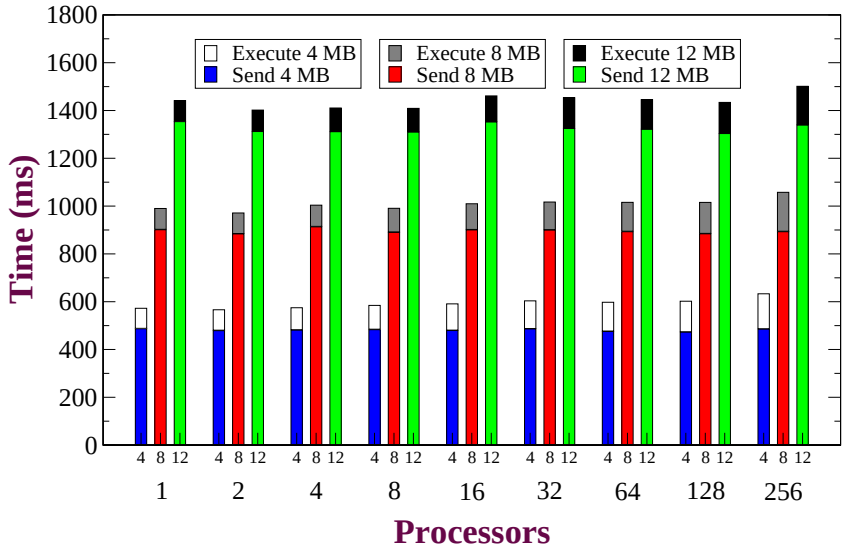


Figure 4.6: Send and execute times on a network-loaded system

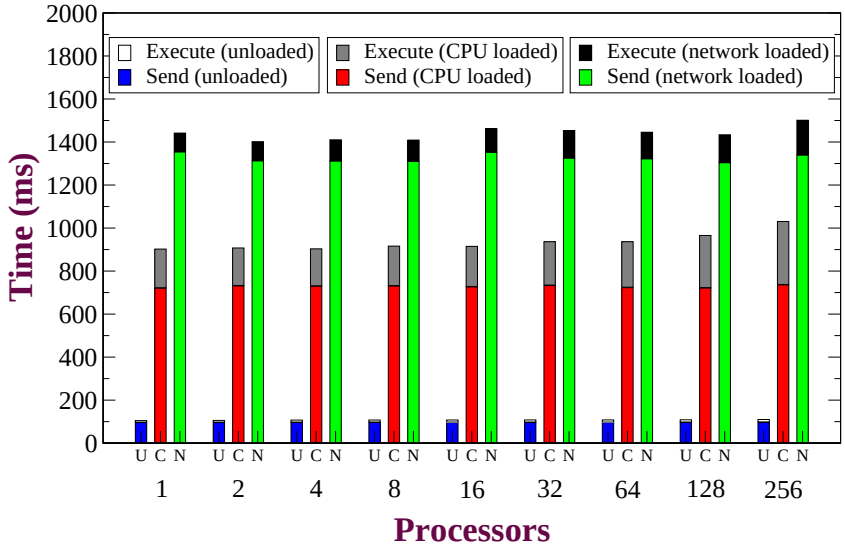


Figure 4.7: Send and execute times for a $12MB$ file under different loading scenarios

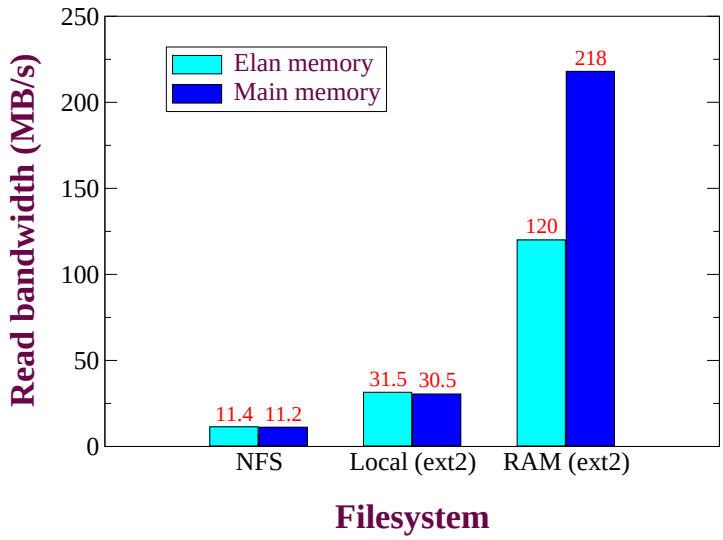


Figure 4.8: Read bandwidth for different files systems and buffer locations ($12MB$ file)

tem. The image can be read from a distributed filesystem such as NFS [100], from a local hard disk, or from a RAM disk.[5] In the case of Wolverine, the NIC can read a file directly from the RAM disk at $218MB/s$. Figure 4.8 shows the bandwidth achieved when the NIC—with assistance from a lightweight process on the host—reads a $12MB$ file from various types of filesystems into either a host- or NIC-resident buffer. As the figure shows, in the slow cases (NFS and local disk), it makes little difference whether the target buffers reside in main memory or NIC memory. However, when reading from a (fast) RAM disk, the data clearly show that keeping data buffers in main memory gives the better performance.

Broadcast time This is the time to broadcast the binary image to all of the compute nodes. If the file is read via a distributed filesystem like NFS, which supports demand paging, the distribution time and the file read time are intermixed. The total launch time in a pipelind implementation of the slowest of read, write, and broadcast times. QsNet's hardware broadcast is both scalable and extremely fast. On the ES40 Alphaserver, the performance for a main-memory-to-main-memory broadcast is therefore limited by the PCI I/O bus. As was shown in [44], the hardware broadcast on 64 nodes can deliver $312MB/s$ when the buffers are in NIC memory but only $175MB/s$ when the buffers are placed in main memory.

Write time We are concerned primarily with the overhead component of the write time. It does not matter much if the file resides in the buffer cache or is flushed to the (RAM) disk. A number of experiments—for brevity, not reported here—show that the read bandwidth is consistently lower than the write bandwidth. Thus, the write bandwidth is not the bottleneck of the file-transfer protocol.

[5] A RAM disk is a segment of RAM that has been confi gured to simulate a disk fi lesystem. This provides better performance than mechanical media but at increased cost, as DRAM is more expensive than disk media for a given capacity.

Execution overhead Some of the time needed to launch a job in STORM is spent waiting for a time slot in which to run the job and collect the termination information in the management node. In our experiments the execution overhead is approximately $10ms$.

Timeslice overhead In addition, events such as process termination are collected by the MM at heartbeat intervals only, so a delay of few time slots can be suffered while the MM processes requests.

The overall launch time T_{launch} can be expressed by the following equation

$$T_{launch} = T_{transfer} + T_{exec} + T_{timeslice} \tag{4.1}$$

Where $T_{transfer}$ represents the binary transfer delay, T_{exec} the execution overhead and $T_{timeslice}$ the overhead induced by STORM's scheduling mechanisms.

Our implementation tries to pipeline the three components of file-transfer overhead—read time, broadcast time, and write time—by dividing the file transmission into fixed-size chunks and writing these chunks into a remote queue that contains a given number of slots. In order to optimize the overall bandwidth of the pipeline, $BW_{transfer}$, we need to maximize the bandwidth of each single stage. $BW_{transfer}$ is bounded above by the bandwidth of the slowest stage of the pipeline:

$$BW_{transfer} \leq \min(BW_{read}, BW_{broadcast}, BW_{write}) = \min(BW_{read}, BW_{broadcast}) \tag{4.2}$$

As previously stated, the buffers into which data is read and from which data is broadcast can reside in either main memory or NIC memory. We have seen that reading into main memory is faster, while broadcasting from NIC memory is faster. The preceding inequality dictates that the better choice is to place the buffers in main memory, as $\min(BW_{read}, BW_{broadcast}) = \min(218MB/s, 175MB/s) = 175MB/s$ when the buffers reside in main memory, versus $\min(BW_{read}, BW_{broadcast}) = \min(120MB/s, 312MB/s) = 120MB/s$ when they reside in NIC memory.

We empirically determined the optimal chunk size and number of multi-buffering slots (i.e. receive-queue length) for our cluster in [44]. The communication protocol is largely insensitive to the number of slots, and the best performance is obtained with two slots of $512KB$. Increasing the number of slots does not provide any extra performance, because it generates more TLB misses in the NIC's virtual memory hardware.

Figure 4.4 showed that the transfer time of a $12MB$ binary is about $96ms$. Of those $96ms$, $4ms$ are owed to skew caused by the OS overhead [88] and the way that STORM dæmons act only on heartbeat intervals ($1ms$). The remaining $92ms$ is determined by a file-transfer-protocol bandwidth of about $131MB/s$. The gap between the previously calculated upper bound, $175MB/s$, and the actual value of $131MB/s$ is due to unresponsiveness and serialization within the lightweight process running on the host, which ser-

vices TLB misses and performs file accesses on behalf of the NIC. Future versions of the QsNet software libraries are supposed to address this problem.

Figure 4.9 illustrates the steps involved in the file-transfer protocol and shows the performance of each stage of the pipeline. The file transfer protocol is initiated by the master node, which broadcasts a descriptor containing information about the data size, destination filename and directory, access rights, etc. At this point, the master opens the source file in read mode and each slave opens the destination file in write mode ("Open file" in Figure 4.9). In the main loop, the master reads a file chunk from the filesystem ("Read chunk"), waits until *all* the slaves are ready to accept it ("Await space"), multicasts the chunk to all of the slaves ("Send chunk"), and waits for an acknowledgment from the network ("Await sent"). Note that the master overlaps the sending of one chunk with the reading of the subsequent chunk. The slaves perform the complementary operations from the master; they repeatedly wait for a chunk from the master ("Await received") and write it to disk ("Write chunk"). The filesystem is the bottleneck in the file transfer. All of the network operations (communication and flow control) take microseconds to complete, while most of the filesystem operations have latencies measured in milliseconds.

4.3.2.2 Scalability Analysis

Because all of STORM's functionality is based on three mechanisms, the scalability of these primitives determines the scalability of STORM as a whole. In fact, TEST-EVENT is a local operation, so scalability is actually determined only by the remaining two mechanisms.

Scalability of COMPARE-AND-WRITE Figure 2.6(b) shows the scalability of QsNet's hardware barrier synchronization (on which COMPARE-AND-WRITE is based [82]) on the ASCI Q supercomputer [84]. We can see that the latency approximately doubles for an increase of 1024X in the number of nodes, and still remains at a very low number of $\approx 10\mu s$. This is a reliable indicator that COMPARE-AND-WRITE, when implemented with the same hardware mechanism, will scale as efficiently.

Scalability of XFER-AND-SIGNAL In order to determine the scalability of XFER-AND-SIGNAL to a large number of nodes, we need to carefully evaluate the communication performance of the hardware broadcast, consider details of the hardware flow control in the network, and take into account the wire and switch delays. QsNet transmits packets with circuit-switched flow control. A message is chunked into packets of 320 bytes of payload, and the packet with sequence number i can be injected into the network only after the successful reception of the ACK token of packet $i-1$. On a broadcast, an ACK is received by the source only when all of the nodes in the destination set have successfully received packet $i-1$. Given that the maximum transmission unit of the QsNet network is only 320 bytes[6], in the presence of long wires

[6]This limitation does not apply to the latest version 2 of QsNet, which allows packets of variable length.

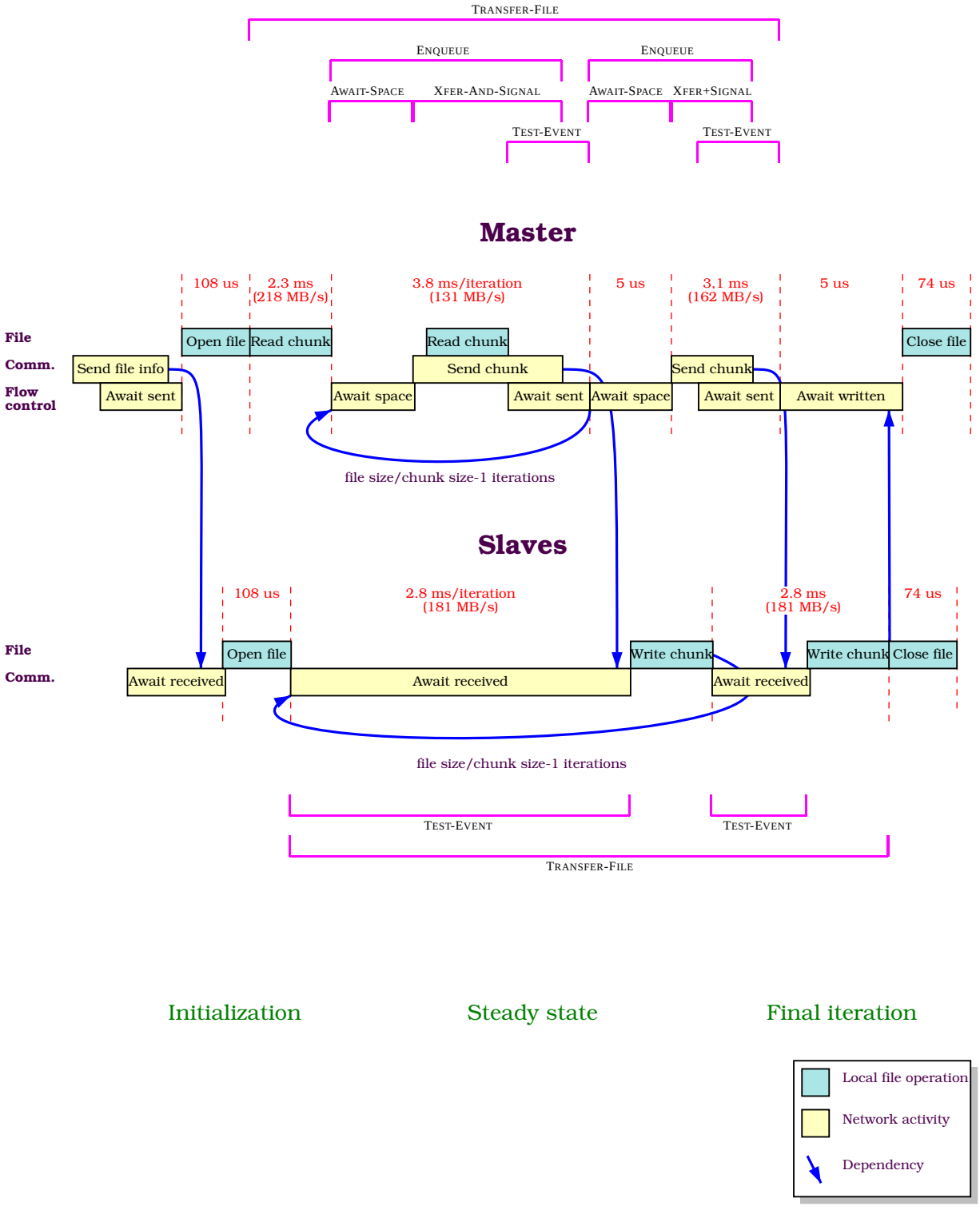


Figure 4.9: Transmission pipeline

(a) The destination NIC sends an acknowledgment after receiving the header of the packet.

(b) If the acknowledgment arrives at the source when the packet is still injected, there is no performance degradation.

(c) therwise the protocol introduces a gap in the transmission.

Figure 4.10: End-to-end flow control in the QsNet network

Component	**Description**	**Value**
Cable	Maximum cable length between any pairs of nodes	input parameter
Switches	Maximum number of switches crossed by a packet	input parameter
Packet Size	Maximum packet size	320 bytes
T_{packet}	Minimum packet delay at peak bandwidth	$953ns$
T_{base}	Base delay for packet processing	$580ns$
T_{cable}	Cable delay per meter	$3.94ns/m$
T_{switch}	Sum of the forward data delay and ACK delay	$73ns$

Table 4.2: Legend of terms used in the scalability model

and/or many switches, the propagation delay of the acknowledgment token can introduce a bubble in the communication protocol's pipeline and hence a reduction of the asymptotic bandwidth.

Figure 4.10 describes QsNet's end-to-end flow-control algorithm. The destination NIC sends an acknowledgment token immediately after receiving the packet header (Figure 4.10(a)). If the token arrives at the source NIC while the packet body is still being transmitted (Figure 4.10(b)), the next packet in the sequence can proceed without delay. Otherwise, the protocol introduces a transmission gap before injecting the following packet (Figure 4.10(c)).

Equation 4.3 describes the asymptotic bandwidth of QsNet as a function of the maximum cable length and the number of switches that a packet must traverse in the worst case. The equation distinguishes the case where the protocol can pipeline packets without interruptions, thus delivering the peak bandwidth, and the other case, in which the combination of wire and the switch delays introduces communication gaps.

$$BW_{QsNet}(cable, switches) = \frac{Packet\,size}{\max(T_{packet}, T_{base} + 2 \times Cable \times T_{cable} + 2 \times Switches \times T_{switch})} \tag{4.3}$$

Table 4.2 describes the components of Equation 4.3 and provides an estimate of their values. This

Nodes	Processors	Stages	Switches	$10m$	$20m$	$30m$	$40m$	$50m$	$60m$	$70m$	$80m$	$90m$
4	16	1	1	**320**	320	320	315	291	271	253	238	224
16	64	2	3	**320**	319	295	274	256	240	226	213	202
64	256	3	5	**298**	277	258	242	228	215	204	194	184
256	1024	4	7	261	**244**	230	217	206	195	186	177	170
1024	4096	5	9	232	219	207	197	**187**	178	171	163	157
4096	16384	6	11	209	198	188	180	172	164	158	152	**146**

Table 4.3: Bandwidth scalability for different cable lengths

Figure 4.11: The ASCI Q machine at LANL

analytical model was used in the procurement of the ASCI Q machine [1] at LANL and has been validated on several network configurations with a prediction error of less than 5%. Table 4.3 shows the asymptotic bandwidth BW_{QsNet} for networks with up to 4,096 nodes and physical diameters of up to $90m$.

To make BW_{QsNet} (and consequently $BW_{broadcast}$) dependent upon only the number of nodes, we compute a conservative estimate of the diameter of the floor plan of the machine, which approximates the maximum cable length between two nodes. We assume that computers in the cluster are arranged in a square. Considering that with current technology (see Fig. 4.11) we can stack between four and six ES40 Alphaserver nodes in a single rack with a footprint of $1m^2$, we estimate the floor space required by four nodes to be $4m^2$ ($1m^2$ for the rack surrounded by $3m^2$ of floor space). The following equation therefore provides a conservative estimate of the diameter in meters as a function of the number of nodes:

$$Diameter = \sqrt{2 \times nodes} \tag{4.4}$$

In a quaternary fat tree, the maximum number of switches traversed by a packet, as a function of the

number of nodes is:

$$Switches(nodes) = (2 \times \log_4(nodes)) - 1 \tag{4.5}$$

By replacing the cable length and the number of switches in Equation 4.3, we obtain the asymptotic bandwidth BW_{QsNet} as a function of the number of nodes:

$$BW_{QsNet}(nodes) = \frac{Packet\,size}{\max(T_{packet}, T_{base} + 2 \times \sqrt{2 \times nodes} \times T_{cable} + [(2 \times \log_4(nodes)) - 1] \times T_{switch})} \tag{4.6}$$

Scalability of the Binary Transfer Protocol We now consider a model of the launch time for a binary of $12MB$. The model contains three parts. The first part represents the actual transmission time and is inversely proportional to the available bandwidth for the given configuration. The second part is the local execution time of the job, followed by the notification to the MM, which is about $10ms$. The third part is the timeslice overhead, which is spent in OS overhead and waiting for the end of the STORM timeslices.

$$T_{launch}(nodes) = \frac{12}{BW_{transfer}(nodes)} + T_{exec} + T_{timeslice} \tag{4.7}$$

We now apply this model to two node configurations. The first, represented by Equation 4.8, represents our current cluster Wolverine, based on ES40 Alphaservers that can deliver at most $131MB/s$ over the I/O bus, while the second configuration, Equation 4.9, represents an idealized Alphaserver cluster that is limited by the network broadcast bandwidth (i.e. the I/O bus bandwidth is greater than the network broadcast bandwidth).

$$BW^{ES40}_{transfer}(nodes) = \min(131, BW_{broadcast}(nodes)) \tag{4.8}$$

$$BW^{ideal}_{transfer}(nodes) = BW_{broadcast}(nodes) \tag{4.9}$$

Figure 4.12 shows measured launch times for network configurations up to 64 nodes and estimated launch times for network configurations up to 16,384 nodes. The model shows that in an ES40-based Alphaserver, the launch time is scalable and only slightly sensitive to the machine size. A $12MB$ binary can be launched in $135ms$ on 16,384 nodes. The graph also shows the expected launch times in an ideal machine in which the I/O bus is not the bottleneck (and in which a lightweight processes on the host can responsively handle the requests of the NIC). Both models converge with networks larger than 4,096 nodes, because for such configurations they share the same bottleneck, which is the network broadcast bandwidth.

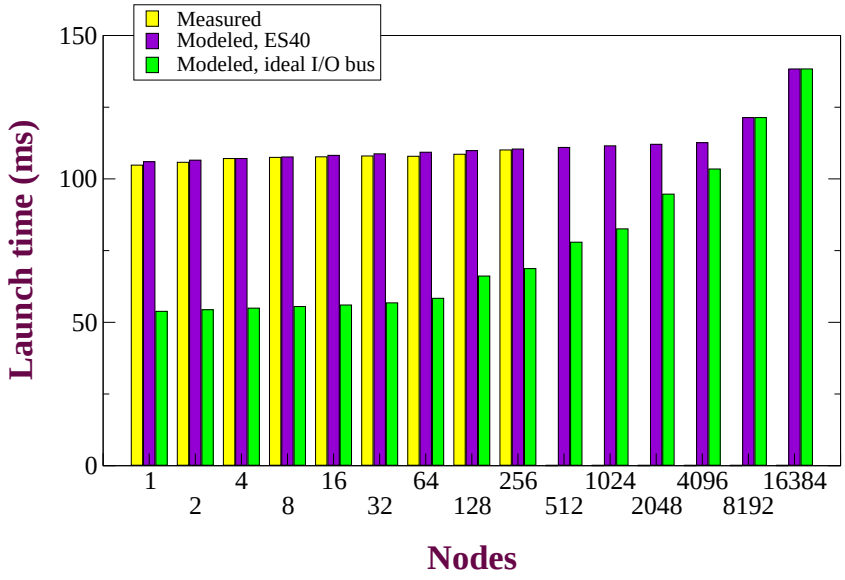


Figure 4.12: Measured and estimated launch times

4.3.3 Multiprogramming Performance

Although STORM supports a variety of process scheduling algorithms we conducted a scheduling performance and overhead test specifically with gang scheduling [22, 31, 78], which is one of the simplest and most popular coscheduling algorithms. The following are the issues we address regarding gang scheduling:

Effect of timeslice on overhead Smaller timeslices yield better response time at the cost of decreased throughput (due to scheduling overhead that cannot be amortized). In Section 4.3.3.1, we evaluate STORM's scheduling overhead and show it to be low enough to support workstation time quanta with virtually no performance penalty.

Scalability Because gang scheduling requires global coordination, the cost of enacting a global decision frequently increases with the number of processors. Section 4.3.3.2 demonstrates that by using scalable network primitives, STORM exhibits such low overhead that applications running on large clusters can be coscheduled about as rapidly as on small clusters or workstations.

Effect of MPL The multiprogramming level (MPL) is the amount of over-subscription of processors to processes. Ideally, if there are P processes per processor (i.e. MPL $=$ P), the turnaround time should be P times what it would be with a single process per processor (i.e. MPL $=$ 1). In practice, schedulers require a certain amount of time to switch processes, which causes performance degradation. In addition, context switches often flush the working set that resides in the cache memory, further hampering the applications. We evaluate these effects in Section 4.3.3.3, and provide data showing that application performance under STORM is robust to increased MPL.

