

UNRESPONSIVENESS-TOLERANT COLLECTIVE COMMUNICATION

BY

SCOTT DOV PAKIN

B.S., Carnegie Mellon, 1992

M.S., University of Illinois at Urbana-Champaign, 1995

THESIS

Submitted in partial fulfillment of the requirements  
for the degree of Doctor of Philosophy in Computer Science  
in the Graduate College of the  
University of Illinois at Urbana-Champaign, 2001

Urbana, Illinois

© Copyright by Scott Pakin, 2001

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN  
GRADUATE COLLEGE

OCTOBER 2001

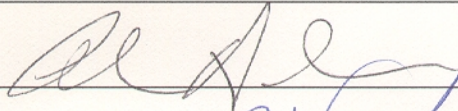
date

WE HEREBY RECOMMEND THAT THE THESIS BY  
SCOTT DOV PAKIN

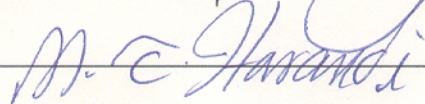
UNRESPONSIVENESS-TOLERANT COLLECTIVE COMMUNICATION  
ENTITLED \_\_\_\_\_

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR  
DOCTOR OF PHILOSOPHY

THE DEGREE OF \_\_\_\_\_

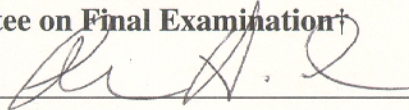


Director of Thesis Research

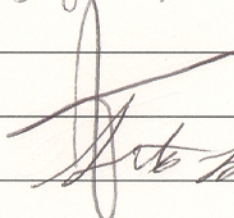
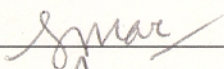


Head of Department

Committee on Final Examination†



Chairperson



†Required for doctor's degree but not for master's.

# Abstract

Collective communication is an important mechanism for parallel programs to efficiently synchronize, distribute data, reorganize the layout of distributed data structures, and maintain a coherent view of global state. Collective-communication operations have traditionally been implemented in the context of parallel computers, which makes assumptions that do not hold for high-performance PC clusters. Due to the structure of user-level messaging layers and the characteristics of COTS hardware and system software, processes are frequently unable to service the network in a timely manner. As a result, collective-communication operations perform only as well as their slowest (i.e., least responsive) participant. As cluster sizes scale upwards, the problem is exacerbated, with there being an increasing likelihood that *some* process is unresponsive to the needs of the group.

For my thesis, I have created and implemented new *unresponsiveness-tolerant* collective-communication mechanisms. The idea is to enable responsive processes to work around unresponsive ones. That is, unresponsive processes should not cause a collective-communication operation to block unless it is absolutely impossible for the program to make progress without a contribution from the unresponsive processes. In addition, those unresponsive processes will complete the collective operation immediately upon again becoming responsive.

I have evaluated my new mechanisms on a PC cluster interconnected with a VIA-based network. The specific contributions of my thesis are a quantification of the levels of unresponsiveness in PC clusters and a classification of their source, a series of techniques designed to tolerate unresponsiveness that stems from each of these sources, and an evaluation of how well each technique performs towards the final goal of increasing the performance of collective-communication-centric parallel applications.

Being a graduate student is like becoming all of the Seven Dwarves. In the beginning you're Dopey and Bashful. In the middle, you are usually sick (Sneezy), tired (Sleepy), and irritable (Grumpy). But at the end, they call you Doc, and then you're Happy.

*Ronald T. Azuma*

SO LONG, AND THANKS FOR THE PH.D., 2000

# Acknowledgments

I would like to thank everyone who supported me intellectually, socially, emotionally, academically, and/or financially during my many years of graduate school at the University of Illinois at Urbana-Champaign. The list of people who deserve my gratitude is Brobdingnagian, and I am sure to offend a large number of people by not naming them here. Nevertheless, their contributions to my education are greatly appreciated.