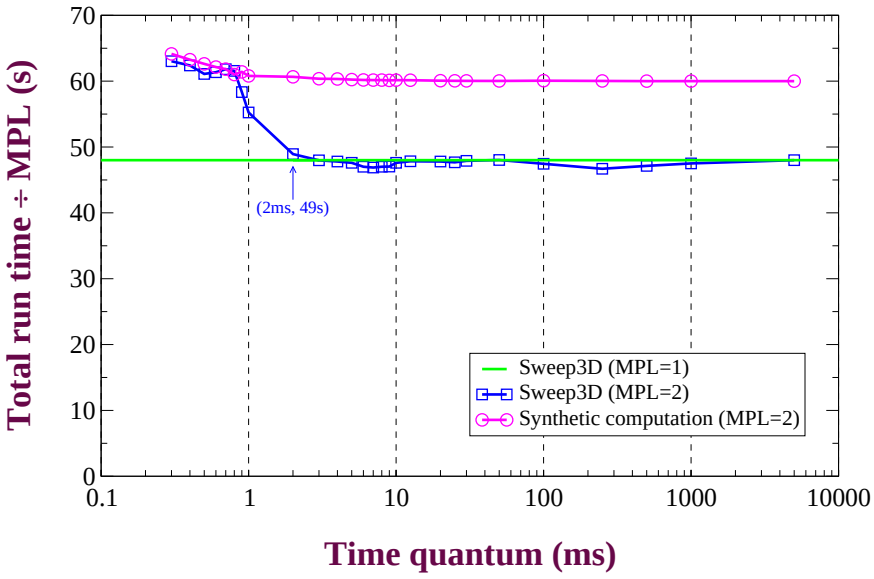


Figure 4.13: Effect of time quantum with MPL 2 on 32 nodes (Crescendo)

The application we use for our experiments in this section is SWEEP3D, which is described in Section 1.4. In tests that involve a multiprogramming level (MPL) of more than one, we launch all the jobs at the same instant—even though this may not be a realistic scenario—to further stress the scheduler.

4.3.3.1 Effect of Time Quantum

As a first gang-scheduling experiment, we analyze the range of usable timeslice values to better understand the performance limits of STORM's coordinated scheduling capabilities. Figure 4.13 shows the average run time of the jobs for various timeslice values, from $300\mu s$ to $8s$, running on the Crescendo cluster with 32 nodes/64 PEs (points represent measured data). The shortest timeslice that the NM can handle in Crescendo is $300\mu s$, below which it cannot process the incoming strobe messages at the rate they arrive. Even more significant is the fact that with a timeslice as small as $2ms$, STORM can run multiple instances of a parallel application with little performance degradation compared to a single instance of the application.[7] This timeslice is an order of magnitude shorter than the default local Linux scheduler's quanta, and multiple orders of magnitude better than the shortest time quanta that conventional gang schedulers can handle with no performance penalties [42]. This allows for good system responsiveness and usage of the parallel system for interactive jobs. Furthermore, a short quantum allows the implementation of advanced scheduling algorithms that can benefit greatly from short time quanta, such as BCS, ICS, and periodic boost. Because STORM can handle small time quanta with no performance penalty, we chose the value of $50ms$ for the next sets of experiments, which provides a fairly responsive system.

[7] This result is also influenced by the poor memory locality of SWEEP3D so running multiple processes on the same processor does not pollute their working sets.

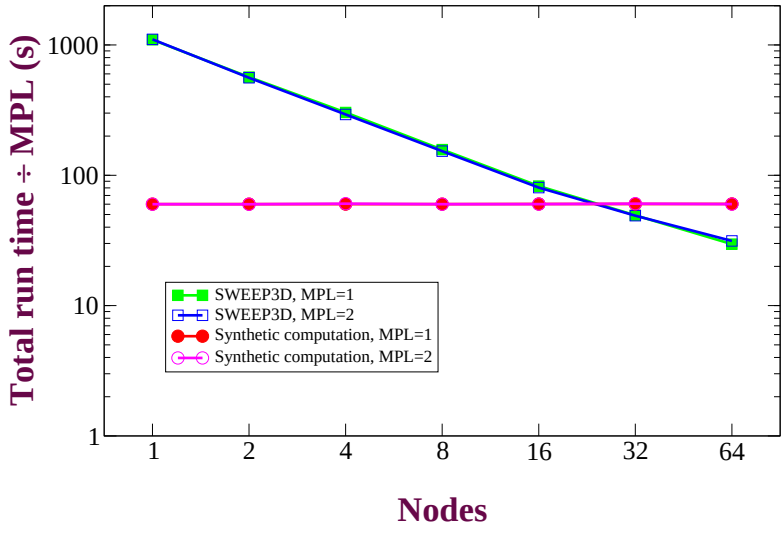


Figure 4.14: Context switch scalability on Crescendo with MPLs 1 & 2

4.3.3.2 Node Scalability

An important metric of a resource manager is the scalability with the number of nodes it manages. To test this, we measured the effect on application run time when running on an increasing number of nodes.

Figure 4.14 shows the results for running the programs on varying number of nodes in the range 1–64 for MPL values of 1 and 2 using a timeslice of $50ms$ (Results for MPL 2 are normalized by dividing the total runtime of all jobs by 2). The graph shows no visible increase in either the application run time or overhead with the increase in the number of PEs.

4.3.3.3 Effect of MPL

The overhead incurred by a context-switcth operation can have a limiting effect on the scheduler's multiprogramming ability. Context switches can cause performance degradation due to loss of cache state, synchronization difficulties across nodes, and the need to change the communication context gracefully, including the handling of in-transit messages[8] [24]. The context-switch operation in STORM is rather rudimentary and involves very little computation in order to determine the next process to run, suspend the current process, and resume the next one. This is in reality less overhead than is required by most UNIX schedulers for performing a local context switch [109]. To evaluate the overhead of the context switch operation in STORM, we measured the effect of multiprogramming on the SWEEP3D parallel application. Figure 4.15 shows the results of running 1, 2, 4, or 8 jobs together, with a timeslice of $50ms$, and compared to linear slowdown. All jobs were launched concurrently and run on all of Crescendo's 64 PEs. We can

[8]Context switches can improve performance in some cases. Some interconnects allow for the background processing of pending communication while the main CPU is busy with another computation, possibly of another process altogether. This overlap of computation and communication can thus lead to benefi t from context switches and yield a super-linear speedup when multiprogramming parallel jobs [41, 42].

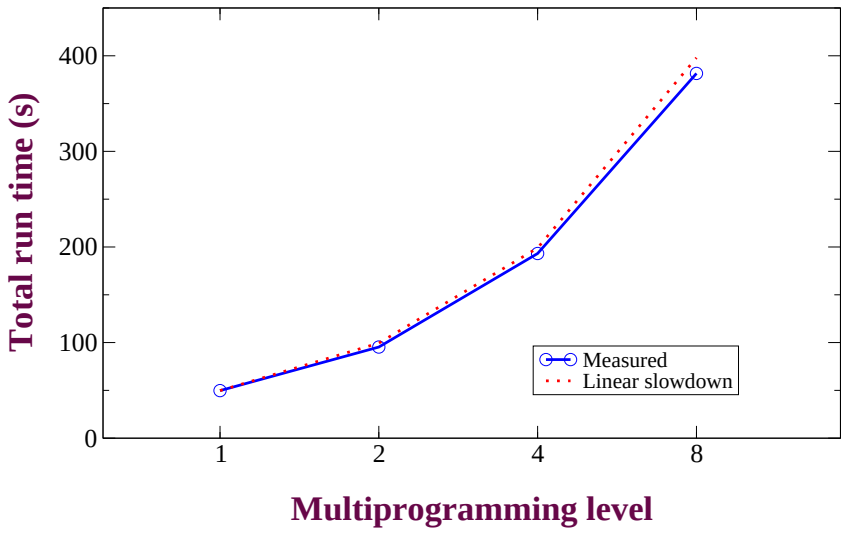


Figure 4.15: Effect of multiprogramming level on SWEEP3D run time (64 PEs)

see no degradation in performance compared to a linear slowdown, and in fact we can observe a slight performance gain, due to the overlapping of computation and communication. This effect cannot be extended further beyond an MPL of 8 for this configuration, due to the so-called “memory wall” - more instances of SWEEP3D would exhaust the physical memory of the machine and lead to paging and swapping.

4.4 Related Work

Although powerful hardware solutions for high-performance computing are already available, the largest challenge in making large-scale clusters usable lies in the system software. This chapter focuses on increasing the scalability and performance of job launching and process scheduling. We now examine previous attempts at improving these two RM functions.

4.4.1 Job Launching

Many run-time environments, such as the Portable Batch System (PBS) [51], distribute executable files to all nodes via a globally mounted filesystem, typically NFS [100]. This design, in which potentially many clients are simultaneously accessing a single file on a single server, is inherently non-scalable. Even worse, file servers are frequently unable to handle extreme loads gracefully; heavy loads lead to increased delays, which induce network timeouts, which cause read failures and aborted application launches. A workaround commonly taken by cluster administrators is to employ a simple shell script that iteratively starts processes on each node of the cluster. While this approach reduces contention on the file server, it still has severe performance and scalability limitations on clusters with more than several tens or hundreds of nodes. In contrast, by implementing the STORM mechanisms in terms of a tree-based multicast, STORM overhead

grows logarithmically, not linearly, with the number of nodes.

The ParPar cluster environment [59] addresses the problem of the distribution of control messages from a management node to a set of clients. ParPar utilizes a special-purpose multicast protocol, *reliable datagram multicast* (RDGM), which broadcasts UDP datagrams on the network and adds selective multicast and reliability. Each datagram is prepended by a bit string that identifies the set of destinations, and each node in the destination set sends an acknowledgment to the management node after the successful delivery of the broadcast datagram. By using RDGM, a job can be launched in a few tens of seconds on a cluster with 16 nodes and with relatively good scalability. Nevertheless, this is still significantly slower (albeit on a slower network) than the launch time of a sequential job on an individual workstation and enough to annoy users who are waiting for an interactive job to launch. One of the goals for STORM was to ensure that launch times on large clusters are no slower than those on a single node. As shown in Section 4.3, an efficient implementation of the STORM mechanisms leads to measured launch times of under $110ms$ for 256 processors and predicted launch times of under $135ms$ for 16,384 processors.

GLUnix [47] is a module of operating system middleware for clusters of workstations, designed to provide transparent remote execution, load balancing, coscheduling of parallel jobs, and fault-detection. The creators of GLUnix note that when forking a parallel job, the overhead in the master node increases by a small but linear amount: an average of $220\mu s$ per client node. Extrapolating, this implies just over 50 seconds to launch a job on 4,096 nodes (16,384 processors).

When GLUnix launches a job, remote execution messages are sent from the management node to all the dæmons that will run the job. Each of these dæmons generates a reply message, indicating success or failure. When performing remote execution to more than 32 nodes over switched Ethernet, the replies from earlier dæmons in the communication schedule collide with the remote execution requests sent to later dæmons [47]. This causes a substantial performance degradation. STORM on the other hand can benefit from QsNet's network conditionals, which utilize a combining tree to reduce network contention and improve performance and scalability [83].

Scalability problems are already evident in ASCI-scale machines (with thousands of nodes). The Computational Plant (Cplant) project [11, 97] at Sandia National Laboratories utilizes several large-scale commodity-based clusters. In order to enhance scalability, Cplant uses a high-performance interconnect, Myrinet [8, 12], and a custom, lightweight communication protocol based on Portals [11]. When Cplant's RM system launches a job, it first identifies a group of active worker nodes, organizes them into a logical tree structure, and then fans out the executable to the nodes. Experimental results show that a large, parallel application can be launched on a 1010-node cluster in about 20 seconds [11]. Cplant is the closest project in spirit to STORM, in that it identifies poor RM performance as a problem worth studying and approaches the problem by replacing a traditionally non-scalable algorithm with a scalable one. Given the same platform, the STORM mechanisms would likely be implemented fairly similarly to Cplant's. However, on a platform such as QsNet, which boasts hardware collectives, STORM is able to exploit the underlying

hardware to improve job-launching performance by a hundredfold.

BProc [52], the Beowulf Distributed Process Space, takes a fairly different approach to job launching from STORM and the other works described above. Rather than copy a binary file from a disk on the master to a disk on each of the slaves and then launching the file from disk, BProc replicates a *running* process into each slave's memory—the equivalent of Unix's `fork()` and `exec()` plus an efficient migration step. The advantage of BProc's approach is that no filesystem activity is required to launch a parallel application once it is loaded into memory on the master. Even though STORM utilizes a RAM-disk based filesystem, the extra costs of reading and writing to the filesystem add a significant amount of overhead relative to BProc's remote process spawning. STORM's advantage over BProc is that the same functions STORM uses to transmit executable files can also be used to transmit data files. BProc has no equivalent mechanism, although a system could certainly benefit from BProc's single-system-image features and from STORM's underlying communication protocols together.

Table 4.4 shows a sampling of job-launch times found in the literature; Table 4.5 presents the same data extrapolated out to 4,096 nodes (twice the size of ASCI Q [1]); and Figure 4.16 graphs both the measured and extrapolated (to 16,384 nodes) data. Although the different cluster types and sizes make the comparison imprecise, the aforementioned tables and figures at least give a general indication that STORM does, in fact, provide a significant performance improvement over previous works.

Table 4.4: A selection of job-launch times found in the literature

Resource manager	Job-launch time	
`rsh`	90	seconds to launch a minimal job on 95 nodes [47]
RMS	5.9	seconds to launch a 12 MB job on 64 nodes [44]
GLUnix	1.3	seconds to launch a minimal job on 95 nodes [47]
Cplant	20	seconds to launch a 12 MB job on 1,010 nodes [11]
BProc	2.7	seconds to launch a 12 MB job on 100 nodes [52]
STORM	0.11	seconds to launch a 12 MB job on 64 nodes

Table 4.5: Extrapolated job-launch times

Resource manager	Job-launch time extrapolated to 4,096 nodes		
`rsh`	3827.10	seconds for 0 MB	$(t = 0.934n + 1.266)$
RMS	317.67	seconds for 12 MB	$(t = 0.077n + 1.092)$
GLUnix	49.38	seconds for 0 MB	$(t = 0.012n + 0.228)$
Cplant	22.73	seconds for 12 MB	$(t = 1.379 \lg n + 6.177)$
BProc	4.88	seconds for 12 MB	$(t = 0.413 \lg n - 0.084)$
STORM	0.11	seconds for 12 MB	(see Section 4.3.2.2)

To clarify the performance improvement provided by STORM, Figure 4.17 renormalizes the extrapolated Cplant and BProc data to the extrapolated STORM performance, which is defined as 1.0. Cplant and BProc are the two pieces of related work that, like STORM, scale logarithmically, not linearly, in the number of nodes. The figure shows a decrease in the Cplant and BProc slowdown at 4,096 nodes. This is

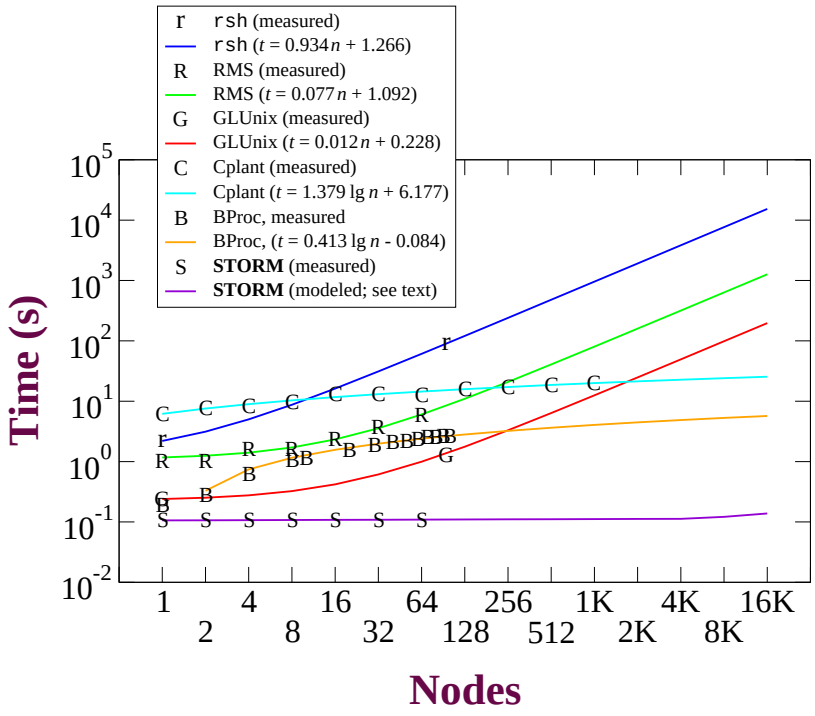


Figure 4.16: Measured and predicted performance of various job launchers

an artifact of the conservative performance model we used for STORM in Section 4.3.2.2, which indicates decreased network bandwidth as cluster sizes—and hence, cable lengths—increase. We extrapolated the performance of Cplant, BProc, and all of the other job-launchers presented in Table 4.4, Table 4.5, and Figure 4.16 under the unrealistic assumption that network performance scales indefinitely. Nevertheless, even though STORM's model is more conservative than the others, the crossover point between BProc and STORM is expected to be on a system of approximately 1 billion nodes, and the crossover point between Cplant and STORM is expected to be on a system containing approximately 17 billion nodes.

4.4.2 Process Scheduling

Many recent research results show that good job scheduling algorithms can substantially improve scalability, responsiveness, resource utilization, and usability of large-scale machines [2, 31]. Unfortunately, the body of work developed in the last few years has not yet led to many practical implementations of such algorithms on production machines. Arguably, one of the main reasons for this is the lack of flexible and efficient run-time systems that can support the implementation and evaluation of new scheduling algorithms, in order to convincingly demonstrate their superiority over today's entrenched, space-shared schedulers. Its flexibility positions STORM as a suitable vessel for *in vivo* experimentation with alternate scheduling algorithms, so researchers can determine the best way to manage cluster resources.

As far as traditional gang-schedulers are concerned, the SCore-D scheduler [55, 56] is one of the fastest.

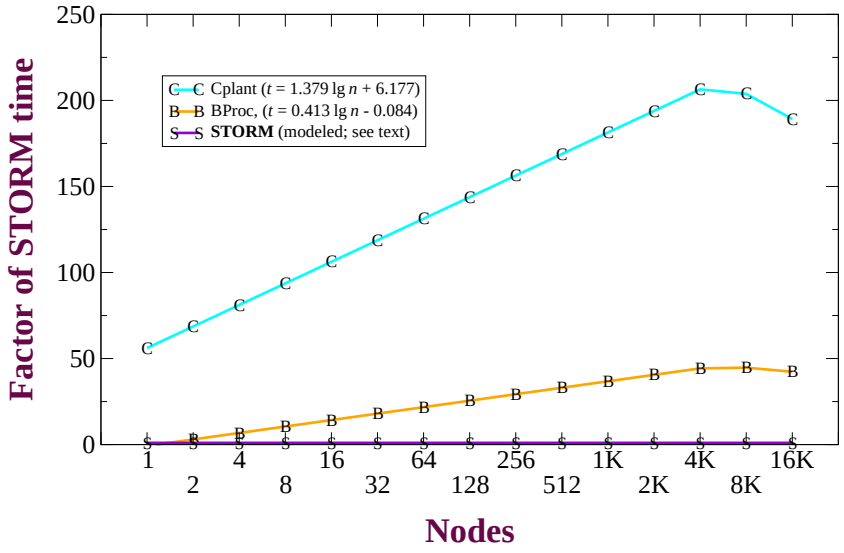


Figure 4.17: Normalized performance of Cplant, BProc, and STORM

By utilizing the messaging layer, PM [112], SCore-D is able to force communication into a quiescent state, save the entire global state of the computation, and restore another application's global state with only $\sim$2% overhead when using a $100ms$ time quantum. While this is admirable performance, STORM is able to do significantly better. Because the STORM mechanisms can be written to exploit QsNet's process-to-process communication (versus PM/Myrinet's node-to-node communication), STORM does not need to force the network into a quiescent state before freezing one application and thawing another [42]. As a result, STORM can gang-schedule applications with no noticeable overhead when using quanta as small as $2ms$ on Wolverine (on Accelerando, where the network is somewhat slower and the processors are faster, higher time quanta may be required to hide the context switch overhead).

5 Communication Library

5.1 Background

As described in the introduction, system software for large-scale parallel machines is undergoing an increasing rise in complexity and requirements. System software consists of various components, including communication libraries, resource management (the software infrastructure in charge of resource allocation and accounting), services for parallel file system, and fault tolerance. The current state of the art is to design these components separately, in order to have a modular design and allow different developers to work concurrently while limiting cross-dependencies. Some of these components have many elements in common, such as communication mechanisms, that are implemented and re-implemented several times over. In some cases the lack of a single source of system services is also detrimental to performance: most parallel systems cannot guarantee quality of service (QoS) for user-level traffic and system-level traffic in the same interconnection network. Low-priority, best-effort traffic generated by the parallel file system can interfere with higher-priority, latency-sensitive traffic generate at user level. As another example, system dæmons that perform resource management can introduce computational "holes" of several hundreds of ms that can severely impact fine-grained scientific simulations, since they are not coordinated with the communication library's activities [88]. In this chapter we promote the idea of using the same basic mechanisms that we use for resource management and scheduling purposes to simplify the communication layer, based on the Buffered Coscheduling (BCS) model.

BCS [25, 35, 80, 81] is a new design methodology proposed by Petrini et al. for the system software that attempts to tackle both problems: the complexity of a large-scale parallel machine and the redundancy of its software components. The vision behind BCS is that both size and complexity of the system software can be substantially reduced by using a common and coordinated view of the system. BCS tries to globally organize all the activities of such machines at a fine granularity, (a few hundreds of μs). In a sense, BCS represents an effort to implement a SIMD global OS at a granularity coarser than the single instruction, and yet fine enough so as not to harm application performance. Both computation and communication are globally scheduled at regular intervals, and the scheduling decisions are taken after exchanging all the required information. The separate operating systems of each node are coalesced into a single system view,

without incurring any significant performance penalty.

We argue that the core primitives laid out in Chapter 3 are on the one hand general enough to cover many of the requirements of system software, including the application communication layer, and yet on the other hand, close enough to the hardware layer to exploit the highest level of performance. Chapter 4 demonstrates that it is possible to implement a scalable resource management system that is orders of magnitude faster than existing production-level software, by using these core primitives.In this chapter we extend and generalize our research to another aspect of system software, the communication library. We have chosen to implement a variant of the popular MPI library, called BCS-MPI. BCS-MPI is designed following the BCS methodology. It is hierarchically built on top of our core primitives and its scheduling decisions are globally coordinated.

Related Work The LogP model developed by Culler et al. [20] uses four parameters, (computing bandwidth, communication bandwidth, communication delay and the efficiency of coupling communication and computation) to model asynchronous message passing. This model encourages the programmer to optimize single point-to-point communications while our approach tries to optimize the global state of the machine in order to reduce the non-determinism. The Bulk-Synchronous Parallel (BSP) model was introduced by Valiant in [115] (described in Section 3.3). This model constitutes a first attempt to optimize the entire application performance rather than latency or bandwidth for single point-to-point messages. Buffered Coscheduling (BCS), as proposed by Petrini et al. in [26] is the methodology for time-sharing communicating processes on which this study is based.

The advent of programmable network interface cards enables the offloading of a considerable part of the communication protocol to the NIC [12, 13, 101]. However, most of the previous work in this area tends to be focused on optimizing latency and bandwidth performance. To the best of our knowledge, BCS-MPI is the first attempt to globally synchronize all the system activities in order to schedule communications.

5.2 The BCS-MPI Model

The main research trend in the design of communication libraries over the past decade has been to minimize the point-to-point latency by removing kernel overhead [6, 18, 79, 93, 102, 112, 118]. Libraries such as Active Messages or Fast Myrinet attempt to move the data communication handling into the user level . BCS-MPI follows a different path: rather than optimizing the single point-to-point communication in isolation, it tries to optimize the entire communication pattern. Communication is scheduled globally by dividing time into short slices, and using a distributed algorithm to schedule the point-to-point communication that will occur at each time slice. Communication is scheduled only at the beginning of a time slice and performed at kernel level [15]. The shortest latency that a message will experience will be at least one time slice, which is in the order of few hundreds of μs with current technology. On the other hand, we

gain better ordering and determinism of the communication behavior, which can have significant benefits. For example, the fact that the communication state of all processes is known at the beginning of every time slice facilitates the implementation of checkpointing and debugging mechanisms.

The primary contribution here is in demonstrating that a constrained communication library such as BCS-MPI provides approximately the same performance of a production-level version of MPI on a large set of scientific applications, but with a much simpler software design. In fact, BCS-MPI is so small that it runs almost entirely on the network interface processor, and its activity is completely overlapped with the computation of the processing node. We also discuss the importance of non-blocking communication compared to blocking communication and how minor changes in the communication pattern (e.g. replacing blocking calls with non-blocking counterparts) can substantially improve the application performance.