Let me start by acknowledging a number of my groupmates, in roughly chronological order of their joining the Concurrent Systems Architecture Group. Vijay Karamcheti and John Plevyak were clearly the head and heart of CSAG, and I truly looked up to them. While I learned a great deal from Vijay and John, I regret not having taken even more advantage of their knowledge, experience, and wisdom when I had the opportunity. Julian Dolby and Xingbin Zhang were terrific officemates, and we had a number of fascinating discussions over the years. Julian, in particular, gave me much to think about in terms of compilers, programming languages, and life in general. Mario Lauria was a wonderful sounding board for research ideas. He always asked the right questions and voiced the right concerns; I greatly enjoyed working with him on the Fast Messages project. Finally, I had a lot of good discussions with Geetanjali Sampemane, both in the office and over myriad lunches—not to mention that her Linux expertise certainly came in handy on many occasions.

I would be remiss if I neglected to acknowledge Patrick Sobalvarro, with whom I had the pleasure of working with on implementing dynamic coscheduling. Patrick's honesty and integrity made quite an impression on me. I'm glad my advisor invited Patrick over from MIT to work with me on some joint research.

Speaking of my advisor, Andrew Chien led a fine research group. From all of the paper-readings, discussions, and presentations he had us do, I truly felt like I was getting an education, not merely a degree. It was clear from the seminar classes I took that Andrew's students are better read than the average UIUC graduate student, are better able to raise intelligent points during lectures and discussions, and have a better and deeper grasp of research material. I appreciate all of the intellectual discussions we had together. Andrew can read a paper or hear an idea and very quickly analyze it and provide insight that anyone else would miss. This ability of his has definitely helped cultivate my critical eye towards research. I further

feel that my writing and presentation skills have improved greatly under Andrew's supervision.

Lastly, I would like to thank my family for their support, understanding, and patience. I know I put a lot of stress on my wife, Anya, with all of the times I came home from the office late and exhausted, yet she never ceased to give me love and encouragement. I love you all very much.

#### DEDICATION

To myself, without whose inspired and tireless efforts this book would not have been possible.

*Al Jaffee*

# Table of Contents

|   |            |
|---|------------|
| <b>Abstract</b> . . . . .                               | <b>iii</b> |
| <b>Acknowledgments</b> . . . . .                        | <b>iv</b>  |
| <b>1 Introduction</b> . . . . .                         | <b>1</b>   |
| <b>2 Background</b> . . . . .                           | <b>8</b>   |
| 2.1 Collective communication . . . . .                  | 8          |
| 2.2 Cluster technology . . . . .                        | 11         |
| 2.2.1 Software . . . . .                                | 12         |
| 2.2.2 Network (VIA) . . . . .                           | 13         |
| <b>3 Problem Statement</b> . . . . .                    | <b>15</b>  |
| 3.1 Context . . . . .                                   | 15         |
| 3.2 Problem . . . . .                                   | 17         |
| 3.3 Solution space . . . . .                            | 19         |
| 3.4 Thesis statement . . . . .                          | 24         |
| 3.5 Success criteria . . . . .                          | 25         |
| <b>4 Nonblocking Barriers</b> . . . . .                 | <b>26</b>  |
| 4.1 Introduction . . . . .                              | 26         |
| 4.2 Algorithm . . . . .                                 | 31         |
| 4.3 Example . . . . .                                   | 35         |
| 4.4 Ordering semantics . . . . .                        | 36         |
| 4.4.1 Definitions . . . . .                             | 36         |
| 4.4.2 Reordering rules . . . . .                        | 38         |
| 4.4.3 Implications . . . . .                            | 41         |
| 4.4.4 Restrictions . . . . .                            | 44         |
| 4.5 Alternative implementations . . . . .               | 46         |
| 4.5.1 Single logical clock . . . . .                    | 46         |
| 4.5.2 All-hardware implementation . . . . .             | 47         |
| 4.6 Alternative techniques . . . . .                    | 52         |
| 4.6.1 IntrabARRIER unresponsiveness tolerance . . . . . | 53         |
| 4.6.2 Explicit unresponsiveness detection . . . . .     | 55         |
| 4.6.3 Operating system support . . . . .                | 56         |
| 4.7 Discussion . . . . .                                | 57         |
| <b>5 Experiments</b> . . . . .                          | <b>60</b>  |
| 5.1 Experimental setup . . . . .                        | 60         |
| 5.1.1 Applications . . . . .                            | 61         |