BCS-MPI's simplified design relies on the primitives described in Chapter 3, which in turn rely on advanced networks' features. In some of these networks (Gigabit Ethernet, Myrinet and Infiniband) these primitives need to be emulated through a thin software layer, while in other networks there is a one-to-one mapping with native hardware mechanisms. We argue that in both cases – with or without hardware support – these primitives represent an ideal abstract machine that can export the raw performance of the network, while still providing a general-purpose basis for designing simple and efficient system software.

5.3 BCS-MPI Design and Implementation

To evaluate and validate the framework proposed in the previous section, we developed a fully functional version of BCS-MPI and incorporated it with STORM. For quick prototyping and portability, BCS-MPI is initially implemented as a user-level communication library, and some typical kernel level functionalities such as process scheduling are implemented with STORM's dæmons. This user-level implementation is expected to be slower than a kernel-level one, albeit at increased flexibility and ease-of-use. One of our future goals is migrating the performance-sensitive parts of STORM and BCS-MPI to the kernel.

The communication library is hierarchically designed on top of a small set higher-level primitives (the BCS API [35]), which are in turn based on the three network primitives (Figure 5.2(a)). This approach greatly simplifies the design and implementation of BCS-MPI in terms of complexity, maintainability and extensibility. BCS-MPI is built on top of the BCS API by simply mapping MPI calls to BCS calls[1].

5.3.1 BCS-MPI Design

Unlike typical implementations of MPI, BCS-MPI globally schedules the system activities on all the nodes: a synchronization broadcast message or *global strobe* – implemented with XFER-AND-SIGNAL – is sent to all nodes at regular intervals or *timeslices*. Thus, all the system activities are tightly coupled since they occur concurrently on all the nodes. Both computation and communication are scheduled and the

[1]The details of the BCS API are shown in [35].

communication requests generated by each application process are buffered. At the beginning of every time slice a partial exchange of communication requests – implemented with XFER-AND-SIGNAL and TEST-EVENT – provides information to schedule the communication requests issued during the previous time slice. Subsequently, all the scheduled communication operations are also performed using the primitives XFER-AND-SIGNAL and TEST-EVENT.

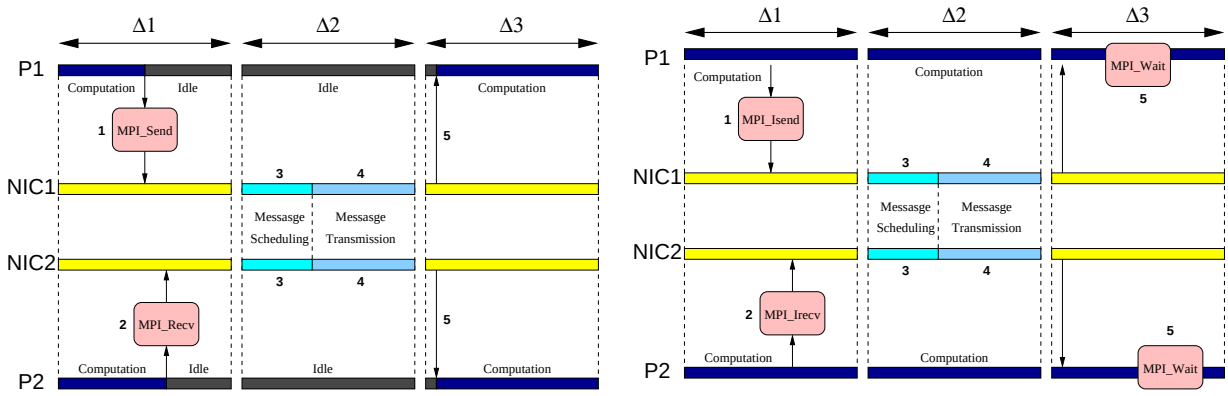
The BCS-MPI communication protocol is implemented almost entirely on the NIC. This enables BCS-MPI to overlap the communication with the computation executed on the host CPUs. The application processes interact directly with threads running on the NIC. When an application process invokes a communication primitive, it posts a descriptor in a region of NIC memory that is accessible to a NIC thread. Such a descriptor includes all the communication parameters that are required to complete the operation. The actual communication will be performed by a set of cooperating threads running on the NICs involved in the communication protocol. In the QsNet network these threads can directly read/write from/to the application process memory space so that no copies to intermediate buffers are needed. The communication protocol is divided into micro-phases within every time slice and its progress is also globally synchronized, as described in Section 5.3.3. To better explain how BCS-MPI communication primitives work, two possible scenarios for blocking and non-blocking MPI point-to-point primitives are described below.

5.3.1.1 Blocking Send/Receive Scenario

In this scenario, a process P_1 sends a message to process P_2 using MPI_Send and process P_2 receives a message from P_1 using MPI_Recv (see Figure 5.1(a)):

1. P_1 posts a send descriptor to the NIC and blocks.
2. P_2 posts a receive descriptor to the NIC and blocks.
3. The transmission of data from P_1 to P_2 is scheduled since both processes are ready (all the pending communication operations posted before time slice i are scheduled, if possible). If the message cannot be transmitted in a single time slice, then it is chunked and scheduled over multiple time slices.
4. The communication is performed (all the scheduled operations are performed before the end of time slice $i + 1$).
5. P_1 and P_2 are restarted at the beginning of time slice $i + 2$.
6. P_1 and P_2 resume computation.

Note that the delay per blocking primitive is 1.5 time slices on average. However, this performance penalty can be alleviated by using non-blocking communication (see Section 5.4.3) or by scheduling a different parallel job in time slice i+1 .



(a) Blocking MPI_Send/MPI_Recv (b) Non-blocking MPI_Isend/MPI_Irecv

Figure 5.1: Blocking and Non-Blocking Scenarios

5.3.1.2 Non-Blocking Send/Receive Scenario

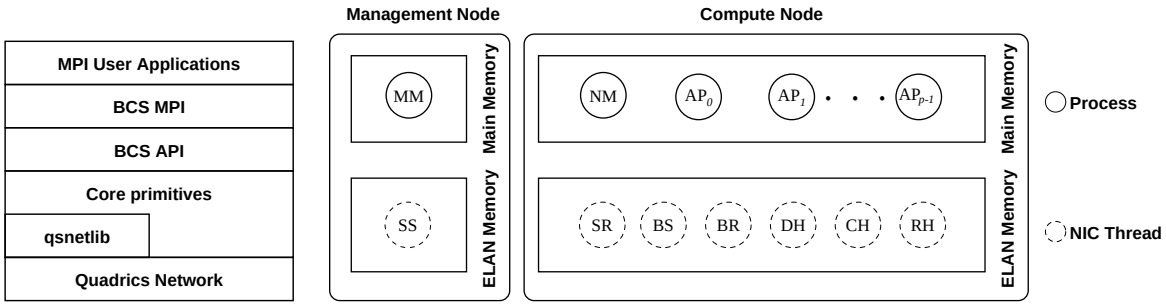
In this scenario, a process P_1 sends a message to process P_2 using MPI_Isend and process P_2 receives a message from P_1 MPI_Irecv (see Figure 5.1(b)):

1. P_1 posts a send descriptor to the NIC.
2. P_2 posts a receive descriptor to the NIC.
3. The transmission of data from P_1 to P_2 is scheduled since both processes are ready (all the pending communication operations posted before time slice i are scheduled if possible).
4. The communication is performed (all the scheduled operations are performed before the end of time slice $i + 1$).
5. P_1 and P_2 verify that the communication has been performed and continue their computation.

Here the communication is completely overlapped with the computation with no performance penalty.

5.3.2 Processes and Threads

With the current user-level implementation, the BCS-MPI runtime system consists of STORM's dæmons and a set of threads running on the Elan NIC. The processes and NIC threads that constitute the BCS-MPI runtime system are shown in Figure 5.2(b). As described in Chapter 4, the Machine Manager (MM), runs on the management node. This dæmon coordinates the use of system resources issuing regular heartbeats and controls the execution of parallel jobs. The Strobe Sender (SS) is a NIC thread forked by the MM that implements the global synchronization protocol as described in Section 5.3.3. The Node Manager (NM) dæmons run on every compute node. This process executes all the commands issued by the MM, manages the local resources, and schedules the execution of the local processes. The Strobe Receiver (SR), the



(a) Library Hierarchy (b) Processes and threads

Figure 5.2: BCS-MPI architecture

Buffer Sender (BS), the Buffer Receiver (BR), the DMA Helper (DH), the Collective Helper (CH) and the Reduce Helper (RH) are all NIC threads forked by the NM in each compute node. The SR is the counterpart of the SS in the compute nodes and coordinates the execution of all the local threads. The BS and the BR handle the descriptors posted by the application processes whenever a communication primitive is invoked, and schedule the point-to-point and collective communication operations. The DH carries out the actual data transmission for the point-to-point operations. Finally, the CH and the RH perform the barrier and broadcast operations, and the reduce operations, respectively.

5.3.3 Global Synchronization Protocol

The BCS-MPI runtime system globally schedules all the computation, communication and synchronization activities of the MPI jobs at regular intervals. Each time slice is divided into two main phases and several micro-phases, as shown in Figure 5.3. The two phases are the *global message scheduling* and the *message transmission*. The global message scheduling phase schedules all the descriptors posted to the NIC during the previous time slice. A partial exchange of control information is performed during the *descriptor exchange micro-phase* (DEM). The point-to-point and collective communication operations are scheduled in the *message scheduling micro-phase* (MSM) using the information gathered during the previous micro-phase. The *message transmission phase* performs point-to-point operations, barrier and broadcast collectives, and the reduce operations, respectively, during its three micro-phases.

To execute the global synchronization mechanism, the SS and the SR threads synchronize at the beginning of every micro-phase with a micro-strobe (using XFER-AND-SIGNAL). The SS checks whether all the nodes have completed the current micro-phase (using COMPARE-AND-WRITE) and, if so, sends a micro-strobe to all the SRs. The SR running on every node subsequently wakes up the relevant local NIC thread(s) for the new micro-phase. The BS and the BR run during the descriptor exchange micro-phase to process the descriptors and during the message scheduling micro-phase to schedule the messages. The DH, the CH and the RH run during the point-to-point, the broadcast and barrier, and the reduce micro-phases, respectively, to perform all the operations scheduled for execution in the global message scheduling phase.

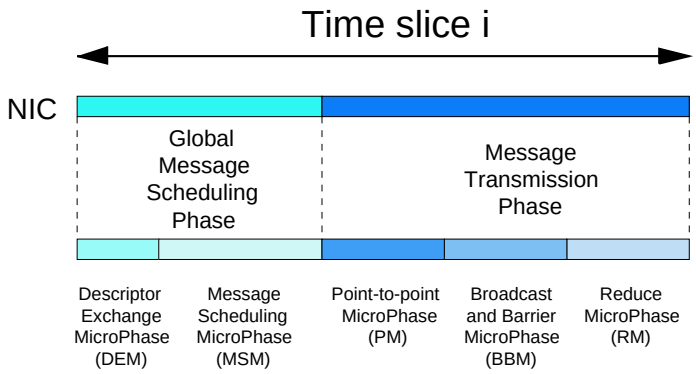


Figure 5.3: Global synchronization protocol

5.3.4 Point-to-point

Every time a user process invokes a point-to-point MPI primitive, it initializes a descriptor in a region of memory accessible to the NIC threads which will initiate the operation on its behalf (Figure 5.1). All the descriptors for either blocking or non-blocking send-operations are posted to the BS thread while all the descriptors for either blocking or non-blocking receive-operations are posted to the BR thread. Each application process involved in the communication protocol is suspended only if the invoked primitive is blocking. All the descriptors posted during time slice i-1 will be scheduled for execution, if possible, at time slice i as follows (see also Figure 5.4).

Descriptor Exchange Microphase The BS delivers each send descriptor posted in time slice i-1 to the BR running on the destination node.

Message Scheduling Microphase The BR matches the remote send descriptor list against the local receive descriptor list. For each matching pair, the BR builds a descriptor with the information required to complete the data transfer, and schedules the operation for execution. If the message is too large and cannot be scheduled within a single time slice, the BR splits it into smaller chunks. The first chunk of the message is scheduled during the current time slice and the remaining chunks in the following time slices. In the current implementation, these two phases take approximately $125\mu s$.

Point-to-point Microphase For each matching descriptor created in the previous micro-phase by the BR, the DH performs the real data transmission. Note that no intervention from the two application processes involved is required.

5.3.5 Collective Communication

Every time a user process calls a collective MPI function such as MPI_Barrier, MPI_Broadcast, MPI_Reduce or MPI_Allreduce, BCS-MPI posts a descriptor to the BR thread, which in turn initiates the operation on

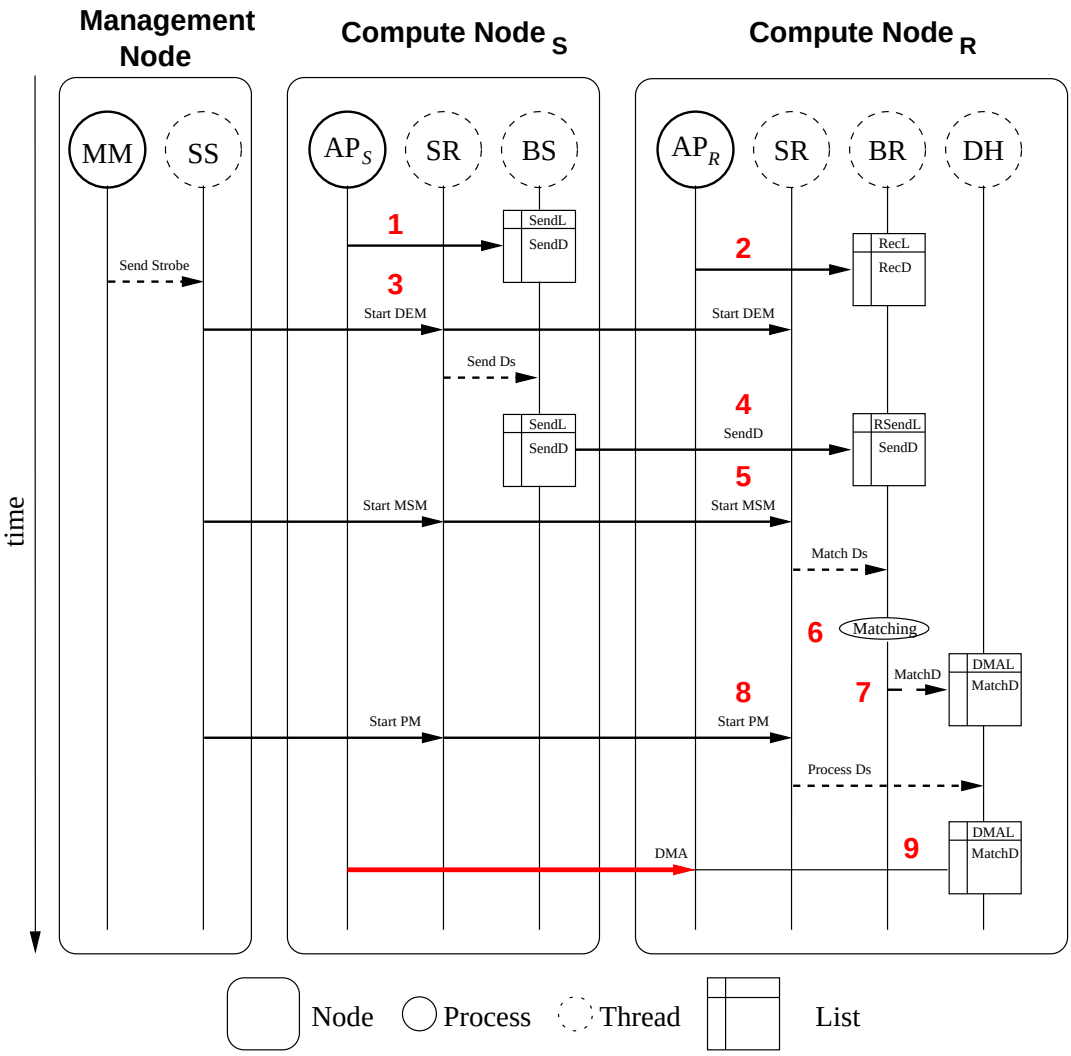


Figure 5.4: Send/Receive with BCS-MPI

(1) The sender process posts a descriptor to the BS (2) The receiver process posts a descriptor to the BR (3) SS sends a micro-strobe to signal all the SRs the beginning of the Descriptor Exchange Micro-phase (DEM) (4) BS sends the descriptor to the BR running on the receiving end (5) SS sends a micro-strobe to signal the beginning of the Message Scheduling Micro-phase (MSM) (6) BR matches the remote send and the local receive descriptors (7) SS sends a micro-strobe to signal the beginning of the Point-to-point Micro-phase (PM) (8) BR schedules the operation for execution (9) DH performs the get (one-sided communication).

its behalf, and blocks. The BR preprocesses all the collective descriptors. If all the local processes of a parallel job have invoked the collective primitive, a local flag for that job is set. Following that, all the collective descriptors except for those corresponding to the job master processes, are discarded. All the descriptors posted during time slice i-1 will be scheduled, if possible, in time slice i as follows:

Message Scheduling Microphase For each collective descriptor corresponding to a job master process, the BR tests if all the application processes of that MPI parallel job had invoked the collective primitive in all nodes. In order to accomplish this, the BR issues a query broadcast (using COMPARE-AND-WRITE) message that checks the flag for that job. If the flag is set on all nodes, the collective operation is scheduled for execution.

Broadcast and Barrier/Reduce Microphase The scheduled broadcast operations are performed by the CH broadcasting the data to all the processes of the MPI parallel job. The barrier operation is a special case of a broadcast operation with no data. The scheduled reduce operations are carried out by the RH on the NIC by using a binomial tree to gather the partial reduce results. The Elan NIC has no floating-point unit. Hence, an IEEE compliant library for binary floating-point arithmetic has been used to compute the reduce in the NIC (SoftFloat [128]). Since most applications reduce over a very small number of elements [73, 116], computing the reduce in the NIC is faster than sending the data through the PCI bus to perform the operation in the host [73].

Figure 5.5 illustrates the execution of a broadcast operation. The MPI program in this example is composed of four processes running on two different nodes.

5.4 Experimental Results

In this section we analyze the results of running BCS-MPI synthetic applications, as well as SAGE, SWEEP3D, and several NAS-benchmark applications. All the results were obtained on Crescendo with a $500\mu s$ timeslice.

5.4.1 Synthetic Benchmarks

Many scientific codes display a bulk-synchronous behavior [115] and can be characterized by a nearest-neighbor communication stencil, optionally followed by a global synchronization operation such as barrier, broadcast or reduce [53, 61]. Therefore, we designed two synthetic benchmarks that represent this pattern (see Section 1.4) to compare our experimental BCS-MPI with the production-level Quadrics MPI.

In the first synthetic benchmark, every process computes for a parametric amount of time and barrier-synchronizes with all the other processes in a loop. The slowdown of BCS-MPI in relation to Quadrics MPI for different computational granularities is shown in Figure 5.6(a) . As expected, the slowdown decreases as we increase the computational granularity since the effect of the delay introduced by the barrier

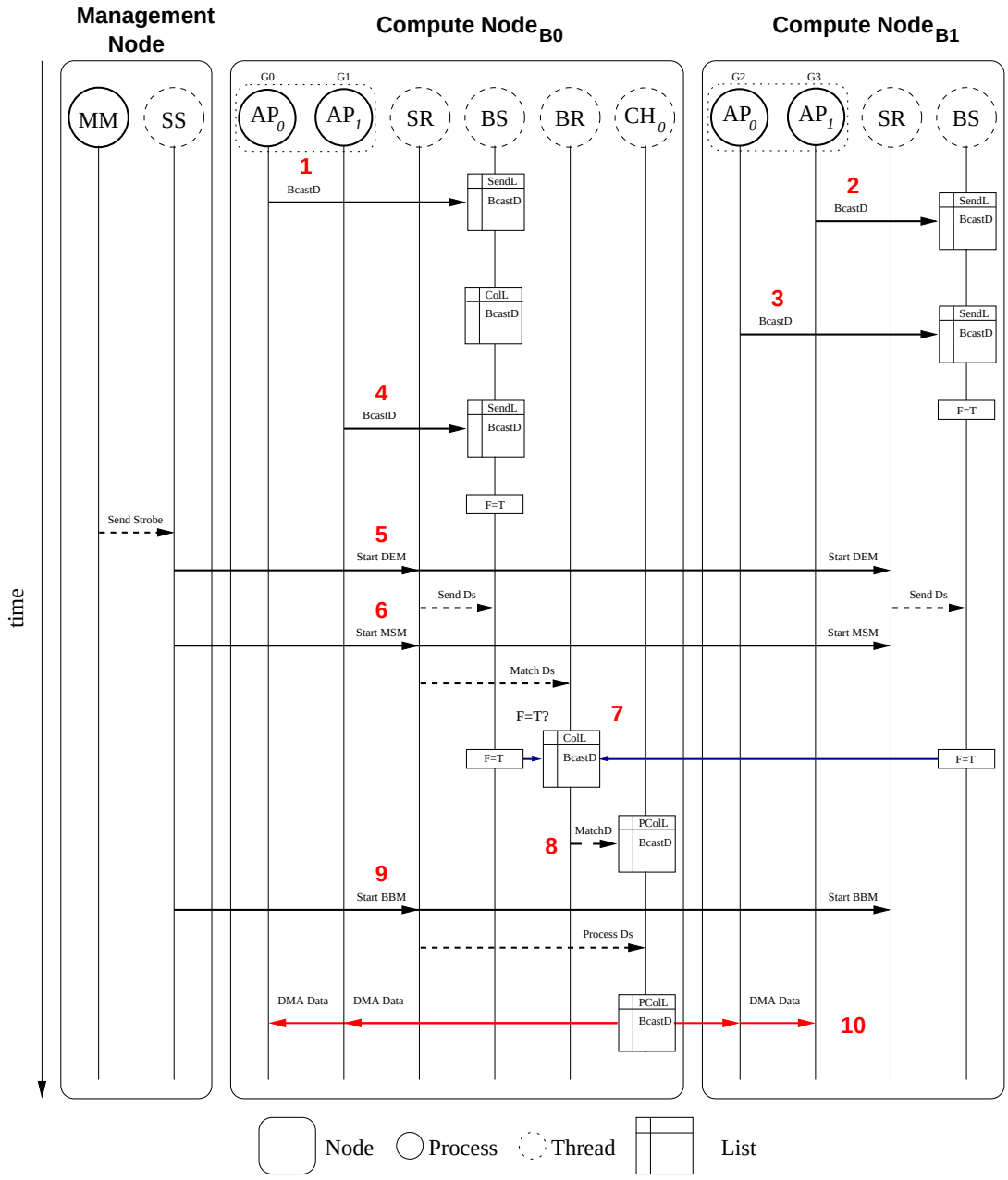
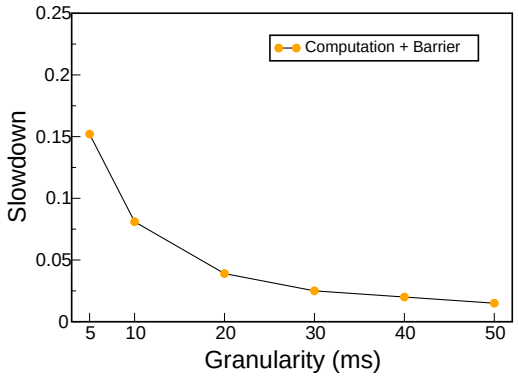
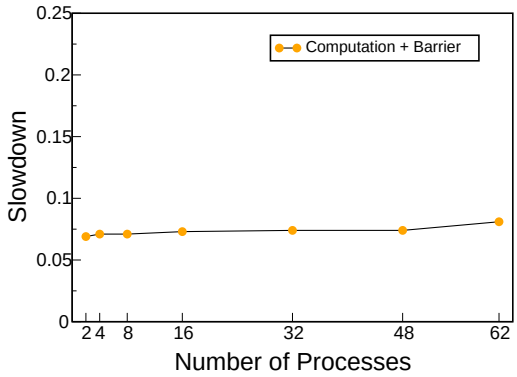


Figure 5.5: Broadcast with BCS-MPI

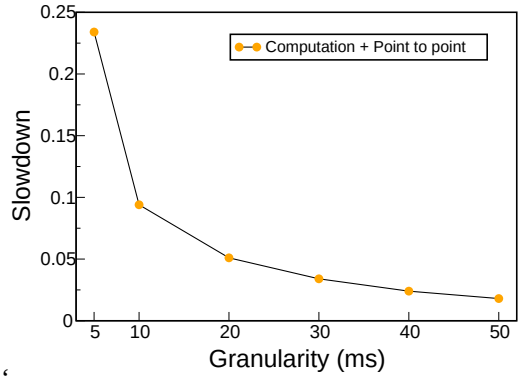
(1) Application Process (AP) G_0 posts a descriptor to the local BS. G_0 is the master process and its descriptor is copied to the Collective List (2) G_3 posts a descriptor to the local BS. The descriptor is processed and discarded (3) G_2 posts a descriptor to the local BS. The descriptor is processed: all the local processes have reached the barrier and Flag F is set to True. Descriptor is discarded (4) G_4 posts a descriptor to the local BS. The descriptor is processed: all the local processes have reached the barrier and Flag F is set to True. The descriptor is discarded (5) The SS sends a micro-strobe to signal all the SRs the beginning of the Descriptor Exchange Micro-phase (DEM) (6) SS sends a micro-strobe to signal the beginning of the Message Scheduling Micro-phase (MSM) (7) BR checks whether all the processes are ready (8) BR schedules the broadcast operation for execution (9) SS sends a micro-strobe to signal the beginning of the Broadcast and Barrier Micro-phase (BBM) (10) CH performs the broadcast.



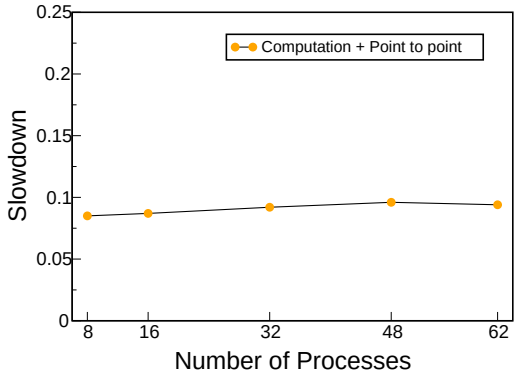
(a) Barrier synchronization: 62 processes



(b) Barrier synchronization: $10ms$ granularity



(c) Nearest-neighbor synchronization:
62 processes, 4 neighbors, 4KB messages



(d) Nearest-neighbor synchronization:
$10ms$ granularity, 4 neighbors, 4KB messages

Figure 5.6: BCS-MPI Synthetic Benchmarks

synchronization is amortized. The figure shows that the slowdown is less than 7.5% with a computation granularity of $10ms$ when we run this benchmark on the entire cluster. Figure 5.6(b) shows the slowdown of BCS-MPI versus Quadrics MPI as a function of the number of processes. In this case, the results indicate that BCS-MPI scales well for barrier synchronization operations, and it is almost insensitive to the number of processors.

In the second synthetic benchmark, every process computes for a parametric amount of time, exchanges a fixed number of non-blocking point-to-point messages with a set of neighbors, and waits for the completion of all the communication operations in a loop. The slowdown for different computational granularities is shown in Figure 5.6(c). Like the previous scenario, the slowdown decreases as the computational granularity increases, remaining below 8% for granularities larger than $10ms$. Finally, from Figure 5.6(d) we can observe that BCS-MPI scales well with point-to-point operations as well.

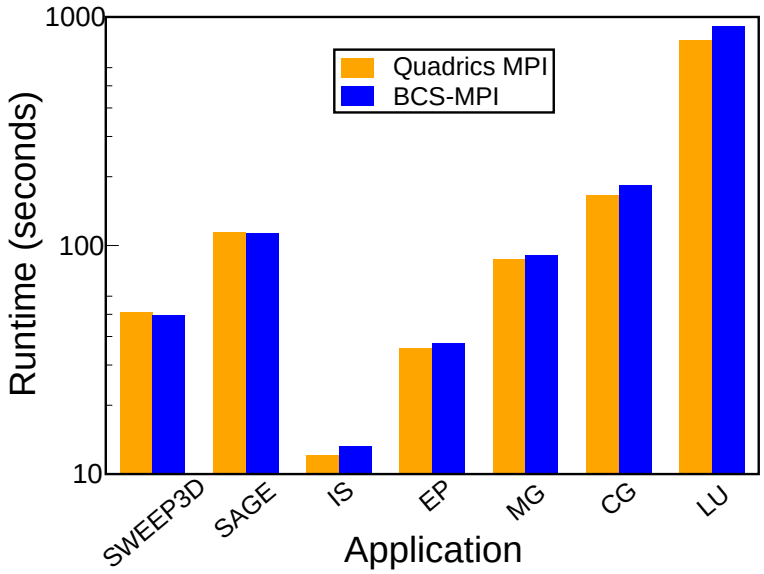


Figure 5.7: Benchmarks and Applications

5.4.2 NAS Benchmarks and Applications

In this section we use the NAS Parallel Benchmarks (NPB 2.4) [4, 123], SWEEP3D and SAGE (SAIC's Adaptive Grid Eulerian hydrocode) [61]. The NAS Parallel Benchmarks are a set of eight programs designed to help in evaluating the performance of parallel supercomputers. The suite, which is derived from computational fluid dynamics (CFD) applications, consists of five kernels and three applications. Since BCS-MPI does not support MPI groups yet, we were only able to use four kernels and one application: Integer Sort (IS), Embarrassingly Parallel (EP), Conjugate Gradient (CG), Multigrid (MG) and LU solver (LU). All programs are written in Fortran 77 (except for IS which is written in C) and use MPI for interprocessor communications. All the benchmarks were compiled for the *class C* workload.

SAGE and SWEEP3D (described in Section 1.4) are part of LANL's Crestone project, whose goal is the investigation of continuous adaptive Eulerian techniques to stockpile stewardship problems. SAGE is characterized by a nearest-neighbor communication pattern that uses non-blocking communication operations followed by a reduce operation at the end of each compute step. The *timing.input* data set was used in all the experiments. SWEEP3D is characterized by a fine granularity (each compute step takes $\approx 3.5ms$ on Crescendo) and a nearest-neighbor communication stencil with blocking send/receive operations.

For each of these applications we compare the runtime of BCS-MPI to that of Quadrics MPI, and analyze the results. The final runtime was computed as the average of five executions.

The application run times for both Quadrics MPI and BCS-MPI are shown in Figure 5.7. The slowdown of BCS-MPI in comparison to Quadrics MPI is computed in Table 5.1. All NPB benchmarks (except

Application	Slowdown
SAGE	-0.42%
SWEEP3D	-2.23%
IS	10.14%
EP	5.35%
MG	4.37%
CG	10.83%
LU	15.04%

Table 5.1: Application slowdown of BCS-MPI compared to Quadrics MPI

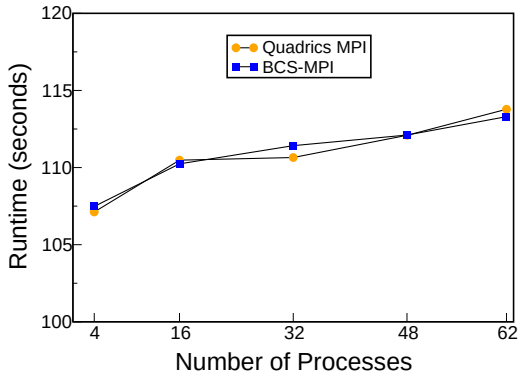
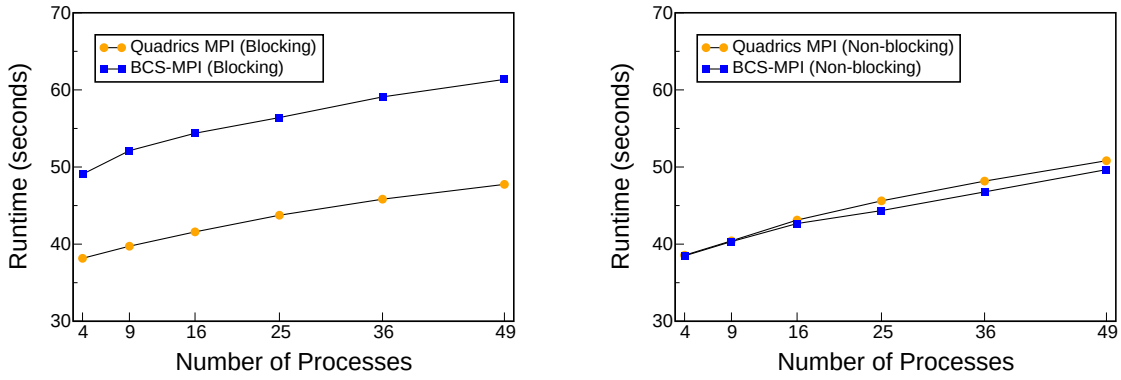


Figure 5.8: SAGE performance

LU) and SAGE perform reasonably well with BCS-MPI. The NPB programs are coarse-grained bulk-synchronous applications that show an expected moderate slowdown of up to 8%, as discussed in Section 5.4.1. However, three programs do not meet the expectations. IS takes approximately only $\approx 12s$ to run in this configuration and consequently pays a relatively high price for the overhead of initializing the BCS-MPI runtime system. CG and LU use several consecutive blocking calls inside a loop which introduce a considerable delay, since no overlap between computation and communication is possible for several time slices. This problem can be reduced by using non-blocking communication, as described in the context of SWEEP3D in Section 5.4.3 below.

SAGE is a medium-grained application and the non-blocking communications mitigate the performance penalty of the global synchronization operation performed at the end of each compute step. The slight performance improvement is obtained thanks to the negligible overhead of the non-blocking calls, that only initialize a communication descriptor. We repeated the SAGE measurements with various cluster sizes, and the results (shown in Figure 5.8) indicate that BCS-MPI scales similarly to Quadrics' MPI for this application. The performance of SWEEP3D is discussed in the next section.



(a) Blocking calls

(b) Non-blocking calls

Figure 5.9: SWEEP3D Performance

5.4.3 Blocking vs. Non-blocking Communications

As stated in the previous section, bulk-synchronous applications with non-blocking or infrequent blocking communications run efficiently with BCS-MPI. However, fine-grained applications that use blocking communications or applications that group blocking communications are expected to perform poorly with BCS-MPI. The delays introduced by the blocking communications can considerably increase the applications' run time. Two approaches can alleviate this problem. The simplest option is to schedule a different parallel job whenever the application blocks for communication, thus making use of the CPU. This addresses the problem without requiring any code modification, but is not always practical due to memory and performance considerations. Alternatively, we have empirically seen that in such cases it is often feasible to transform the blocking communication operations into non-blocking ones, with a few simple code modifications.

To illustrate the second technique, we look at the SWEEP3D application [53]. Figure 5.9(a) shows the run time of SWEEP3D for both Quadrics MPI and BCS-MPI as a function of the numbers of processes. The slowdown amounts to approximately 30% in all configurations. Each process exchanges four messages with its nearest neighbors on every compute step using blocking send/receive operations. This communication pattern combined with the relatively fine granularity incurs a high overhead. On every compute step, the process blocks for 1.5 timeslices on average for every blocking operation. To eliminate this delay, we replaced every matching pairs of MPI_Send/MPI_Recv with MPI_Isend/MPI_Irecv and added MPI_Waitall after the four calls. That involved changing less than fifty lines of source code and improved dramatically the application performance, as shown in Figure 5.9(b). In this case, the overlapping of computation and communication along with the minimal overhead of the MPI calls allow BCS-MPI to slightly outperform Quadrics MPI.

6 Job Scheduling

6.1 Background

Job scheduling is arguably one of the least understood areas of parallel system software, and yet one that has a profound effect on the usability and productivity of these systems. The rapid evolution in supercomputing hardware and capabilities was not matched by a similar progression in production job schedulers [105]. In fact, most of the widely-used job schedulers are based on batch scheduling methods, which date back to the early supercomputers [28]. While these resource management systems and others incorporate some modern features such as load-balancing, process migration and checkpointing [5], little attention has been given so far to advances in job scheduling techniques outside academia. In particular, it has been shown that various methods of coscheduling (running several globally-coordinated jobs, time-sharing the same computation resources) can be used to increase overall system performance and utilization [31, 48]. Yet, many supercomputing centers still do not use these techniques despite their potential benefits, due to various reasons, including implementation difficulties, performance, and scalability issues.

The main goal of this dissertation is to demonstrate that through the use of modern interconnect features, the design and implementation of advanced job scheduling techniques can be simplified immensely, while still striving for high performance and scalability. In addition, by implementing and refining two novel job scheduling techniques, FCS and BCS, we lay out the foundations for several important techniques and innovations in parallel system software, such as dynamic monitoring of processes' scheduling requirements, load balancing in homogeneous and heterogeneous environments, deterministic scheduling of user and system activities and communication, and transparent fault tolerance. Another important contribution of this work is the analysis of complex, dynamic workloads under varying offered loads and scheduling algorithms. While this area of job scheduling research is still poorly understood, we offer several new methodologies, metrics, and insights on the subject.

The complexity inherent to such parallel systems and job scheduling leads many studies of the field to make simplifying assumptions and use various kinds of simulations. This work stresses the importance of realistic experimentation in job scheduling studies to promote reliable, reproducible results. We therefore implemented and refined several job scheduling algorithms on real clusters, using actual MPI applications.

We were thus able to evaluate issues such as scalability, performance, and responsiveness directly, without having to make any assumptions on the myriad known and unknown system parameters. The dynamic workloads did rquire a few assumptions however, since workloads and applications vary from site to site.

Another important aspect of parallel job scheduling that we chose to focus on is load imbalance. One source of load imbalance is heterogeneous architectures, such as computational grids and networks of workstations (NOWs), where different nodes can have different computation capabilities, different memory hierarchy properties or even a different number of PEs per node. Another factor is application load imbalance, that occurs when different parallel computation threads require different computation resources, and take varying times to complete. These can occur either as a result of poor programming, or more typically, by a data set that creates uneven loads on the different computation threads. Even when using homogeneous architectures and well-balanced software, load imbalance might show up. This can happen for instance when the compute nodes are not dedicated entirely to a single parallel computation, and may be used for other programs as well. This uneven taxing of resources again creates a situation where some of the parallel program processes run slower than others, and a load imbalance occurs.

Load imbalances have a marked detrimental effect on many parallel programs. Many HPC applications behave in a bulk-synchronous parallel (BSP) model (see Section 3.3 and [33, 62, 115]). A load imbalance can harm the performance of the parallel application because each computation thread requires a different amount of time to complete, but the entire program must wait for the slowest thread before it can synchronize. Since these computation/synchronization cycles are potentially executed many times throughout the lifetime of the program, the cumulative effect on the application run time and the system resource utilization can be quite high.

This chapter is dedicated to the comparative evaluation of several scheduling algorithms on a wide spectrum of workload parameters, with an emphasis on actual implementation and applications. In most experiments we use the following scheduling algorithms: first-come-first-serve (with or without backfilling), gang scheduling, spin-block, flexible coscheduling and buffered coscheduling. In the next section, we describe these algorithms and other relevant work in the field. The following sections describe and analyze various aspects of job scheduling, including static workload performance and load-balancing capabilities, dynamic workloads and scheduling metrics, and overlapping of system resources.

6.2 Related Work

Many previous studies have also proposed to increase parallel systems and applications performance by using job scheduling techniques. Traditional job allocation and scheduling methods such as those employed in PBS [51], NQS [64] and LSF [121] use space sharing, wherein every job receives a dedicated partition of nodes. While this allocation is preferable from the application's point of view, it often exposes various performance problems that lead to wasted system resources. Furthermore, it reduces system responsiveness

since jobs often have to wait in queues before they can be allocated a dedicated partition.

To address the responsiveness problem, Ousterhout first suggested in [78] a method to share machine resources over time as well as space (generally called *coscheduling* jobs). Perhaps the simplest approach to time sharing is local scheduling, where each node's operating system schedules processes as it sees fit, with no attempt of global coordination whatsoever. This approach was shown to be very detrimental to parallel application performance [22, 39]. Another approach, called *Gang Scheduling* (GS) partitions time into slots, so that each job runs in one timeslot[1], and thus receives the service of a dedicated virtual machine, but without having to wait for all the previous jobs to terminate. This offers users similar advantages to those of multiprogramming on a single machine – namely improved responsiveness, but also incurs higher memory pressure and job turnaround time. The main problem with this approach is resource fragmentation, as discussed in Section 6.5. Another limitation with most existing implementations of GS is that context-switching between jobs can be both costly in overhead and non-scalable due to the requirement of a central synchronization signal.

Some contemporary approaches to job scheduling try to avoid this limitation by eliminating the global synchronization. All these approaches try to coschedule synchronizing processes without explicit coordination messages by taking scheduling decisions locally on every node. One approach, called Demand-Based Coscheduling (DCS) [103, 104], tries to synchronize processes by giving an immediate priority boost to any process for which an incoming message arrives. This causes a context-switch to the prioritized process, which in many cases is a desired effect, (since its communication peer is also scheduled at this time with a high probability), but can also cause fairness and overhead problems. Implicit Coscheduling (ICS) [2] methods try to coschedule communication processes with a different approach, by using a spin-block (SB) mechanism. Processes that perform blocking (synchronous) communication try to actively wait (spin) for a given interval before blocking while waiting for the call to complete. Thus, processes tend to self-synchronize and be coscheduled, completing their communication calls before the blocking time. In [76], a comparative study presents an elaborate taxonomy and comparison of implicit coscheduling algorithms, and also suggests some new schemes based on periodic rescheduling. These methods, called *Periodic Boost* (PB), are based on boosting the priority of communicating processes on a periodic basis, thus eliminating the need for an interrupt on communication events.

While all these methods can be effective in coscheduling regular applications, they have no special provisions for irregular applications, such as load-imbalanced jobs or jobs requiring gang-scheduling due to strict synchronization needs. Lee et al. suggested in [66] that a scheduler should gather meaningful statistics about the running processes, and adapt the scheduling to the processes' specifics requirements. Based on this idea, we developed and implemented a first version of such a scheduler, called FCS [38]. FCS combines the advantages of global coordination with local scheduling decisions based on gathered process statistics and was shown to perform well in a varied range of workloads. Another scheduling

[1] Some versions of GS allow jobs to run on more than one time slot, if it can still run all the jobs' processes concurrently [22].

algorithm that is based on both global coordination and local decisions is Buffered Coscheduling [80]. Under this method, process communication is not executed immediately but rather buffered till the next system heartbeat, when it can be scheduled with the peers. FCS was first implemented in the author's M.Sc work, and was later simplified and re-evaluated for this work. BCS was simulated in previous work [80, 81], but never implemented on a real cluster before this work. Both strategies require some support from an advanced interconnect to perform effectively, and are described in more detail below. More information on other job scheduling methods can be found in [28].

6.2.1 Flexible Coscheduling

Flexible Coscheduling (FCS) is a coscheduling strategy proposed by Frachtenberg and Feitelson to address issues of load imbalance and resource fragmentation [38]. FCS has undergone several iterations of refinements and simplifications since, and was shown in [39, 40] to substantially increase the resource utilization in a cluster in several load-imbalance scenarios. The main idea behind FCS is to dynamically detect and compensate for load imbalances. Dynamic detection of load imbalances is performed by (1) monitoring the communication behavior of applications, (2) defining metrics for their communication performance that try to detect possible load imbalances, and (3) classification of the applications according to these metrics. Using this classification, FCS attempts to coschedule processes that would most benefit from it, while scheduling other processes to increase overall system utilization and throughput. FCS works for optimizing the global resources of a cluster: a specific application that suffers from load imbalances will not complete faster with this scheduler compared to other schedulers.[2] Instead, FCS prevents load-imbalanced jobs from wasting too many system resources, and thus improving the overall system efficiency and responsiveness.

Process Classification

Application processes are categorized into one of three classes (Figure 6.1):

1. *CS* (coscheduling): These processes communicate often, and must be coscheduled (gang-scheduled) across the machine to run effectively, due to their demanding synchronization requirements.

2. *F* (frustrated): These processes have enough synchronization requirements to be coscheduled, but due to load imbalance, they often cannot make full use of their allotted CPU time.

3. *DC* (don't-care): These processes rarely synchronize, and can be scheduled independently of each other without penalizing the system's utilization or the job's performance. For example, a job using a coarse-grained workpile model would be categorized as *DC*.

We can also define another class, *RE* (rate-equivalent), for jobs that have little synchronization, but require a similar amount of CPU time for all their processes. However, detection of *RE* processes cannot be made in

[2] Obviously, any given application receivess the best service when running by itself, as when running in batch mode.

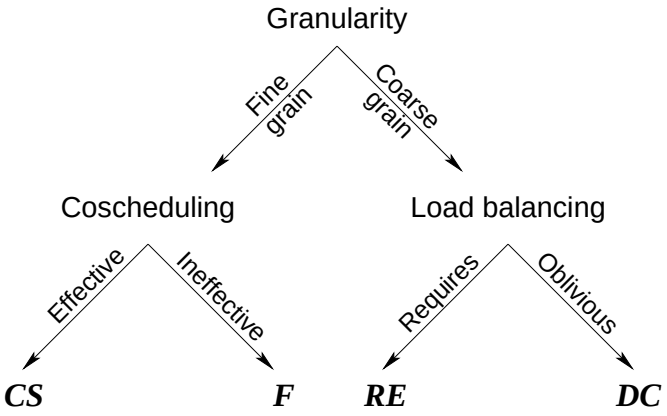


Figure 6.1: Decision tree for FCS process classification

run-time with local information only, so they are classified as *DC* instead, due to their low synchronization needs.

Figure 6.1 shows the decision tree for process classification. Each process is evaluated at the end of its timeslice. When a process communicates at relatively coarse granularity, it is classified *DC*. Otherwise, the process is classified according to how effectively it communicates when coscheduled: if effective, it is a *CS* process. Otherwise, some load imbalance prevents the process from communicating effectively, and it is considered *F*. Classification is based on dynamic measurement of each process' communication granularity and effectiveness. To estimate the granularity and effectiveness of a process's communication operations, we modified the MPI library to take time measurements for blocking communication calls, and expose these to the NM[3]. More details on the measurement and classification mechanisms can be found in [39].

FCS Scheduling

The principles behind scheduling in FCS are as follows:

- *CS* processes should be coscheduled and should not be preempted.
- *F* processes should be coscheduled but can be preempted when synchronization is not effective.
- *DC* processes impose no restrictions on scheduling.

Another difference between a *CS* process and a *F*/*DC* process is that the former always block on synchronous communication, while the latter performs spin-blocking. The reasoning for this is that *CS* processes are known to be coscheduled, and cannot benefit from blocking, whereas other processes can yield the CPU to other needing processes when they are waiting for their peers to communicate.

[3]We use some of QsNet non-intrusive mechanisms for this, to avoid interrupting the CPU.

The infrastructure used to implement this scheduling algorithm is based on the implementation of STORM's gang scheduler. The machine manager (MM) packs the jobs into an Ousterhout matrix. It periodically sends multi-context-switch messages to the node managers (NM) instructing them to switch from one slot to another. A crucial characteristic is that the node managers do not have to comply: they are free to overrule the MM's directives based on their local measurements and classifications. On each global context switch they determine whether to run the next process exclusively (if it is *CS*), concurrently with other *DC* processes but prioritized (if it is *F*), or non-prioritized (for *DC* processes). In practice, there are several technical and implementation details that need to be accounted for, and can be found in [39].

6.2.2 Buffered Coscheduling

BCS is also conceptually composed of two parts: communication handling and scheduling. The communication buffering, scheduling, and execution mechanisms are described in Section 5.3. The important thing to keep in mind is that any blocking communication prompts an immediate blocking of the communicating process, since the actual data transfer will not occur until the next micro-timeslice. This affects the BCS job scheduling, causing an immediate context-switch for every blocking communication call. This might be detrimental to highly-synchronous applications' performance, due to the context-switch overhead, or to wasted cycles if only a single application is running. On the other hand, non-blocking applications do not necessarily suffer from BCS when running alone (see Section 5.4.2). Furthermore, BCS offers an opportunity for overlapping of communication and computation between different applications, since the communication is handled entirely by the NIC, irrespective of which application is currently running.