|          |   |            |
|----------|---|------------|
| 5.1.2    | Workloads . . . . .   | 67         |
| 5.2      | Preliminary experiments . . . . .   | 70         |
| 5.2.1    | Total unresponsiveness . . . . .  | 71         |
| 5.2.2    | Characterizing unresponsiveness . . . . .                                 | 77         |
| 5.2.3    | Summary . . . . .   | 83         |
| 5.3      | Nonblocking barrier performance . . . . .                                 | 85         |
| 5.3.1    | Bookkeeping overhead . . . . .  | 85         |
| 5.3.2    | Performance gain from nonblocking barriers . . . . .                      | 87         |
| 5.3.3    | Sources of performance gain . . . . .                                     | 93         |
| 5.4      | Performance robustness . . . . .  | 96         |
| 5.4.1    | Compatibility with other unresponsiveness-tolerating techniques . . . . . | 98         |
| 5.4.2    | Robustness to cluster scale . . . . .                                     | 103        |
| 5.5      | Comparative performance . . . . .   | 105        |
| 5.6      | Discussion . . . . .  | 107        |
| <b>6</b> | <b>Related Work . . . . .</b>   | <b>110</b> |
| 6.1      | Collective-communication libraries . . . . .                              | 110        |
| 6.2      | Collective-communication algorithms . . . . .                             | 111        |
| 6.3      | Collective communication in clusters . . . . .                            | 111        |
| 6.4      | Wide-area collective communication . . . . .                              | 113        |
| 6.5      | Application/runtime-system techniques . . . . .                           | 114        |
| 6.6      | Evaluating unresponsiveness . . . . .                                     | 116        |
| <b>7</b> | <b>Conclusions . . . . .</b>  | <b>117</b> |
| 7.1      | Summary . . . . .   | 117        |
| 7.2      | Experience gained . . . . .   | 119        |
| 7.3      | Future work . . . . .   | 120        |
| 7.4      | Perspective . . . . .   | 122        |
| 7.4.1    | Problem importance . . . . .  | 122        |
| 7.4.2    | Advantages of nonblocking barriers . . . . .                              | 123        |
| 7.4.3    | Applicability . . . . .   | 124        |
| 7.4.4    | Notification mechanism . . . . .  | 124        |
| 7.5      | Contributions . . . . .   | 125        |
|          | <b>References . . . . .</b>   | <b>128</b> |
|          | <b>Colophon . . . . .</b>   | <b>141</b> |
|          | <b>Vita . . . . .</b>   | <b>142</b> |



# List of Tables

|     |   |     |
|-----|---|-----|
| 2.1 | Assumptions of the parallel and distributed computing models . . . .                | 8   |
| 4.1 | Analogous operations . . . . .  | 38  |
| 4.2 | Ordering semantics for nonblocking barriers . . . . .                               | 39  |
| 4.3 | Hardware/firmware/software implementation tradeoffs . . . . .                       | 58  |
| 5.1 | Platform characteristics . . . . .  | 60  |
| 5.2 | Benchmarks used in Chapter 5 . . . . .  | 61  |
| 5.3 | Selected <i>cholesky</i> performance numbers . . . . .                              | 63  |
| 5.4 | Variables used in Algorithms 5.2–5.4 . . . . .                                      | 63  |
| 5.5 | Independent variables . . . . .   | 70  |
| 5.6 | Experimental setup for <i>mg</i> . . . . .  | 76  |
| 5.7 | Giganet cLAN1000 NIC parameters . . . . .   | 87  |
| 5.8 | Sets of unresponsiveness-tolerating techniques used in radix-sort figures . . . . . | 99  |
| 5.9 | Factor of responsive <i>radix sort</i> time (16 processes) . . . . .                | 108 |