Job scheduling in BCS, like in FCS, is based on global (gang) scheduling, with an autonomy for the NM to make local scheduling decisions. In BCS, the MM issues both global context-switch signals like in GS or FCS (in the order of magnitude of tens of ms), and micro-timeslice strobes (every few hundreds of μs), that signal the start of the next communication phase. The NM obeys the global context-switch signals by switching to the appropriate task on all nodes. Whenever a process blocks for a synchronous communication call, the NM will perform an immediate local context switch to the next runnable (non-blocked) process, in a round-robin fashion. Thus, jobs receive a relatively fair share of the PEs and an opportunity to coschedule asynchronous calls while they are not issuing blocking calls, but do not spend any PE cycles waiting for communication to complete. In particular, this allows BCS to adjust very efficiently to load-imbalances, if sufficient applications with non-blocking calls exist to use as filler for the gaps created by load imbalance.

6.3 Static Workload Evaluation

6.3.1 Methodology

Before we proceed to complex and realistic scenarios, it can be productive to compare the basic properties of each scheduler with simple scenarios that are easy to control. To this end we first evaluate and compare different job scheduling algorithms with static workloads (i.e. with no dynamic job arrivals), using both synthetic benchmarks, where we can control every aspect of the workload, and simple workloads of real applications that demonstrate issues of memory pressure, paging and context switch penalty. For the synthetic benchmark, we use the simple BSP program described in Section 1.4, where the computation granularity, type and length of communication, and load imbalance can be easily set in advance. For the application tests, we use the ASCI-representative programs, SAGE and SWEEP3D, also described in Section 1.4.

We use two metrics to compare the performance of different scheduling algorithms [33] :

- Turnaround time is the total running time (in seconds) of the entire workload.
- Average response time is the mean time it takes a job to complete running from the time of submittal (which is not necessarily the actual execution time).

Turnaround time is considered a system-centric metric, since it describes the reciprocal of the system's throughput. Response time on the other hand is more interesting to users, that would like to minimize the time they wait for their individual jobs to complete. In practice, it is difficult to discuss these metrics in isolation, since with real dynamic workloads, various factors and feedback effects create interactions between the metrics [33, 40]. However, the four scenarios we describe in the next section are simple enough to allow a comprehensive understanding of the factors involved. In fact, it is also easy to calculate for each scenario what the optimal turnaround and response time values would be under an ideal scheduler. We believe that this set of synthetic tests covers a wide spectrum of basic workload combinations.

6.3.2 Synthetic benchmarks

The basic "building block" application for this set of experiments is the synthetic BSP program described in Section 1.4. We run this application on four processors (two nodes) of the Accelerando cluster ($25ms$ timeslice), with medium-fine granularity of $3ms$ and approximately $60s$ of total execution time when run in isolation (Fig. 6.2). However, we do vary the amount of work performed in each loop for some processes to create imbalanced scenarios. Each experiment was repeated several times and the results were averaged (in practice, the deviation of results was limited to less than 2%). The communication pattern used was a nearest-neighbor ring pattern with synchronous (blocking) communication calls, which represents a worst-case scenario for BCS. Some variations to this pattern and experimental setup are discussed in Section

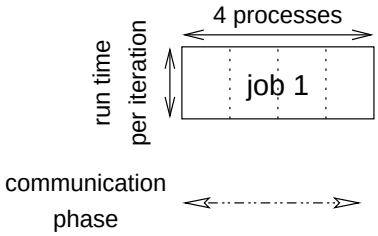


Figure 6.2: One iteration of the "building-block" job

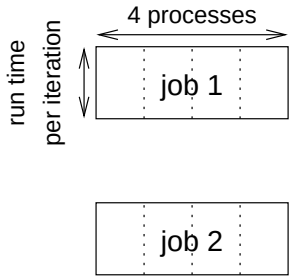


Figure 6.3: Compute time for one iteration of two basic jobs

Algorithm	Job 1	Job 2	Turnaround	Mean Response
FCFS	60.04	120.12	120.12	**90.08**
GS	120.08	120.00	**120.08**	120.04
SB	124.09	124.1	124.09	124.09
FCS	120.35	120.35	120.35	120.35
BCS	131.70	131.72	131.72	131.71

Table 6.1: Balanced workload performance comparison

6.3.2.5. An exploration of the parameter space for different choices of granularity and "noise" value is presented in [39].

6.3.2.1 Balanced Jobs

Many HPC applications such as SWEEP3D are latency bound, in the sense that they synchronize often with short messages [53]. These synchronous applications require that all their processes be coscheduled to communicate effectively: if another application or system dæmon interrupts their synchronization, large skews can develop that significantly hamper their performance [88].

In the first scenario, we emulate such situations by running two identical, medium-grained jobs concurrently. Figure 6.3 depicts the run time *per iteration*, which is balanced and equal for both jobs[4]. Table 6.1 presents the results for running this workload, giving the termination time in seconds for each job and for the complete set. It also shows the total turnaround and mean response times, all in seconds.

[4]In this and the following figures, only the compute time per iteration is shown, omitting the communication phase.

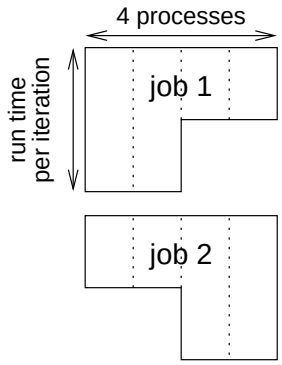


Figure 6.4: Two load-imbalanced jobs

Since synchronous balanced jobs require a dedicated environment to proceed effectively, FCFS scheduling and GS offer the best performance. SB scheduling shows some slowdown when compared to the others, due to the lack of global coordination (SB actually performs much worse in relative terms as the granularity becomes finer). FCS exhibits performance comparable to that of GS, since all processes are classified as *CS*, and are therefore gang-scheduled. Still, total turnaround time is slightly higher than that of GS, due to the added overhead of process classification. BCS displays a $\approx 10\%$ slowdown in the turnaround time since blocking communication calls have an average latency of 1.5 micro-timeslices ($375\mu s$ here), which cannot always be filled by the other process.

When considering response time, batch scheduling is the only algorithm that has a significant advantage over the other algorithms, since job 1, having run in isolation, terminates quickly and lowers the FCFS average.

6.3.2.2 Load-Imbalanced Jobs

This scenario represents a simple load imbalance case with two complementing jobs, as depicted Figure 6.4. Processes 1 and 2 of the first job compute twice as much *per iteration* as the other two processes, while the situation is reversed for the 2nd job [5]. The faster processes compute the same amount as in the previous scenario. In a sense, these workload represents the exact opposite of the previous one, where jobs needed a dedicated partition to communicate effectively. In contrast, these unbalanced jobs are guaranteed to waste compute resources when running in isolation. Table 6.2 shows the performance of each scheduling algorithm.

It can be seen from the data that both FCFS and GS take almost twice as much time to run each job (compared with the previous scenario), whereas the total amount of computation per job is only increased by 50%. SB does a much better job at load-balancing, since the short polling interval allows the algorithm to yield the CPU when processes are not coscheduled, giving the other job a chance to complete its com-

[5] In reality, the speed ratio is slightly over 2:1, to bring the total runtime of each job to $120s$. The gap is produced by the communication time, which is unchanged, requiring additional computation to increase the run time.

Algorithm	Job 1	Job 2	Turnaround	Mean Response
FCFS	111.89	231.91	239.91	**179.90**
GS	240.58	240.60	240.60	240.59
SB	183.34	183.43	**183.43**	183.38
FCS	183.49	183.62	183.62	183.56
BCS	193.69	193.84	193.84	193.77

Table 6.2: Two load-imbalanced jobs performance comparison

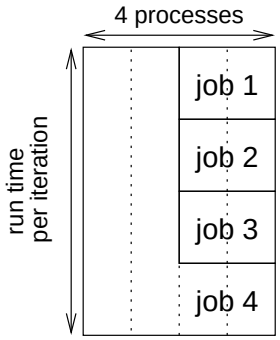


Figure 6.5: Complementing jobs

munication and wasting little CPU time. FCS is also successful in exploiting these computational holes. After a brief interval, it classifies the first job's processes as *DC*, *DC*, *F*, and *F* respectively, and the second job's as *F*, *F*, *DC*, and *DC*. The resulting scheduling is effectively the same as SB's, with the exception that *F* processes are prioritized when their assigned slot is the active one. The total turnaround time is similar to SB's, and represents near-optimum resource utilization: both jobs complete after running for $\approx 150\%$ of the time it took the previous scenario, which corresponds to the new amount of work. BCS is again paying a price for the blocking nature of the communication patter, but still remains within $\approx 10\%$ of the optimal response time and turnaround.

The response time metric again shows some preference to batch scheduling, although FCS and SB are not far behind, due to their lower turnaround time. GS exhibits the same turnaround time as FCFS, but since all the process terminate concurrently, the mean response time is actually higher.

6.3.2.3 Complementing Jobs

The third scenario exposes the ability of various algorithms to pack jobs efficiently in an extremely imbalanced workload. It consists of one four-process job and three two-process non-communicating jobs running on one node (Fig. 6.5). All the jobs running on PEs 3 and 4 compute for about $60s$, as in the previous scenarios, but processes 1 and 2 of job 4 compute four times as much per iteration. An optimal scheduler should pack all these jobs so that the total turnaround time does not exceed that of the first job when run in isolation (assuming zero context-switch overhead). Table 6.3 again shows the run times for

Algorithm	Job 1	Job 2	Job 3	Job 4	Turnaround	Mean Response
FCFS	60.50	120.96	181.50	420.67	420.67	**195.91**
GS	241.19	241.14	241.26	423.30	423.30	286.73
SB	232.61	232.23	232.33	271.09	271.09	242.07
FCS	245.50	245.40	245.40	250.15	**250.15**	246.61
BCS	223.61	223.72	223.55	343.39	343.39	253.57
PBCS	240.32	240.53	240.30	270.17	270.17	247.83

Table 6.3: Complementing jobs performance comparison

each algorithm.

Once more, FCFS and GS exhibit similar turnaround time — the combined run time of all the jobs run in isolation. The imbalance is large enough to allow SB to load-balance the jobs relatively well, resulting in a lower turnaround time, but since it lacks a detailed knowledge of the processes requirements, it can only go so far — Job 4 still shows a significant slowdown ($\approx 31\%$) when compared to FCFS. With FCS the situation is even better. After a short while, the scheduler classifies all the processes as *DC*, except for processes 3 and 4 of job 4 which are classified as *F*. As such, they receive priority in their time slot, and thus the total runtime of job 1 suffers a slowdown of only $\approx 3\%$ from the interference of the other jobs, which pack neatly into the other timeslices. This reduces the turnaround of this workload to within $\approx 3\%$ of the optimally packed value, partly due to the initial classification delay.

While BCS exhibits some degree of load balancing, it fares much worse than FCS and SB. Jobs 1–3 benefit from the many computational "holes" provides by job 4's frequent communication. On the other hand, since they do not communicate, they can make full use of their own slot. The result is a significantly reduced run time for jobs 1–3, at the cost of job 4's run time and the overall turnaround time. Since prioritizing job 4 proved so useful with FCS, we have implemented a variant of BCS, called PBCS, where a higher priority can be assigned to a job, allowing it to run whenever it is ready. As Table 6.3 shows, PBCS's performance is more like that of FCS, losing only 8% in the turnaround time to blocking communication calls that could not be overlapped. Still, the prioritizing must be specified explicitly by the user, whereas FCS automatically detects and prioritizes *F* processes.

The mean response time metric shows a clear preference only to FCFS in this scenario, due to the short jobs in the first part of the workload. FCFS' mean response time can become much worse if the job order were reversed – around $330s$. Time sharing algorithms however are not as sensitive to job order, which becomes an advantage when the order is not known in advance.

6.3.2.4 Mixed Jobs

The last synthetic scenario is designed to expose the interaction between synchronous balanced and imbalanced jobs in a mixed workload. This situation might occur whenever a machine is running more than one type of application, or with different data sets that have different load balancing properties. Even when

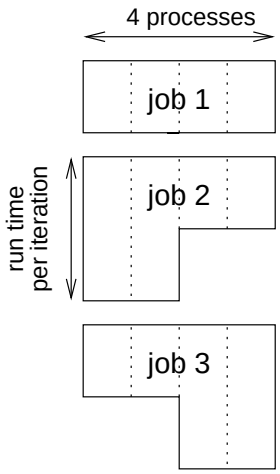


Figure 6.6: Mixed jobs

Algorithm	Job 1	Job 2	Job 3	Turnaround	Mean Response
FCFS	60.01	179.92	299.12	299.12	**179.68**
GS	180.11	298.75	298.72	298.75	259.19
SB	191.14	277.31	277.11	277.31	248.52
FCS	150.59	250.32	250.18	**250.32**	217.03
BCS	186.12	270.97	270.94	270.97	242.67
PBCS	104.22	275.65	275.71	275.71	218.53

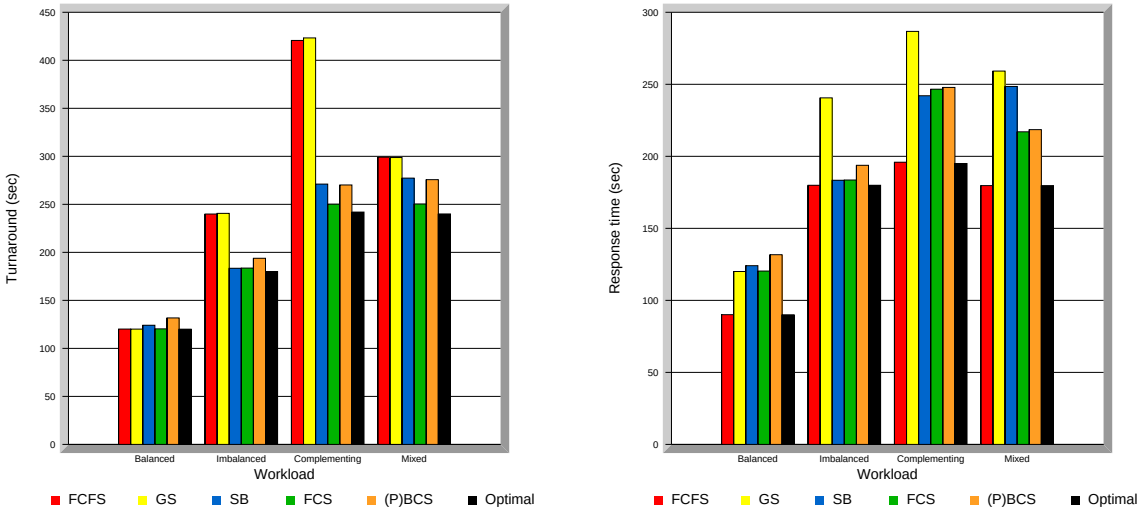
Table 6.4: Mixed jobs performance comparison

the workload is composed of only balanced applications, this situation can occur whenever job arrivals and sizes are dynamic. For example, in a time sharing system, different nodes might run different numbers of jobs, thus creating a dynamic imbalance.

We encapsulate some of this complexity in a set of three jobs. The first job is the basic, load-balanced jobs used in Section 6.1. The last two are complementary imbalanced, and identical to jobs 1 and 2 of the second scenario (Section 6.2). Table 6.4 shows the run time results for the four algorithms.

Once more, batch and gang scheduling perform similarly, with an advantage in response time to FCFS. SB scheduling's major weakness is exposed in this scenario: since it gives an equal treatment to all processes, fine-grained jobs suffer from the interruptions incurred by other jobs. This is clearly shown in the performance of job 1 under SB, which is worse than with any other algorithm. In contrast, FCS, which is another loosely-coupled algorithm, identifies the special requirements of this job and classifies it as *CS*. As such, it receives dedicated time slots that allow it to communicate effectively. In fact, since the *CS* job also receives some CPU time in the *DC* time slots, it actually has a small advantage over the other two jobs, thus completing before the expected time (as in GS). The overall result is a decrease in turnaround time and mean response time when compared to the other algorithms.

BCS suffers from an inability to fill the computational holes efficiently in this scenario: job 1 serves as



(a) Turnaround

(b) Mean response time

Figure 6.7: Comparative performance across scheduling algorithms and workloads

a poor filler on the one hand (due to its coscheduling requirement), while on the other hand, it offers little opportunity for the two other jobs to fill holes. However, prioritizing job 1 with PBCS shows a marked improvement, since it affords the job many more opportunities at coscheduling. While job 1 receives more than its fair share of machine resources, its quick termination actually assists the other two jobs, when compared to GS. In this sense, FCS shows more fairness and balancing in allocation of resources.

6.3.2.5 Discussion and Variations

Figure 6.7(a) summarizes the turnaround time performance of the different algorithms in all four scenarios (PBCS is used instead of BCS for the last two workloads). An additional bar for each cluster shows the performance that would be obtained by an optimal scheduler, derived analytically. Note that even overhead resulting from context switches is not factored into the optimal figure. A clear distinction can be seen between the rigid algorithms (FCFS, GS) and the dynamic ones for all but the balanced scenario. In particular, FCS's performance is always near-optimal, attesting to its ability to adapt to different balancing requirements.

Similarly, Figure 6.7(b) compares the average response time. In the tested scenarios, response time metric clearly favors FCFS, due to ordering shorter jobs first in the workload. In fact, using a different ordering would raise the FCFS bar to levels at or above FCS's response time, while no other bar would be affected. This is an indicator of FCFS' sensitivity to job arrival order (which is typically not tightly controlled). Both FCS and BCS display good response time performance, which is near-optimal when the jobs are re-ordered.

Workload	Job 1	Job 2	Job 3	Job 4	Turnaround	Mean Response
Balanced	122.64	122.65	-	-	122.65	122.65
Imbalanced	188.92	188.60	-	-	188.92	188.76
Complementing	247.00	246.95	246.96	252.00	252.00	248.23
Mixed	156.29	252.82	252.76	-	252.82	220.62

Table 6.5: BCS performance with asynchronous communication

To further understand the sensitivity of different algorithms to the communication mechanism used, we repeated the experiments with two alternate mechanisms: non-blocking point-to-point followed by an MPI_Waitall in the next iteration, and barrier synchronization. As it turns out, the results for barrier synchronization are similar across the board, varying only by a few percent from the blocking point-to-point counterparts. Asynchronous communication on the other hand shows a slight performance improvement for all the algorithms (and the applications, since the effective latency is reduced, as explained in Section 5.4.3). The exception is with BCS, which shows a marked performance improvement in all the workloads. As can be seen in Table 6.5, BCS's performance with non-blocking communication is quite similar to that of FCS, with similar load-balancing advantages. This is due to the fact that like FCS, BCS has the ability to coschedule jobs on their slot, while filling computational holes (unlike GS and SB).

We have repeated some of these experiments on Crescendo, to evaluate the effect of a different architecture [39]. Crescendo represents a much more balanced (albeit slower) architecture than Accelerando, since its processors are slower but its network is faster, due to a better PCI bus implementation. Because of this, the system has more opportunities to overlap communication and computation (which is done automatically by the QsNet NIC, even to some small extent with blocking calls [41]). This is especially apparent with the algorithms that are not strictly coordinated, such as SB and FCS. For example, in the most extreme case, the complementing jobs scenario, they obtain a total turnaround time of $261.2s$ and $236.3s$ respectively, which is approximately 5% better than on Accelerando.

6.3.3 MPI Applications

In this section we evaluate the same scheduling algorithms with simple static workloads composed of two real applications, SAGE and SWEEP3D (see 1.4). These applications are representative of the applications that consume most of the computation cycles on the ASCI machines, including ASCI Q. We ran SWEEP3D with $\approx 40,000$ cells per PE taking up $277MB$ of main memory per process (considered realistic for ASCI workloads). In this configuration, SWEEP3D exhibits a considerably fine computation granularity of $\approx 2ms$ with very little variability or load imbalance. For SAGE, we used the realistic input deck "timing_h.input" with $500,000$ cells per PE, taking up approximately $520MB$ of main memory per process. SAGE's internal work distribution however diverges over time, resulting in varying computation granularities for different processes and times, mostly residing in the range $2-21ms$.

Algorithm	SAGE / SAGE	SWEEP3D / SWEEP3D	SAGE / SWEEP3D
FCFS	224 / 448	217 / 434	223 / 444
GS	527 / 527	445 / 445	446 / 447
SB	644 / 644	593 / 594	369 / 450
FCS	553 / 554	451 / 451	473 / 472
BCS	632 / 633	563 / 563	301 / 541

Table 6.6: Completion time (sec) of ASCI applications with different algorithms and workloads

For both applications we used 49 PEs (the largest cubic configuration on Accelerando, required an optimal data distribution for SWEEP3D), and a timeslice of $75ms$ to minimize the penalties of a context switch (this timeslice can still be considered very responsive in human terms).

We use three simple workloads: one composed of two copies of SAGE, one with two copies of SWEEP3D, and the last with a copy of each. Table 6.6 reports the completion time of each job for each of the algorithms and workloads.

These fine-grained applications prove to be a worst-case scenario for multiprogramming schedulers, and in particular the dynamic ones. The very fine granularity combined with the memory pressure created by these programs favors dedicated or partly-dedicated (GS with a coarser time quantum) allocation of resources. In particular, BCS and SB show considerably poor performance, the former because of the high frequency of blocking communication calls, and the latter due to the lack of coscheduling, which is vital for fine-grained synchronization [22]. It is interesting to note that both algorithms favor SAGE in the mixed workload, presumably due to its coarser granularity. FCS performs somewhat similar to GS in the 2nd workload, but does not fare as well with the other two, when SAGE is involved: that is due to the fact that some of SAGE's processes are classified as F, so coscheduling of the entire job is not always guaranteed.

These HPC applications are an example of highly optimized and even self-balancing programs that have been tuned to maximize resource utilization, leaving little room for a scheduler to improve. However, in many real workloads, much of the load imbalance and resource waste is not caused so much by internal application inefficiency but rather by the dynamic nature of the workload, which creates many "allocation holes". The next section analyzes the ability of different schedulers to deal with more realistic workloads, while in Section 6.5 we analyze what additional improvement in resource utilization can be extracted even from such applications.

6.4 Dynamic Workload Evaluation

6.4.1 Background and Methodology

While the static workloads presented in the previous section serve to understand the basic properties of different scheduling algorithms, they are not very representative of real-world usage of supercomputers. Most

computing centers have a job queue (sometimes, multiple job queues) to which jobs are added dynamically, and these jobs can have almost arbitrary time and space requirements. In order to provide a more realistic comparative evaluation of different scheduling algorithms, dynamic workloads must be addressed.

One important aspect of real workloads is that they are dynamic: jobs arrive at unpredictable times, and run for (largely) unpredictable times. The set of active jobs therefore changes with time. Moreover, the number of active jobs also changes with time: at times the system may be empty, while at others many jobs are waiting to receive service. The overall performance results are an average of the results obtained by the different jobs, which were actually obtained under different load conditions. These results also reflect various interactions among the jobs. Such interactions include explicit ones, as jobs compete for resources, and implicit ones, as jobs cause fragmentation that affects subsequent jobs. The degree to which such interactions occur depends on the dynamics of the workload: which jobs come after each other, how long they overlap, etc. It is therefore practically impossible to evaluate the effect of such interactions with static workloads in which a given set of jobs are executed at the same time.

Many of the supercomputer sites used batch scheduling variants. Recent research has revisited the comparison of gang scheduling with other schemes, and has led to new observations. One is that gang scheduling may be limited due to memory constraints, and therefore its performance is actually lower than what was predicted by evaluations that assumed that memory was not a limiting factor. The reason that memory is a problem is the desire to avoid paging, as it may cause some processes within parallel jobs to become much slower than other processes, consequently slowing down the whole application. The typical solution is to allow only a limited number of jobs into the system, effectively reducing the level of multiprogramming [7, 120]. This hurts performance metrics such as the average response time because jobs may have to wait a long time to run.

Another observation is that alternative scheduling schemes, such as backfilling [68] may provide similar performance. Backfilling is an optimization that improves the performance of pure space slicing by using small jobs from the end of the queue to fill in holes in the schedule. To do so, it requires users to provide estimates of job runtimes. Thus it operates in a more favorable setting than GS, that assumes no such information. Moreover, moving short jobs forward achieves an effect similar to the theoretical "shortest job first" algorithm, which is known to be optimal in terms of average response time.

Our goal in this section is to improve our understanding of these issues, by performing an emulation-based evaluation of several job scheduling schemes. To do so, we need also to find good values for several parameters parameters, namely the MPL and the length of the time slicing quantum. As these parameters are intimately related to the dynamics of the workload, the evaluation is done using a dynamic workload model.

In this section, we evaluate the following scheduling schemes: FCFS, GS, FCS, SB, Backfilling as in EASY, and combinations of EASY with other schemes. The first four algorithms make use of a queue for jobs that arrive and cannot be immediately allocated to processors. The allocation of jobs from the

queue can be handled with many heuristics, such as first-come-first-serve, shortest-job-first, backfilling, and several others. We chose to implement and use EASY backfilling [68], where jobs are allowed to move forward in the queue if they do not delay the first queued job. Zhang et al. studied these issues in [120] and found that combining backfilling with gang-scheduling can reduce the average job slowdown when the MPL is bounded (our experiments described below agree with these results). They also point out that for coscheduling algorithms such as GS, there is no exact way of predicting when jobs will terminate (or start), because the effective MPL varies with load. To estimate these times, we use the method they recommend, which is to multiply the original run time estimate by the maximum MPL.

BCS was excluded from this evaluation due to a bug in Elanlib that prevented running BCS with a large number of jobs[6]. All the experiments described below were run using 32 processors on the Crescendo cluster (see Table 1.1).

Workload

The results of performance evaluation studies may depend not only on the system design but also on the workload that it is processing [29]. It is therefore very important to use representative workloads that have the same characteristics as workloads that the system may encounter in production use.

The two common ways to evaluate a system under a dynamic workload are to either use a trace from a real system, or to use a dynamic workload model. While traces have the benefit of reflecting the real workload on a specific production system, they also risk not being representative of the workload on other systems. We therefore use a workload model proposed by Lublin [70], which is based on invariants found in traces from three different sites, and which has been shown to be representative of other sites as well [111]. This also has the advantage that it can be tailored for different environments, e.g. systems with different numbers of processors.

While the workload model generates jobs with different arrival times, sizes, and runtimes, it does not generate user estimates of runtimes. Such estimates are needed for EASY backfilling. Instead of using the real runtime as an estimate, we used loose estimates that are up to 5 times longer. This is based on results from [75], which show that overestimation commonly occurs in practice and is beneficial for overall system performance.

Using a workload model, one can generate a workload of any desired size. However, large workloads will take a long time to run. We therefore make do with a medium-sized workload of 1000 jobs, that arrive over a period of about 8 days. The job arrivals are bursty, matching observations from real workloads, and most jobs sizes tend to be small (with the median here at just under 4 PEs), and biased toward powers of two. Some statistical properties of the workload can be found in [40].

To enable multiple measurements under different conditions, we reduce time by a factor of 100. This means that both runtimes and inter-arrival times are divided by a factor of 100, and the 8 days can be

[6] This problem is expected to be fixed in a future release of Elanlib.

emulated in about 2 hours. Using the raw workload data, this leads to a very high load of about 98% of the system capacity. To evaluate lower loads we divide the execution times by another factor, to reduce the jobs' run time and thus reduce the load.

Another simplifying assumption was made regarding the application to run. Different sites use a wide spectrum of applications, with different scalability, communication, and memory properties. Instead of trying to model a selection of applications, we use a more general approach. Since a large part of HPC software can be modeled using the BSP model, we chose to use a synthetic test application based on this model, where it is easy to control important parameters, such as execution time, computation granularity and pattern, and so forth. We use the same synthetic application from the previous section, which consists of a loop that computes for some time, and then exchanges information with its nearest neighbors in a ring pattern. The amount of time it spends computing in each loop (the computation granularity) is chosen randomly with equal probability from one of three values appropriate for Crescendo: fine-grained ($5ms$), medium-grained ($50ms$), and coarse-grained ($500ms$).

Metrics

STORM produces log files of each run, containing detailed information on each job (e.g. its arrival, start, and completion times, as well as algorithm-specific information). A set of scripts is used to analyze these log files and calculate various metrics. we focus on average response time (defined as the difference between the completion and arrival times) and average bounded slowdown. The bounded slowdown of a job is defined in [34], and we modified it to make it suitable for time-sharing environments by using:

$$Bounded\,Slowdown = \max\left\{\frac{T_w+T_r}{\max\{T_d,\tau\}}, 1\right\}$$

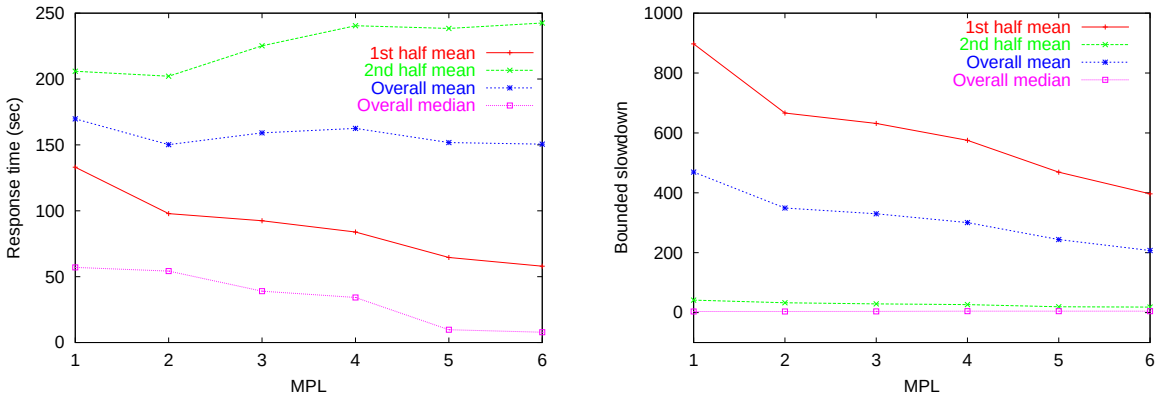
Where:

- T_w is the time the job spends in the queue.
- T_r is the time the job spends running.
- T_d is the time the job spends running in dedicated (batch) mode.
- τ is the "short-job" bound parameter. We use a value of 10 seconds of real time (0.1 sec emulated).

In some cases, we also divide the jobs into two halves: "short" jobs, defined as the 500 jobs with the shortest execution time, and "long" jobs — the complementing group. For this classification, we always use the execution time as measured with FCFS (batch) scheduling, so that the job groups remain the same even when job execution times change with different schedulers.

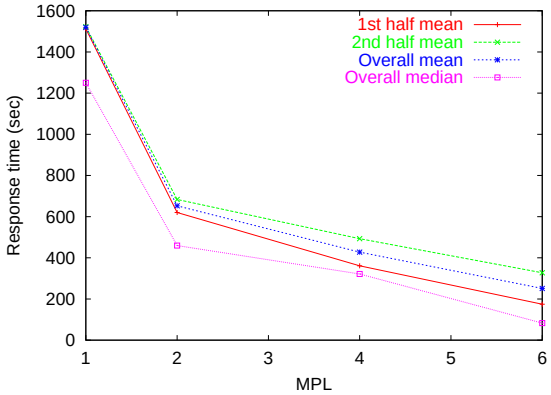
6.4.2 Effect of Multiprogramming level

The premise behind placing a limit on the MPL is that scheduling algorithms should not dispatch an unbounded number of jobs concurrently. One obvious reason for this is to avoid exhausting the physical

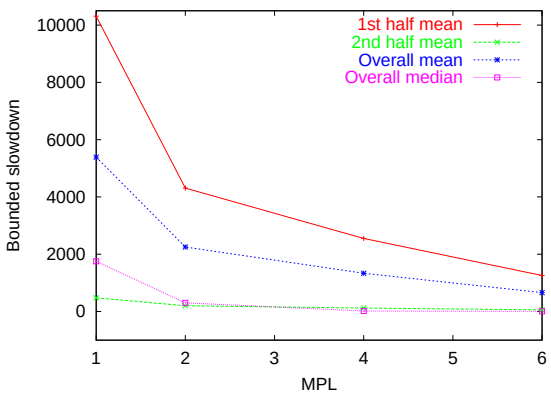


(a) Response time with backfilling

(b) Bounded slowdown with backfilling



(c) Response time, no backfilling



(d) Bounded slowdown, no backfilling

Figure 6.8: Effect of MPL with dynamic workload

memory of nodes. We define the MPL to be the maximum allowable over-subscription of processors. Naturally, the MPL for FCFS is always one, whereas for coscheduling algorithms it can be higher.

We set out to test the effect of the MPL on GS, with two goals in mind: (1) obtain a better understanding on how a limited multiprogramming level affects serving of dynamic workloads, and (2) find a good choice of an MPL value for the other sets of experiments. In practice, the question of how the MPL affects scheduling is sometimes moot, since often applications require a sizable amount of physical memory, which limits the amount of jobs that can run concurrently in a node without swapping. While some applications are not as demanding, or can be "stretched" to use a smaller memory footprint, we do accept the existence of a memory wall. Moreira et al. showed in [74] that an MPL of 5 provides in practice similar performance to that of an infinite MPL, so we put the bound on the maximum value the MPL can reach at 6.

For this section, we use the workload file factored to an average offered load of $\approx 74\%$, chosen so that it would stress the system enough to bring out the difference between different MPL values, without saturating it. GS was chosen due its relative popularity and its simplicity, being the most basic coscheduling method.

Figure 6.8(a) shows the effect of the MPL on response time. The average response time decreases somewhat when changing from batch scheduling (MPL 1) to coscheduling (MPL 2 or more), and then stays at about the same level. This improvement corresponds to an enhanced ability of the scheduler to keep less jobs waiting in the queue. Having more available slots, the scheduler can dispatch more jobs from the queue, which is particularly significant for short jobs: These can start running soon after their arrival time, complete relatively quickly, and clear the system. To confirm this claim, observe that the average response time for the shorter 500 jobs indeed decreases for higher MPLs, while that of the longer 500 jobs increases at a similar rate. Furthermore, the median response time (which is dominated by the shorter jobs), decreases monotonically. [7]

This effect becomes more pronounced when looking at the bounded slowdown (Fig. 6.8(b)). We can clearly see that the average slowdown shows a consistent and significant decrease as the MPL increases. This is especially pronounced for the shorter 500 jobs, that show a marked improvement in slowdown, especially when changing from MPL 1 to MPL 2.

Still, the improvement of these metrics as the MPL increases might seem relatively insignificant compared to our expectations and previous results. To better understand why this might be the case, we repeated these measurements, but with backfilling disabled. Figures 6.8(c) and 6.8(d) show the results of these experiments. Here we can observe a sharp improvement in both metrics when moving from batch to GS, and a steady improvement after that. The magnitude of the improvement is markedly higher than that of the previous set. This might be explained by the fact that when backfilling is used, we implicitly include some knowledge of the future, since we have estimates for job run times. In contrast, GS assumes no such knowledge and packs jobs solely by size. Our results indicate that some knowledge of the future (job estimates) and consequently, their use in scheduling decisions as employed by backfilling, renders the advantages of a higher MPL less pronounced. These results also agree with the simulated evaluations in [120].

Having the best overall performance, we use an MPL of 6 in the other sets of experiments, combined with backfilling.

6.4.3 Effect of Time Quantum

Another important factor that can have an effect on a scheduling system's performance and responsiveness is the time quantum. In Section 4.3.3.1 it was shown that STORM can effectively handle very small time quanta, in the order of magnitude of a few ms, for simple static workloads. This is not necessarily the case for a complex dynamic workload, and a relatively high MPL value, serving to increase the load on the system (more pending communication buffers, cache pressure, etc.).

For this set of experiments, we use again the same workload file with 1000 jobs and an average offered load of $\approx 74\%$. We ran the scheduler with different time quantum values, ranging from $2s$ down to $50ms$.

[7] Note that the mean response time of the 'short' half of the jobs is actually higher than the median response time of all 1000 jobs, a results which might seem counterintuitive. However the jobs are classifi ed to either the short or long group based on their FCFS execution time, and obviously some of the so-called 'short' jobs actually have response times much above the median in GS.

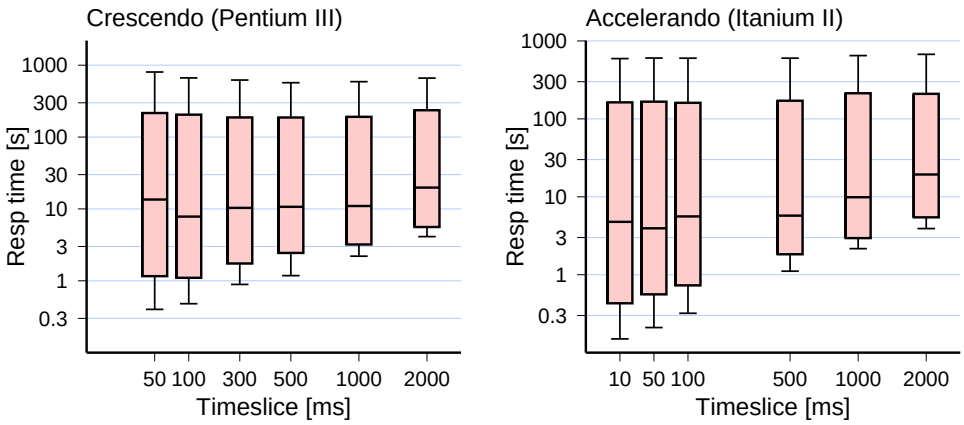


Figure 6.9: Response time distribution as a function of time quantum (log scale)

Since the overhead caused by short time slices is largely affected by the specific architecture, we were also interested in repeating these experiments on a different architecture. To that end, we also ran the same experiment on Accelerando.

Fig. 6.9 shows the distribution of response times with different time quanta, for both architectures. Each bar shows the median response time (central horizontal divider), the 25% and 75% percentiles (top and bottom edges of box), and the 5% and 95% percentiles (whiskers extending up and down). The 5% rank is defined by short jobs, and monotonically decreases with shorter time quanta, which confirms our expectations. The 95% rank represents all but the longest jobs, and does not change much over the quanta range, except for the $50ms$ quantum on Crescendo, where response times for most jobs increase slightly, presumably due to the overhead associated with frequent context switching. The different effect of the quantum on the 5% and 95% ranks suggests that short jobs are much more sensitive to changes in the time quantum than the rest of the jobs. The median reaches a minimum value at a time quantum of $\approx 100ms$ on Crescendo and $\approx 50ms$ on Accelerando. Running with shorter time quantum on Crescendo yields unreliable results, with degraded performance.

Fig. 6.10 shows the distribution of slowdown for the same time quanta values. The interpretation of this figure is *reversed*, since the 5% mark now represents mostly very long jobs (that have a low wait time to run time ratio, and thus a low slowdown value). On the other end, the 95% mark shows the high sensitivity of the slowdown metric to changes in the wait and run times of short jobs. Slowdown also seems to have a minimal median value at $\approx 100ms$ on Crescendo, and $50ms$ (or even $10ms$) on Accelerando. Based on these results, and since we run the other experiments on Crescendo, a time quantum of $100ms$ was chosen for the other measurements.

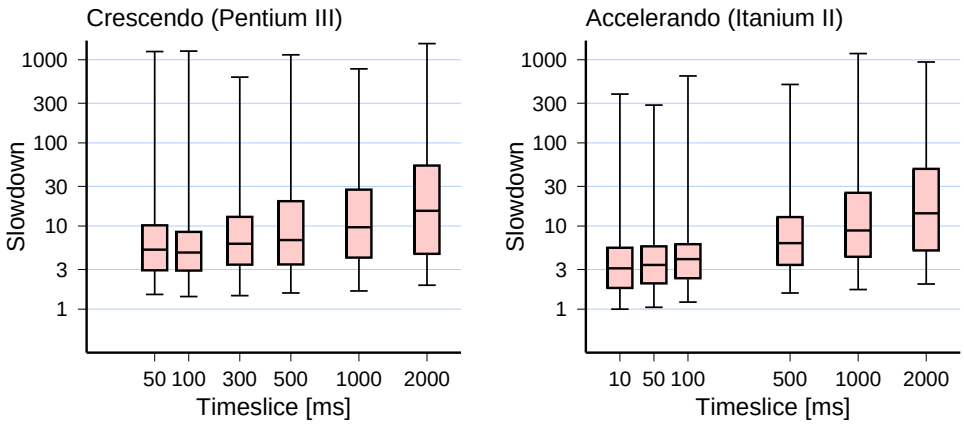


Figure 6.10: Bounded slowdown distribution as a function of time quantum (log scale)

6.4.4 Effect of Load

In this section we investigate the effect of load on different scheduling algorithms, and try to answer the following questions:

- How well do different algorithms handle increasing load?
- How do different scheduling algorithms handle short or long jobs?
- How does the dynamic workload affect the scheduler's performance?

When using finite workloads, one must be careful to identify when the offered load is actually high enough to saturate the system. Using an infinite workload, the job queues would keep on growing on a saturated system, and so will the average response time and slowdown. But when running a finite workload, the queues would only grow until the workload is exhausted, and then the queues would slowly clear since there are no more job arrivals. The metrics we measure for such a workload are therefore meaningless, and we should ignore them for loads that exceed each scheduler's saturation point.

To identify the saturation points, we used graphs like the one shown in Fig. 6.11. This figure shows jobs in the system over time, i.e. those jobs that arrived and are not yet finished, in this example using GS. It is easy to see that the system handles loads of 78% and 83% quite well. However, the burst of activity in the second half of the workload seems to cause problems when the load is increased to 88% or 93% of capacity. In particular, it seems that the system does not manage to clear enough jobs before the last arrival burst at about 7300 seconds. This indicates that the load is beyond the saturation point. Using this method, we identified and discarded those loads that saturate each scheduling scheme.

The results for this workload indicate that FCFS seems to saturate at about 78% offered load, GS and SB at about 83%, and FCS at 88%.

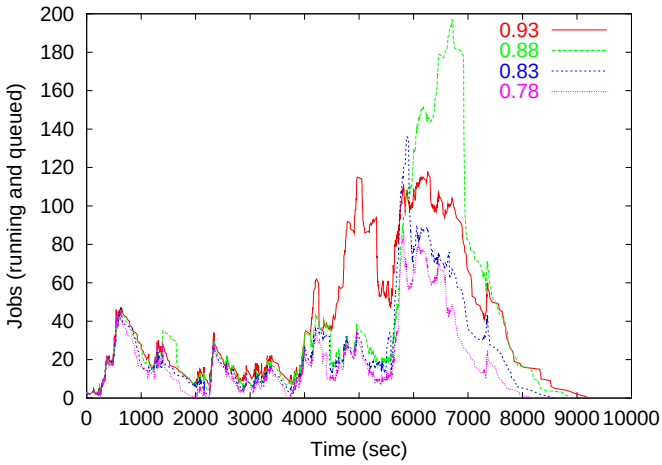


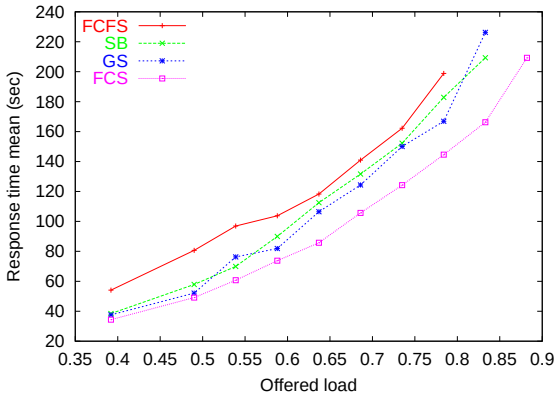
Figure 6.11: Job number in the system over time and different loads with GS

Figures 6.12(a) and 6.12(b) show the average response time and slowdown respectively, for different offered loads and scheduling algorithms. The near-linear growth in response times with load is due to our method of varying load, by multiplying run times of jobs by a load factor. Both metrics suggest that FCS seems to perform consistently better than the other algorithms, and FCFS (batch) seems to perform consistently worse than the others. Also, FCFS saturates at a lower load than the other algorithms, while FCS supports a load of up to 88% in this setup.

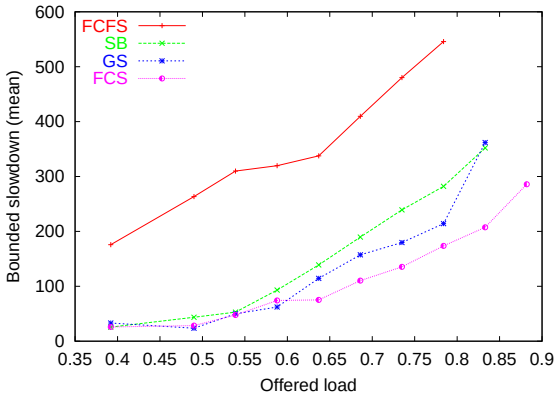
To understand the source of these differences, let us look at the median response time and slowdown (Figures 6.12(c) and 6.12(d) respectively). A low median response time suggests good handling of short jobs, since most jobs are comparatively short. On the other hand, a low median slowdown indicates preferential handling of long jobs, since the lowest-slowdown jobs are mostly long jobs, that are less affected by wait time than short jobs. FCFS shows a high average slowdown and a low median slowdown. This indicates that while long jobs enjoy lower waiting times (driving the median slowdown lower), short jobs suffer enough to significantly raise the average response time and slowdown.

To verify these biases, we look at the CDF of response times for the shorter 500 jobs and longer 500 jobs separately, as defined in Section 6.4.4 (Fig. 6.13). The higher distribution of short jobs with FCS attests to the scheduler's ability to "push" more jobs toward the shorter response times. Similarly, FCFS's preferential treatment of long jobs is reflected in Fig. 6.13(b).

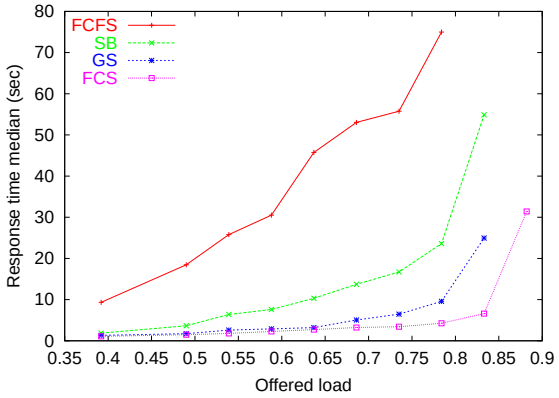
We believe the reason for FCS's good performance is its ability to adapt to various scenarios that occur during the execution of the dynamic workload [39]. In particular, FCS always coschedules a job in its first few seconds of running (unlike SB), and then classifies it according to its communication requirements (unlike GS). If a job is long, and does not synchronize frequently or effectively, FCS will allow other jobs to compete with it for machine resources. Thus, FCS shows a bias toward short jobs, allowing them to clear the system early. Since short jobs dominate the workload, this bias actually reduces the overall system load



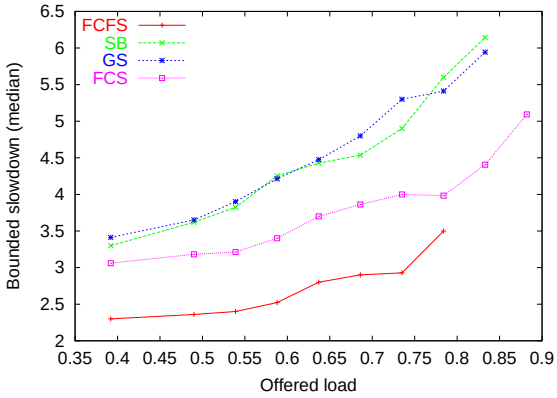
(a) Mean Response time



(b) Mean bounded slowdown

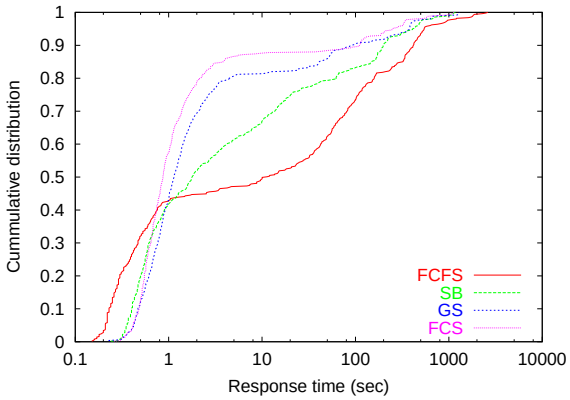


(c) Median response time

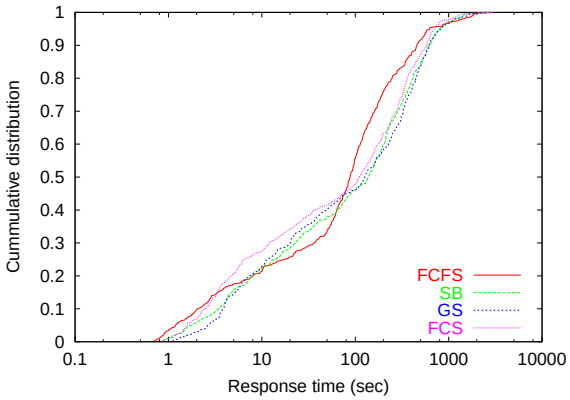


(d) Median bounded slowdown

Figure 6.12: Response time and bounded slowdown as a function of offered load



(a) 500 shortest jobs



(b) 500 longest jobs

Figure 6.13: Cumulative distribution of response times at 74%load and FCS scheduling.

and allows long jobs to complete earlier than with GS or SB. The opposite can be said of the FCFS scheme, which shows a bias toward long jobs, since they do not have to compete with other jobs.

6.5 Resource Overlapping

Resource overlapping refers in this context to the ability to improve resource utilization by exploiting "holes" in their usage by applications. Such holes occur naturally in virtually every parallel computer, and are caused by fragmentation in resource allocation and usage. Fragmentation can be manifested in any of the machine's resources, such as processors, memory, and network segments, and has two main forms.

External fragmentation occurs when not all of the available machine resources are allocated to a parallel application, and some remain unassigned. Such fragmentation is quite typical for machines designated as capacity engines, where many jobs of various sizes and lengths arrive dynamically at the system, making an optimal allocation of resources to jobs all but impossible.

Internal fragmentation occurs when an application does not fully use all its allocated resources consistently. This is in fact the case for almost every non-trivial computer application, including serial ones, and much effort has been put into increasing resource utilization by overlapping resources such as I/O, CPU, and network. Processes rarely make use of all of the available resources, and even single-node operating systems try to exploit this fragmentation by overlapping the use of different resources by different processes concurrently [107, 119]. The situation is exacerbated on parallel machines, since load imbalance and synchronization requirements often lead to even more holes in resource usage, as processes depend on continuous incoming data from their peers to perform their own computation.

We have seen the effect of external fragmentation in Section 6.4, where jobs of different sizes and lengths arrive dynamically. In general, multiprogramming scheduling algorithms are able to improve the system's utilization as a whole, compared to batch scheduling variants, since the added dimension of time gives the system software more opportunities to allocate physical resources to jobs, and fill more of the allocation holes. While resource utilization is increased, and resources are overlapped with context-switches among different jobs, this does not always imply that the system's throughput will be increased, or that users will be more satisfied. Multiprogramming algorithms increase the run time of applications by a factor that is approximately the average MPL they experience, and sometimes even much worse, as is the case with non-coordinated algorithms such as local and SB, for fine-grained parallel jobs. On the other hand, as shown in Section 6.4, by allowing small jobs to complete early, most multiprogramming algorithms can actually lower the average response time for most jobs [33, 40]. Furthermore, multiprogramming allows the single node's system software (in our case, the NM combined with Linux), to exploit intra-node holes in resource allocation, caused by internal fragmentation, which is typically higher for parallel jobs than it is for serial ones. Local operating systems can exploit such holes for resource overlapping relatively well. Parallel applications however make it much more difficult for the operating system, since if only local de-

cisions are employed, synchronous applications will not get coscheduled and consequently could undergo a significant performance hit.

For simple cases, described in Section 6.3.2, we can see that the dynamic scheduling algorithms (SB, FCS, BCS) can do relatively well in most scenarios, increasing utilization to near 100% in some cases. While these synthetic examples are not entirely representative of real-world scenarios, they nevertheless describe situations that can occur in various forms due to application-, workload-, or architecture-induced load imbalance.

Applications such as SAGE and SWEEP3D are much harder to overlap, due to their fine granularity and relatively fine-tuned balance (as seen in Section 6.3.3). That said, we believe there is still some significant underutilization of resources, in particular with SAGE, that can be exploited by BCS to increase the machine's throughput. While still maintaining explicit coscheduling, BCS spends no time at all waiting for synchronous communications (unlike the other algorithms), but rather immediately blocks the calling process, for at least one micro-timeslice. If another process can fill those holes effectively while the communication is taken place, BCS could theoretically run both applications at approximately the same time it takes to run the longest one in dedicated mode.

There are two prerequisites for such a successful overlap: first, there should be at least one application that is always ready to fill the holes, and a large enough amount of holes in the main application to amortize the cost of frequent context-switching. SAGE or SWEEP3D cannot be used as a good filler application, since they both communicate (and therefore block) very frequently. We have therefore used the same synthetic application from Section 6.3.1 with a relatively coarse communication granularity of $100ms$. We ran SAGE with the same input deck as in Section 6.3.3 with different numbers of processors and with the filler application. We used PBCS and prioritized SAGE, so that it runs whenever it is not blocked. The experiment consists of running SAGE in standalone mode in BCS (which yields results similar to those of Quadrics' MPI, as seen in 5.4.2), and then finding the maximum length of a filler application that *does not significantly increase the overall runtime*. For example, we found that for 62 processors, SAGE runs for $271s$ in dedicated mode, and we can concurrently run a low-priority filler application that runs for $58s$ without increasing SAGE's runtime by more than $1-3\%$.

Figure 6.14 shows the rest of the results. The blue bar represents the run time of SAGE, while the magenta bar represents the amount of filler that was squeezed into SAGE's computational holes. The total length of the stacked bars represent the time it would take to run both jobs in FCFS or GS mode. It is interesting to note that the relative size of the usable holes increases with the machine size, probably due to decreased SAGE scalability and increased load-imbalance. Unfortunately, Wolverine was unavailable to us at the time of writing for evaluating this property with even larger configurations. Still, if this trend is verified, running SAGE with other low-priority applications on large-scale machines such as ASCI-Q might significantly increase the throughput of the machine, by exploiting fine-grained resource overlapping.

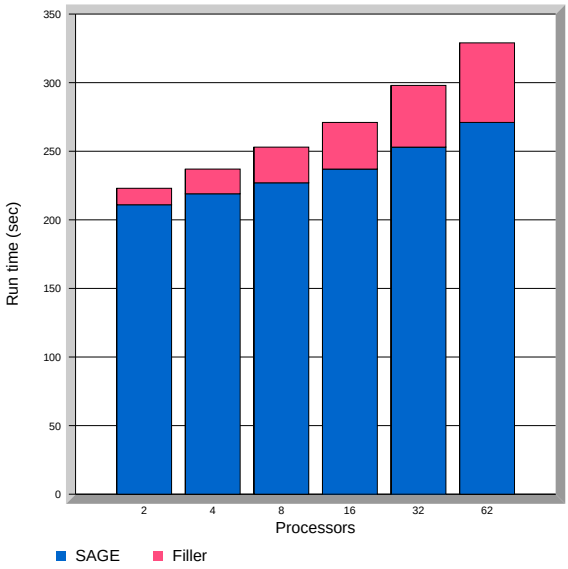


Figure 6.14: Run times of SAGE and filler application with BCS

7 Continuing and Future Work

As described in the introduction, system software encompasses many distinct issues. While many of the important issues are addressed in this dissertation, several others remain to be explored. We continue to develop the single system infrastructure model based on our network mechanisms. Our ongoing research focuses on two fields: resource overlapping and fault tolerance. In the near future we also plan to address several other remaining aspects of system software, such as I/O, quality of service, and debugging abilities. This section expands on some of these activities and in particular on fault tolerance, where our initial results show promise for an efficient and transparent system level checkpoint and recovery mechanism.

Fault Tolerance

Since commodity clusters are being used as high-productivity platforms, system availability and efficiency are increasingly important requirements. The high failure rate of these systems puts more pressure on checkpoint mechanisms. In order to meet these requirements, checkpoints should be taken frequently relative to the failure rate of the system, for example every few minutes. There is an inevitable need for autonomic computing systems which are able to self-heal and self-repair [71]. To achieve this vision, checkpoint and rollback recovery mechanisms must be automatic – they should work without the intervention of programmers, users, or system administrators. Our first objective in tackling this task is to answer the following question. Is it possible to implement an incremental checkpointing system that is (a) completely automatic and user transparent, (b) minimally intrusive, and (c) feasible with current and foreseeable I/O technology? We provide an extensive analysis of several scientific applications in this context in [98]. The following summarizes the main findings of this study.

to evaluate the feasibility of checkpointing, we needed to characterize scientific applications in terms of their memory footprint and rate of change, quantify the required bandwidth for transparent checkpointing, and analyze the applications' sensitivity to factors such as the checkpoint interval. To this end, we implemented a user-level instrumentation library. This library is preloaded by the dynamic linker, making application recompilation unnecessary. The library uses the memory protection mechanism of the virtual memory system to keep track the pages written to by the process during a timeslice. The library reports the bandwidth required to save the entire working set (full checkpointing), as well as the bandwidth required

for incremental checkpointing only [92]. This optimization saves only the data that has changed from the last checkpoint interval to stable storage. Using this library, we evaluated SAGE (with various data sizes), SWEEP3D, and several of the NAS applications on the Accelerando cluster. We have also evaluated the intrusiveness of this measurement by calculating the application slowdown with and without monitoring, and found it to be less than 10% in the worst case.

After initializing memory structures at the start of execution, all the tested applications display a periodic behavior in their incremental working set size (IWS), i.e. the changed subset of the working set for a given timeslice. This periodicity stems from the BSP nature of the applications. It also implies that the applications' communication is bursty rather than continuous, allowing time for the checkpointing traffic in between bursts of application traffic. In fact, the maximum application bandwidth observed with this architecture and data sets does not exceed $34MB/s$ (for NAS's FT), and most applications average less than $10MB/s$ (Table 7.1). This is far below the nominal network MPI bandwidth of $\approx 340MB/s$. The bandwidth required to transfer the changed application data (the IWS) can be significantly higher, especially for small timeslice values, but falls below $20MB/s$ for a timeslice of $10s$ for most applications, as much of the same memory is overwritten in the same timeslice. The choice of timeslice determines how far back in time a system must be restored in case of failure, and is typically in the order of magnitude of hours on current installations. Our data suggest that current network technology combined with smart incremental checkpointing can reduce this interval to tens of seconds or less, significantly shortening the mean time to recovery. Even in the worst case tested, SAGE with a data set of 1GB and a timeslice of $1s$, the overall required checkpointing bandwidth, $275MB/s$ (Table 7.1), falls below contemporary network and storage devices' bandwidth[1]. Furthermore, we found that as either the data set or the number of processors used by the application is increased, both the checkpoint and application bandwidth figure per node decrease, due to the sub-linear scaling of the application. Technological trends suggest that memory sizes, network performance and processor count will continue to grow at a fast rate - faster than main memory's performance growth. These trends indicate that while already viable today, incremental checkpointing will become even more attainable in the future.

Performance and Usability

We believe that resource overlapping approaches such as ours still hold significant unfulfilled potential for increasing system throughput. We are currently working on a comprehensive model and set of metrics to evaluate precisely the potential of different applications for resource overlapping in a parallel system. This model will take into account aspects that are particular to HPC systems and applications, such as network usage, synchronization and load balancing, and memory footprint. The latter in particular may be regarded as a limiting factor for multiprogramming since many HPC applications demand much of a

[1]These bandwidth numbers increase linearly for SMPs, if more than one process per node is running. For example, on a 4-way node the maximum checkpoint bandwidth can far exceed that of a single QsNet NIC, but can probably still be accomodated with multiple rails or with QsNet II's bandwidth of over $900MB/s$.

Application	Application bandwidth		Checkpoint bandwidth	
	Maximum	Average	Maximum	Average
SAGE-1000MB	17.1	4.1	274.9	78.8
SAGE-500MB	16.2	5.0	186.9	49.9
SAGE-100MB	21.8	8.6	42.6	15.0
SAGE-50MB	24.3	11.8	24.9	9.6
SWEEP3D	0.8	0.5	79.1	49.5
SP	17.5	9.7	32.6	32.6
LU	4.9	3.2	12.5	12.5
BT	5.9	3.1	72.7	68.6
FT	33.5	24.7	101.0	92.1

Table 7.1: Network bandwidth for various applications and data sets in MB/s, $1s$ timeslice

system's physical memory. We believe however that various measures can be taken to relax this requirement, and in particular, applications can be "stretched" to use more nodes with a smaller memory footprint on each node[2]. Such solutions are currently not practical, due to issues of reliability, load-imbalance and utilization. However, we believe that a comprehensive solution to all these issues as the vision put forth in this dissertation makes such an approach both feasible and highly efficient.

Quality of service in parallel systems' network traffic can also benefit from the deterministic, globally-synchronized view of the system. Some special-purpose systems such as BG/L simply use multiple separate networks for different communication needs [49], but that is rarely the case with COTS clusters. By controlling communication as in BCS-MPI, we can employ different communication-scheduling algorithms to ensure that QoS and priorities are maintained on a single interconnect. As one example, we can separate system and user traffic, prioritizing the former so that the operating system does not become unresponsive when user applications might otherwise congest the network. Another important aspect of HPC software is parallel I/O, which involves many challenging performance and load-balancing issues. We believe that inter-node I/O should be treated just as any other user- or system network messages, for example using MPI-2's I/O primitives [46]. As such, the same communication scheduling and QoS mechanisms apply to I/O traffic, to prevent deteriorating performance when the I/O load peaks.

Similarly, the precise controlling of communication and process scheduling globally, and the determinism it implies, offers another promising venue for research, namely application development. The ability to "freeze" applications and communication and analyze their state at fine resolutions could prove extremely useful to the development of parallel applications. Furthermore, the increased determinism should simplify the reproduction of bugs and race conditions that are common in these applications.

Last but not least, since we envision a simple, global cluster OS, we plan to migrate our code into the Linux kernel. We expect this to significantly reduce the overhead observed in some scenarios of Chapter 6, as well as simplify the entire design. Conceptually, this would allow the kernel to see the entire job and communication scheduling picture, instead of user-level dæmons with the current implementation.

[2]Many supercomputing installations rarely use partitions that are larger than half the total number of processors.

8 Concluding Remarks

The premise behind this dissertation is that virtually all system software tasks on a large-scale system represent an HPC application with global synchronization requirements. When treated as such, and using a basic set of modern network mechanisms, all aspects of parallel system software can be advanced to a new level of performance and scalability. In this section we briefly describe the major research contributions of this dissertation to each of the studied aspects, as well as the preliminary interconnection studies.

8.1 Summary of Research Contributions

8.1.1 Interconnection Scalability Analysis

We have analyzed the communication performance and scalability of advanced interconnects by using the QsNet network as case study. This network is widely used in many of the top 500 supercomputers and offers advanced architectural features such as its own on-board memory and processor, and support for collective communication in hardware. In particular, QsNet was shown to provide excellent latency and bandwidth for two important collective operations: global comparison/synchronization and multicast – $2.2\mu s$ and over $300MB/s$ respectively on our 256-processor cluster. Several other advanced networks show similar performance and scalability. In contrast, the point-to-point counterparts of these operations do not scale as well. We argue that while these collective operations are basic enough to expose the performance of the novel interconnects, they are nevertheless general enough to provide a vocabulary of communication primitives to be used by most system software tasks.

8.1.2 Network Mechanisms for System Software

In this dissertation we propose a new abstraction layer for large-scale clusters. This layer, which can be implemented by as few as three communication primitives in the network hardware, can immensely simplify the development of system software for these clusters. In our model, the system software is a tightly-coupled parallel application that operates in lockstep on all nodes. If the underlying hardware support for this layer is both scalable and efficient, the system software inherits these properties. Such

software is not only relatively simple to implement, but can also provide parallel programs with most of the services they require to make their development and usage efficient and more manageable. In particular, we discuss how this abstraction layer and the system software can be used for the implementation of efficient, deterministic communication libraries, workstation-class responsiveness and transparent fault tolerance. Our experimental results demonstrate that scalable resource management and application communication are indeed feasible while making the system behave deterministically.

8.1.3 Resource Management

While resource management (RM) is comparatively simple to do well on a small-scale cluster, it is more challenging on a large-scale cluster. Current RM systems require many seconds to launch a large application, and cannot react with user input with small response times. They either batch-schedule jobs—precluding interactivity—or gang-schedule them with such large quanta as to be effectively non-interactive. And they make poor use of resources, because large jobs frequently suffer from internal load imbalance or imperfect overlap of communication and computation, yet scheduling decisions are too costly to warrant lending unused resources to alternate jobs. These problems are addressed with STORM, a lightweight, flexible, and scalable environment for performing RM in large-scale clusters. In terms of both job launching and process scheduling, STORM is 1–2 orders of magnitude faster than the best reported results in the literature. The key to STORM's performance lies in its design methodology. Rather than implement heartbeat issuance, job launching, process scheduling, and other routines as separate entities, we designed these functions in terms of the small set of network mechanisms discussed above. We validated STORM's performance on a 256-processor cluster and demonstrated it to perform well on that cluster, and it is expected to perform comparably well on significantly larger clusters.

An important conclusion of our work is that it is indeed possible to scale up a cluster without sacrificing fast job-launching times, machine efficiency, or interactive response time. STORM can launch parallel jobs on a large-scale cluster almost as fast as a node OS can launch a sequential application on an individual workstation. And STORM can schedule all of the processes in a large, parallel job with the same granularity and with almost the same low overhead at which a sequential OS can schedule a single process. By improving the performance of various RM functions by two orders of magnitude, STORM represents an important step toward making large-scale clusters as efficient and easy to use as a workstation.

8.1.4 Communication and Fault Tolerance

This dissertation discusses an alternative approach to the design of communication libraries for large-scale parallel computers. The emphasis is moved to the optimization of the *global* state of the machine in order to reduce the system software complexity. We have provided insight on the global coordination protocols used by BCS-MPI and described a prototype implementation running almost entirely on the network interface

of the QsNet network.

The experimental results have shown that the performance of BCS-MPI is comparable to the production-level MPI for most applications. The performance of some applications, as SWEEP3D, can be improved by modifying their communication pattern from a blocking one to a non-blocking one (typically with minimal changes). Such applications can actually improve their performance when compared to the production level MPI, thanks to BCS-MPI's low overhead in the compute nodes.

These results pave the way to future advances in the design of the communication libraries for large-scale parallel machines, as part of the vision of a globally synchronized operating system. Such system, which relies on scalable network primitives, schedules globally not only user-level communication, but also I/O activities, resource management information exchange, and job scheduling. We argue that such a design not only simplifies system software, but also improves its performance and scalability. Moreover, a scheduled, deterministic communication behavior at system level could provide a solid infrastructure for implementing transparent fault tolerance. Our initial results demonstrate that frequent, user-transparent, automatic incremental checkpointing is a viable technique. We also prove that this can be achieved without using specialized hardware and within the limitations imposed by current technology. We anticipate that technological trends in networking and secondary storage will make these methods is more feasible.

8.1.5 Job Scheduling

The bulk of this thesis concentrates on job scheduling algorithms. We have implemented two novel such algorithms FCS and BCS, and several traditional ones (FCFS, GS, SB, and backfilling), and evaluated them in a comprehensive set of experiments. Unlike previous scheduling approaches that either rely solely on global coordination or on local decisions, FCS and BCS make use of advanced network features and performance to provide both global coscheduling for highly-synchronized jobs, and adaptiveness for imbalanced jobs. In synthetic scenarios, this translates to an increased capacity of FCS to provide the best scheduling for a variety of balanced and imbalanced situations, and to BCS's ability to make use of small internal fragmentation for increased resource utilization, even with highly-parallel jobs.

This work emphasizes realistic evaluation as a critical factor in the complex field of parallel job scheduling. This translates in our case not only to a complete implementation of the various algorithms on several cluster architectures, but also to the use of dynamic, complex workloads running communicating applications over a considerable amount of time. Through these experiments, we were able to gain insights into the differences between job scheduling algorithms, and offer several explanations on the relationships between workloads, scheduling algorithms, job types, and metrics. Some more general observations were also obtained, such as the benefits of backfilling for all job scheduling algorithms, and the effect of the multiprogramming level. The dynamic workload also confirmed the advantages of using novel scheduling techniques such as FCS, that make use scalably both of local and global information.

Glossary

AA	All-to-All (communication pattern)
API	Application Programmer's Interface
BCS	Buffered Coscheduling
BSP	Bulk-Synchronous Parallel
CMS	Cluster Management System
Coscheduling	Scheduling all processes of a job concurrently
COTS	Commercial-of-the-shelf
DCS	Dynamic Coscheduling
DMA	Direct Memory Access
ECC	Error Correcting Code
FCS	Flexible CoScheduling
FCFS	First-Come-First-Serve
FCFS-BF	First-Come-First-Serve with backfilling
Flit	Flow control digit
GS	Gang Scheduling
GUI	Graphical User Interface
HPC	High-Performance Computing
ICN	Interconnection Network
ICS	Implicit Coscheduling
I/O	Input/Output
IWS	Incremental Working Set
Job	A parallel program possibly consisting of multiple processes
LANL	Los Alamos National Laboratory
LSF	Load Sharing Facility
MM	Machine Manager (STORM dæmon)

MMU	Memory Management Unit
MPI	Message Passing Interface library
MPL	Multi-Programming Level
MPP	Massively Parallel Processing
NIC	Network Interface Card
NN	Nearest-Neighbour (communication pattern)
NM	Node Manager (STORM dæmon)
NOW	Network of Workstations
NQS	Network Queueing System
OS	Operating System
PB	Periodic Boost
PBS	Portable Batch System
PE	Processing Element
PID	Process Identifier
PRNG	Pseudo-Random Number Generator
Process	A computation using a single PE
QoS	Quality of Service
RAM	Random Access Memorys
RDMA	Remote Direct Memory Access
RMS	Resource Management System
SB	Spin-block
SIMD	Single Instruction Multiple Data
SMP	Symmetrical Multi Processing
STORM	Scalable TOol for Resource Management
PL	Program Launcher (STORM dæmon)
VPID	Virtual Process ID

Bibliography

[1] ASCI Technology Prospectus: Simulation and Computational Science. Technical Report DOE/DP/ASC-ATP-001, National Nuclear Security Agency (NNSA), July 2001.

[2] Andrea Carol Arpaci-Dusseau. Implicit Coscheduling: Coordinated Scheduling with Implicit Information in Distributed Systems. *ACM Transactions on Computer Systems (TOCS)*, 19(3), 2001.

[3] Infiniband Trade Association. Infiniband specification 1.0a, June 2001. Available from `http://www.infinibandta.org`.

[4] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[5] Mark A. Baker, Geoffrey C. Fox, and Hon W. Yau. Cluster computing review. NPAC Technical Report SCCS-748, Syracuse University, November 1995.

[6] Anindya Basu, Vineet Buch, Werner Vogels, and Thorsten von Eicken. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95)*, Copper Mountain, CO, December 1995. Available from `http://www.cs.cornell.edu/tve/u-net/papers/sosp.pdf`.

[7] Anat Batat and Dror G. Feitelson. Gang scheduling with memory considerations. In *Proceedings of the International Parallel and Distributed Processing Symposium 2000, IPDPS2000*, number 14, pages 109–114, May 2000.

[8] Raoul A.F. Bhoedjang, Tim Rühl, and Henri E. Bal. Efficient multicast on Myrinet using link-level flow control. In *Proceedings of the* 27^{th} *International Conference on Parallel Processing (ICPP'98)*, pages 381–390, Minneapolis, MN, August 1998. Available from `ftp://ftp.cs.vu.nl/pub/raoul/papers/multicast98.ps.gz`.

[9] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawick, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995. Available from `http://www.myri.com/research/publications/Hot.ps`.

[10] George Bosilca, Aurelien Bouteiller, Franck Cappello, Samir Djailali, Gilles Fedak, Cecile Germain, Thomas Herault, Pierre Lemarinier, Oleg Lodygensky, Frederic Magniette, Vincent Neri, and Anton Selikhov. MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes. In *Proceedings of IEEE/ACM Supercomputing 2002 (SC'02)*, Baltimore, MD, November 2002. Available from `http://www.sc2002.org/paperpdfs/pap.pap298.pdf`.

[11] Ron Brightwell and Lee Ann Fisk. Scalable Parallel Application Launch on Cplant. In *Supercomputing 2001*, Denver, CO, November 2001.

[12] Darius Buntinas, Dhabaleswar Panda, José Duato, and P. Sadayappan. Broadcast/multicast over Myrinet using NIC-assisted multidestination messages. In *Workshop on Communication, Architecture, and Applications for Network-Based Parallel Computing (CANPC '00)*,

High Performance Computer Architecture (HPCA-6) Conference, Toulouse, France, January 2000. Available from `ftp://ftp.cis.ohio-state.edu/pub/communication/papers/canpc00-nic-multi\%cast.pdf`.

[13] Darius Buntinas, Dhabaleswar Panda, and William Gropp. NIC-based atomic operations on Myrinet/GM. In *SAN-1 Workshop, High Performance Computer Architecture (HPCA-8) Conference*, Boston, MA, February 2002. Available from `ftp://ftp.cis.ohio-state.edu/pub/communication/papers/san-1-atomic_oper\%ations.pdf`.

[14] Helen Chen and Pete Wyckoff. Simulation studies of Gigabit ethernet versus Myrinet using real application cores. In *Proceedings of CANPC'00, Workshop of High-Performance Computer Architecture*, Toulouse, France, January 2000.

[15] Giovanni Chiola and Giuseppe Ciaccio. GAMMA: a Low-cost Network of Workstations Based on Active Messages. In *Proceedings of 5th EUROMICRO workshop on Parallel and Distributed Processing (PDP'97)*, London, UK, January 1997. Available from `ftp://ftp.disi.unige.it/pub/project/GAMMA/pdp97.ps.gz`.

[16] Salvador Coll, Eitan Frachtenberg, Fabrizio Petrini, Adolfy Hoisie, and Leonid Gurvits. Using Multirail Networks in High-Performance Clusters. In *Proceedings of the IEEE Conference on Cluster Computing*, Newport Beach, CA, October 2001.

[17] Salvador Coll, Eitan Frachtenberg, Fabrizio Petrini, Adolfy Hoisie, and Leonid Gurvits. Using Multirail Networks in High-Performance Clusters. *Concurrency and Computation: Practice and Experience*, 15(7-8):625–651, April 2003.

[18] Compaq, Intel, and Microsoft. The Virtual Interface Architecture (VIA) Specification. Available Available from `http://www.viarch.org`.

[19] Cray Research Inc. *Cray T3D System Architecture Overview*, 1^{th} edition, September 1993.

[20] David E. Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993.

[21] William J. Dally and Charles L. Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Transactions on Computers*, C-36(5):547–553, May 1987.

[22] Dror G. Feitelson and Larry Rudolph. Gang Scheduling Performance Benefits for Fine-Grain Synchronization. *Journal of Parallel and Distributed Computing*, 16(4), 1992.

[23] José Duato, Sudhakar Yalamanchili, and Lionel Ni. *Interconnection Networks: an Engineering Approach*. IEEE Computer Society Press, 1997.

[24] Yoav Etsion and Dror G. Feitelson. User-Level Communication in a System with Gang Scheduling. In *Proceedings of the International Parallel and Distributed Processing Symposium 2001, IPDPS2001*, San Francisco, CA, April 2001.

[25] Fabrizio Petrini and Wu-chun Feng. Scheduling with Global Information in Distributed Systems. In *Proceedings of the The 20th International Conference on Distributed Computing Systems*, Taipei, Taiwan, Republic of China, April 2000.

[26] Fabrizio Petrini and Wu-chun Feng. Time-Sharing Parallel Jobs in the Presence of Multiple Resource Requirements. In *6th Workshop on Job Scheduling Strategies for Parallel Processing*, Cancun, MX, May 2000.

[27] Dror G. Feitelson. Packing Schemes for Gang Scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing – Proceedings of the IPPS'96 Workshop*, volume 1162, pages 89–110. Springer, 1996.

[28] Dror G. Feitelson. Job scheduling in multiprogrammed parallel systems. Research Report RC 19970, IBM T. J. Watson Research Center, Yorktown Heights, NY, 1997. Second Revision.

[29] Dror G. Feitelson. The Forgotten Factor: Facts; on Performance Evaluation and Its Dependence on Workloads. In Burkhard Monien and Rainer Feldmann, editors, *Euro-Par 2002 Parallel Processing*, pages 49–60. Springer-Verlag, Aug 2002. Lect. Notes Comput. Sci. vol. 2400.

[30] Dror G. Feitelson, Anat Batat, Gabriel Benhanokh, David Er-El, Yoav Etsion, Avi Kavas, Tomer Klainer, Uri Lublin, and Marc Volovic. The ParPar System: a Software MPP. In Rajkumar Buyya, editor, *High Performance Cluster Computing*, volume 1: Architectures and systems, pages 754–770. Prentice-Hall, 1999.

[31] Dror G. Feitelson and Morris A. Jette. Improved Utilization and Responsiveness with Gang Scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*, pages 238–261. Springer-Verlag, 1997.

[32] Dror G. Feitelson and Larry Rudolph. Evaluation of design choices for gang scheduling using distributed hierarchical control. *Journal of Parallel and Distributed Computing*, 35(1):18–34, May 1996.

[33] Dror G. Feitelson and Larry Rudolph. Metrics and benchmarking for parallel job scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1495 of *Lecture Notes in Computer Science*, pages 1–24. Springer-Verlag, 1998.

[34] Dror G. Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C. Sevcik, and Parkson Wong. Theory and Practice in Parallel Job Scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lect. Notes Comput. Sci.*, pages 1–34. Springer Verlag, 1997.

[35] Juan Fernandez, Eitan Frachtenberg, and Fabrizio Petrini. BCS-MPI: A New Approach in the System Software Design for Large-Scale Parallel Computers. In *Proceedings of the IEEE/ACM Conference on Supercomputing*, Phoenix, AZ, November 2003.

[36] Juan Fernandez, Eitan Frachtenberg, Fabrizio Petrini, and Jose Carlos Sancho. An Abstract Interface for System Software on Large-Scale Clusters. Submitted to Parallel Computing.

[37] Juan Fernandez, Eitan Frachtenberg, Fabrizio Petrini, and Jose Carlos Sancho. Architectural Support for System Software on Large-Scale Clusters. Submitted to IPDPS'04.

[38] Eitan Frachtenberg. Flexible coscheduling, December 2001. Master's thesis, Hebrew University, Jerusalem, Israel.

[39] Eitan Frachtenberg, Dror Feitelson, Fabrizio Petrini, and Juan Fernandez. Flexible CoScheduling: Dealing with Load Imbalance and Heterogeneous Resources. In *"Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS2003)"*, Nice, France, April 2003.

[40] Eitan Frachtenberg, Dror G. Feitelson, Juan Fernandez-Peinador, and Fabrizio Petrini. Parallel Job Scheduling under Dynamic Workloads. In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 2862 of *Lect. Notes Comput. Sci.*, pages 208–227. Springer Verlag, 2003.

[41] Eitan Frachtenberg and Fabrizio Petrini. Overlapping Communication and Computation in the Quadrics Network. Technical Report LAUR 01-4695, Los Alamos National Laboratory, August 2001.

[42] Eitan Frachtenberg, Fabrizio Petrini, Salvador Coll, and Wu chun Feng. Gang Scheduling with Lightweigth User-Level Communication. In *Proceedings of the International Conference on Parallel Processing (ICPP2001), Workshop on Scheduling and Resource Management for Cluster Computing*, Valencia, Spain, September 2001.

[43] Eitan Frachtenberg, Fabrizio Petrini, Juan Fernandez, and Scott Pakin. STORM: Scalable resource management for large-scale parallel computers. *ACM Transactions on Computer Systems (TOCS)*, 2003. Submitted for publication.

[44] Eitan Frachtenberg, Fabrizio Petrini, Juan Fernandez, Scott Pakin, and Salvador Coll. STORM: Lightning-Fast Resource Management. In *Proceedings of the IEEE/ACM Conference on Supercomputing*, Baltimore, MD, November 2002.

[45] Hubertus Franke, Pratap Pattnaik, and Larry Rudolph. Gang Scheduling for Highly Efficient Distributed Multiprocessor Syetems. In *6th Symposium on the Frontiers of Massively Parallel Computation (FRONTIERS '96)*, pages 1–9, Annapolis, MD, October 1996.

[46] Al Geist, William Gropp, Steve Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, William Saphir, Tony Skjellum, and Marc Snir. MPI-2: Extending the Message Passing Interface. In *Second International Euro-Par Conference, Volume I*, number 1123 in LNCS, pages 128–135, Lyon, France, August 1996.

[47] Douglas P. Ghormley, David Petrou, Steven H. Rodrigues, and Amin M. Vadhar. GLUnix: a GLobal Layer Unix for a Network of Workstations. *Software - Practice and Experience*, 28(9), 1998.

[48] Anoop Gupta, Andrew Tucker, and Shigeru Urushibara. The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications. In *Proceedings of the 1991 ACM SIGMETRICS Conference*, pages 120–132, May 1991.

[49] Manish Gupta. Challenges in developing scalable scalable software for bluegene/l. In *Scaling to New Heights Workshop*, Pittsburgh, PA, May 2002. Available from `http://www.psc.edu/training/scaling/gupta.ps`.

[50] Hermann Hellwagner. The SCI Standard and Applications of SCI. In Hermann Hellwagner and Alexander Reinfeld, editors, *SCI: Scalable Coherent Interface*, volume 1291 of *Lecture Notes in Computer Science*, pages 95–116. Springer-Verlag, 1999.

[51] R. L. Henderson. Job scheduling under the portable batch system. *Lecture Notes in Computer Science*, 949:279–294, 1995.

[52] Erik Hendriks. BProc: The Beowulf distributed process space. In *Proceedings of the* 16^{th} *Annual ACM International Conference on Supercomputing (ICS '02)*, New York, NY, June 22–26, 2002.

[53] Adolfy Hoisie, Olaf Lubeck, and Harvey Wasserman. Scalability Analysis of Multidimensional Wavefront Algorithms on Large-Scale SMP Clusters. In *The Ninth Symposium on the Frontiers of Massively Parallel Computation (Frontiers'99)*, Annapolis, MD, February 1999.

[54] Atsushi Hori, Hiroshi Tezuka, and Yukata Ishikawa. Overhead Analysis of Preemptive Gang Scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1459 of *Lecture Notes in Computer Science*, pages 217–230. Springer-Verlag, 1998.

[55] Atsushi Hori, Hiroshi Tezuka, and Yutaka Ishikawa. Highly Efficient Gang Scheduling Implementation. In *Supercomputing 98*, Orlando, FL, November 1998.

[56] Atsushi Hori, Hiroshi Tezuka, Yutaka Ishikawa, NoriYuki Soda, Hiroki Konaka, and Muneori Maeda. Overhead Analysis of Preemptive Gang Scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1459 of *Lecture Notes in Computer Science*, pages 217–230. Springer-Verlag, 1998.

[57] Morris A. Jette, Andy B. Yoo, and Mark Grondona. SLURM: Simple linux utility for resource management. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 37–51. Springer-Verlag, 2003.

[58] Tomio Kamada, Satoshi Matsuoka, and Akinori Yonezawa. Efficient parallel global garbage collection on massively parallel computers. In G. M. Johnson, editor, *Proceedings of IEEE/ACM Supercomputing 1994 (SC'94)*, pages 79–88, 1994. Available from `http://citeseer.nj.nec.com/kamada94efficient.html`.

[59] Avi Kavas, David Er-El, and Dror G. Feitelson. Using Multicast to Pre-Load Jobs on the ParPar Cluster. *Parallel Computing*, 27:315–327, 2001.

[60] Avi Kavas and Dror G. Feitelson. Comparing Windows NT, Linux, and QNX as the basis for cluster systems. *Concurrency & Comput.: Pract. & Exp.*, 13(15):1303–1332, Dec 2001.

[61] Darren Kerbyson, Hank Alme, Adolfy Hoisie, Fabrizio Petrini, Harvey Wasserman, and Mike Gittings. Predictive Performance and Scalability Modeling of a Large-Scale Application. In *IEEE/ACM SC2001*, Denver, CO, November 2001.

[62] JunSeong Kim and David J. Lilja. Characterization of communication patterns in message-passing parallel scientific application programs. In *Proceedings of the Workshop on Communication, Architecture, and Applications for Network-Based Parallel Computing (CANPC)*, pages 202–216, Las Vegas, NV, February 1998.

[63] Ken Koch. How does ASCI actually complete multi-month 1000-processor milestone simulations? In *Proceedings of the Conference on High Speed Computing*, Gleneden Beach, Oregon, April 22–25, 2002.

[64] William T. C. Kramer and James M. Craw. Effective use of cray supercomputers. In *Proceedings of the Supercomputing 89*, pages 721–731, New York, NY, 1989. ACM Press.

[65] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[66] Walter Lee, Matthew Frank, Victor Lee, Kenneth Mackenzie, and Larry Rudolph. Implications of I/O for Gang Scheduled Workloads. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[67] Charles E. Leiserson. Fat-Trees: Universal Networks for Hardware Efficient Supercomputing. *IEEE Transactions on Computers*, C-34(10):892–901, October 1985.

[68] David Lifka. The ANL/IBM SP Scheduling System. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 295–303. Springer-Verlag, 1995. Lect. Notes Comput. Sci. vol. 949.

[69] Jiuxing Liu, Balasubramanian Chandrasekaran, Jiesheng Wu, Weihang Jiang, Sushmitha Kini, Weikuan Yu, Darius Buntinas, Pete Wyckoff, and Dhabaleswar K. Panda. Performance Comparison of MPI Implementations over InfiniBand, Myrinet and Quadrics. In *Proceedings of the IEEE/ACM Conference on Supercomputing*, Phoenix, AZ, November 2003.

[70] Uri Lublin and Dror G. Feitelson. The workload on parallel supercomputers: Modeling the characteristics of rigid jobs. *Journal of Parallel and Distributed Computing*, 63(11):1105–1122, Nov 2003.

[71] E. Mainsah. Autonomic Computing: the Next Era of Computing. *IEEE Electronics Communication Engineering Journal*, 14(1):2–3, February 2002.

[72] Meiko world Inc. *Meiko Computing Surface Communications Processor Overview*, 1993.

[73] Adam Moody, Juan Fernández, Fabrizio Petrini, and Dhabaleswar Panda. Scalable NIC-based Reduction on Large-scale Clusters. In *Proceedings of SC2003*, Phoenix, Arizona, November 10–16, 2003. Available from `http://www.c3.lanl.gov/~fabrizio/papers/sc03_reduce.pdf`.

[74] Jose E. Moreira, Waiman Chan, Liana L. Fong, Hubertus Franke, and Morris A. Jette. An Infrastructure for Efficient Parallel Job Execution in Terascale Computing Environments. In *Supercomputing'98*, Nov 1998.

[75] Ahuva W. Mu'alem and Dror G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. *IEEE Transactions on Parallel and Distributed Systems*, 12(6):529–543, June 2001.

[76] Shailabh Nagar, Ajit Banerjee, Anand Sivasubramaniam, and Chita R. Das. A Closer Look At Coscheduling Approaches for a Network of Workstations. In *Eleventh ACM Symposium on Parallel Algorithms and Architectures, SPAA'99*, Saint-Malo, France, June 1999.

[77] Lionel M. Ni, Yadong Gui, and Sherry Moore. Performance Evaluation of Switch-Based Wormhole Networks. *IEEE Transactions on Parallel and Distributed Systems*, 8(5):462–474, May 1997.

[78] J. K. Ousterhout. Scheduling Techniques for Concurrent Systems. *Proceedings of Third International Conference on Distributed Computing Systems*, pages 22–30, 1982.

[79] Scott Pakin, Mario Lauria, and Andrew Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Proceedings of IEEE/ACM Supercomputing 1995 (SC'95)*, San Diego, CA, December 1995. Available from `http://www.supercomp.org/sc95/proceedings/567_SPAK/SC95.PDF`.

[80] Fabrizio Petrini and Wu chun Feng. Buffered coscheduling: A new methodology for multitasking parallel jobs on distributed systems. In *Proceedings of the International Parallel and Distributed Processing Symposium 2000, IPDPS2000*, volume 16, Cancun, MX, May 2000.

[81] Fabrizio Petrini and Wu chun Feng. Improved Resource Utilization with Buffered Coscheduling. *Journal of Parallel Algorithms and Applications*, 2000.

[82] Fabrizio Petrini, Wu chun Feng, Adolfy Hoisie, Salvador Coll, and Eitan Frachtenberg. Quadrics Network (QsNet): High-Performance Clustering Technology. In *Hot Interconnects 9*, Stanford University, Palo Alto, CA, August 2001.

[83] Fabrizio Petrini, Wu chun Feng, Adolfy Hoisie, Salvador Coll, and Eitan Frachtenberg. The Quadrics Network: High Performance Clustering Technology. *IEEE Micro*, 22(1):46–57, January-February 2002.

[84] Fabrizio Petrini, Salvador Coll, Juan Fernandez, and Eitan Frachtenberg. Scalable Collective Communication on the ASCI Q Machine. In *10th Hot Interconnects conference*, Stanford University, Palo Alto, CA, August 2003.

[85] Fabrizio Petrini, Salvador Coll, Eitan Frachtenberg, and Adolfy Hoisie. Hardware- and Software-Based Collective Communication on the Quadrics Network'. In *IEEE International Symposium on Network Computing and Applications (NCA 2001)*, Boston, MA, October 2001.

[86] Fabrizio Petrini, Salvador Coll, Eitan Frachtenberg, and Adolfy Hoisie. Performance Evaluation of I/O Traffic and Placement of I/O Nodes on a High Performance Network. In *Workshop on Communication Architecture for Clusters 2002 (CAC '02), International Parallel and Distributed Processing Symposium 2002 (IPDPS '02)*, Fort Lauderdale, FL, April 2002.

[87] Fabrizio Petrini, Eitan Frachtenberg, Adolfy Hoisie, and Salvador Coll. Performance Evaluation of the Quadrics Interconnection Network. *Journal of Cluster Computing*, 6(2):125–142, April 2003.

[88] Fabrizio Petrini, Darren Kerbyson, and Scott Pakin. The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q. In *Proceedings of the IEEE/ACM Conference on Supercompting*, Phoenix, Arizona, November 10–16, 2003. Available from `http://www.c3.lanl.gov/~fabrizio/papers/sc03_noise.pdf`.

[89] Fabrizio Petrini and Marco Vanneschi. k-ary n-trees: High Performance Networks for Massively Parallel Architectures. In *Proceedings of the 11th International Parallel Processing Symposium, IPPS'97*, pages 87–93, Geneva, Switzerland, April 1997.

[90] Fabrizio Petrini and Marco Vanneschi. Performance Analysis of Wormhole Routed k-ary n-trees. *International Journal on Foundations of Computer Science*, 9(2):157–177, June 1998.

[91] Gregory F. Pfister and V. Alan Norton. "Hot Spot" Contention and Combining in Multistage Interconnection Networks. *IEEE Transactions on Computers*, C-34(10):943–948, October 1985.

[92] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent Checkpointing under Unix. In *Proceedings of the Usenix Winter 1995 Technical Conference*, New Orleans, Louisiana, January 16–20, 1995. Available from `http://www.cs.utk.edu/~plank/plank/papers/USENIX-95W.html`.

[93] Loïc Prylli and Bernard Tourancheau. BIP: A New Protocol Designed for High Performance Networking on Myrinet. In *Proceedings of IPPS/SPDP'98 Workshop on Personal Computer Based Networks of Workstations*, Orlando, FL, April 1998. Available from `http://ipdps.eece.unm.edu/1998/pc-now/prylli.pdf`.

[94] Quadrics Supercomputers World Ltd. *Elan Programming Manual*, January 1999.

[95] Quadrics Supercomputers World Ltd. *Elan Reference Manual*, January 1999.

[96] Quadrics Supercomputers World Ltd. *Elite Reference Manual*, November 1999.

[97] Rolf Riesen, Ron Brightwell, Lee Ann Fisk, Tramm Hudson, Jim Otto, and Arthur B. Maccabe. Cplant. In *Proceedings of the 1999 USENIX Annual Technical Conference, Second Extreme Linux Workshop*, Monterey, CA, June 6–11, 1999. Available from `http://www.cs.sandia.gov/~rolf/papers/extreme/cplant.ps.gz`.

[98] Jose Carlos Sancho, Fabrizio Petrini, Greg Johnson, Juan Fernandez, and Eitan Frachtenberg. On the Feasibility of Incremental Checkpointing for Scientific Computing. Submitted to IPDPS'04.

[99] Rich Seifert. *Gigabit Ethernet: Technology and Applications for High Speed LANs*. Addison-Wesley, May 1998.

[100] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. NFS version 4 protocol. RFC 3010, Internet Engineering Task Force, Network Working Group, December 2000. Available from `http://www.rfc-editor.org/rfc/rfc3010.txt`.

[101] Piyush Shivam, Pete Wyckoff, and Dhabaleswar Panda. EMP: Zero-copy os-bypass nic-driven gigabit ethernet message passing. In *Proceedings of IEEE/ACM Supercomputing 2001 (SC'01)*, Denver, CO, November 10–16, 2001. Available from `http://www.sc2001.org/papers/pap.pap315.pdf`.

[102] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI - The Complete Reference*, volume 1, The MPI Core. The MIT Press, 1998.

[103] Patrick Sobalvarro, Scott Pakin, William E. Weihl, and Andrew A. Chien. Dynamic Coscheduling on Workstation Clusters. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1459 of *Lecture Notes in Computer Science*, pages 231–256. Springer-Verlag, 1998.

[104] Patrick Sobalvarro and William E. Weihl. Demand-Based Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessors. In *Proceedings of the 9th International Parallel Processing Symposium, IPPS'95*, Santa Barbara, CA, April 1995.

[105] L. Kent Steiner. Evolution of supercomputers. In *ACM annual conference on The range of computing : mid-80's perspective*, pages 112–116, Denver, CO, October 1985. Assoc. for Computing Machinery, ACM PressNew York, NY, USA.

[106] T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer. BEOWULF: A parallel workstation for scientific computation. In *Proceedings of the 24th International Conference on Parallel Processing*, pages I:11–14, Oconomowoc, WI, 1995. Available from `http://citeseer.nj.nec.com/sterling95beowulf.html`.

[107] W. R. Stevens. *Advanced Programming in the Unix Environment*. Addison Wesley, June 1993.

[108] W. R. Stevens. *UNIX Network Programming*. Prentice Hall, 1997.

[109] Jeffrey H. Straathof, Ashok K. Thareja, and Ashok K. Agrawala. UNIX Scheduling for Large Systems. In *Proceedings of the USENIX Winter Conference*, pages 111–139, Denver, CO, 1986.

[110] V. S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[111] David Talby, Dror G. Feitelson, and Adi Raveh. Comparing Logs and Models of Parallel Workloads Using the Co-Plot Method. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 43–66. Springer Verlag, 1999. Lect. Notes Comput. Sci. vol. 1659.

[112] H. Tezuka, A. Hori, Y. Ishikawa, and M. Sato. PM: An Operating System Coordinated High-Performance Communication Library. In *Proceedings of High-Performance Computing and Networking '97*, pages 708–717, April 1997.

[113] Thinking Machines Corporation. *NI Systems Programming*, October 1992. Version 7.1.

[114] D. Tolmie, T. M. Boorman, A. DuBois, D. DuBois, W. Feng, and I. Philp. From HiPPI-800 to HiPPI-6400: A Changing of the Guard and Gateway to the Future. In *Proceedings of the 6th International Conference on Parallel Interconnects (PI'99)*, October 1999.

[115] Leslie G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[116] Jeffrey S. Vetter and Frank Mueller. Communication Characteristics of Large-Scale Scientific Applications for Contemporary Cluster Architectures. Fort Lauderdale, FL, April 2002. Available from `http://www.csc.ncsu.edu/faculty/mueller/ftp/pub/mueller/papers/jpdc02.p\%df`.

[117] Werner Vogels, David Follett, Jenwi Hsieh, David Lifka, and David Stern. Tree-Saturation Control in the AC^3 Velocity Cluster. In *Hot Interconnects 8*, Stanford University, Palo Alto CA, August 2000.

[118] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

[119] Carl A. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management*. PhD thesis, Massachusetts Institute of Technology, September 1995.

[120] Yanyong Zhang, Hubertus Franke, José Moreira, and Anand Sivasubramaniam. Improving Parallel Job Scheduling by Combining Gang Scheduling and Backfilling Techniques. In *Proceedings of the International Parallel and Distributed Processing Symposium 2000, IPDPS2000*, Cancun, MX, May 2000.

[121] Songnian Zhou. LSF: load sharing in large-scale heterogeneous distributed systems. In *Proceedings of the Workshop on Cluster Computing*, Tallahassee, FL, 1992.

[122] http://www.top500.org.

[123] http://www.nas.nasa.gov/Software/NPB/.

[124] http://www.quadrics.com.

[125] http://www.linux.org.

[126] http://www.openpbs.org.

[127] http://www.cs.huji.ac.il/labs/parallel/parpar.shtml.

[128] http://www.jhauser.us/arithmetic/SoftFloat.html.