# List of Figures

|      |   |    |
|------|---|----|
| 1.1  | Sample 32-node multicast tree in which all nodes are responsive to the network . . . . .                            | 4  |
| 1.2  | Sample 32-node multicast tree in which node 3 is temporarily unresponsive . . . . .                                 | 5  |
| 2.1  | Sample parallel program that uses collective communication . . . . .  | 10 |
| 2.2  | PC cluster architecture . . . . .   | 12 |
| 3.1  | Unresponsiveness observed in a PC cluster . . . . .   | 18 |
| 3.2  | Correlation between the number of barriers in each time range and the number of context switches observed . . . . . | 20 |
| 3.3  | Components of an endpoint . . . . .   | 21 |
| 4.1  | Basic, logarithmic-time, 8-process barrier . . . . .  | 27 |
| 4.2  | Basic, logarithmic-time, 8-process barrier, in which process 7 is temporarily unresponsive . . . . .                | 28 |
| 4.3  | Packet formats used for nonblocking barriers . . . . .  | 35 |
| 4.4  | Bookkeeping for a sample sequence of operations . . . . .   | 37 |
| 4.5  | Processor consistency in the context of nonblocking barriers . . . . .  | 40 |
| 4.6  | Dependencies between barrier and non-barrier operations . . . . .   | 42 |
| 4.7  | Timelines for traditional and optimistic collective communication . . . . .   | 43 |
| 4.8  | Prohibited group-communication pattern . . . . .  | 44 |
| 4.9  | Bookkeeping with a single logical clock . . . . .   | 47 |
| 4.10 | Hardware for an unresponsiveness-tolerant barrier (send side) . . . . .   | 49 |
| 4.11 | Hardware for an unresponsiveness-tolerant barrier (receive side, final stage) . . . . .                             | 50 |
| 4.12 | Hardware for an unresponsiveness-tolerant barrier (receive side, intermediate stages) . . . . .                     | 51 |
| 4.13 | Sample prototype of a VIA barrier function . . . . .  | 53 |
| 4.14 | A nonblocking barrier . . . . .   | 54 |
| 4.15 | A three-way handshake . . . . .   | 56 |
| 5.1  | Communication structure of the Cholesky code . . . . .  | 62 |
| 5.2  | Data dependencies in <i>radix sort</i> . . . . .  | 69 |
| 5.3  | Binary versus flat trees . . . . .  | 70 |
| 5.4  | Time spent in barriers as a function of “computation” time . . . . .  | 72 |
| 5.5  | Measured vs. predicted tally of slow barriers . . . . .   | 73 |
| 5.6  | Barrier program efficiency . . . . .  | 75 |
| 5.7  | Computation time for MG class A, 8 nodes . . . . .  | 77 |
| 5.8  | Correlation between the number of barriers in each time range and the number of context switches observed . . . . . | 78 |

|      |  |     |
|------|--|-----|
| 5.9  | <i>radix sort</i> performance lost to unresponsiveness . . . . .   | 79  |
| 5.10 | Speedup of the <i>cholesky</i> code . . . . .  | 80  |
| 5.11 | Impact of internal contention (naive reductions) . . . . .   | 81  |
| 5.12 | <i>cholesky</i> performance relative to ideal . . . . .  | 83  |
| 5.13 | <i>cholesky</i> performance in light of a CPU-intensive competitor . . . . .   | 84  |
| 5.14 | Performance of the “naive” <i>prefix scan</i> . . . . .  | 86  |
| 5.15 | Communication pattern for a prefix-scan operation . . . . .  | 88  |
| 5.16 | Performance of the cluster-optimized <i>prefix scan</i> . . . . .  | 89  |
| 5.17 | Comparison of measured vs. analytic <i>prefix scan</i> time (polling notification) . . . . .                                   | 91  |
| 5.18 | Performance of the cluster-optimized <i>prefix scan</i> with one competitor per node . . . . .                                 | 92  |
| 5.19 | Performance gain from tolerating unresponsiveness in the cluster-optimized <i>prefix scan</i> . . . . .                        | 93  |
| 5.20 | Performance gain from tolerating unresponsiveness in <i>radix sort</i> . . . . .   | 94  |
| 5.21 | Cumulative time blocked on receives in <i>radix sort</i> . . . . .   | 95  |
| 5.22 | Time blocked on receives in <i>radix sort</i> , expressed as the difference in the percentage tally of receive times . . . . . | 97  |
| 5.23 | Radix sort performance (polling notification) . . . . .  | 100 |
| 5.24 | Radix sort performance (blocking notification) . . . . .   | 102 |
| 5.25 | Performance improvement in <i>radix sort</i> as a function of the number of processes . . . . .                                | 104 |
| 5.26 | Comparison of nonblocking barriers to implicit coscheduling . . . . .  | 106 |
| 7.1  | Best reported performance on the SPEC CFP95 benchmark over time . . . . .  | 119 |

# List of Algorithms and Procedures

|     |  |    |
|-----|--|----|
| 4.1 | Unresponsiveness-tolerant barrier . . . . .                          | 33 |
| 4.2 | Nonblocking receive that supports unresponsiveness-tolerant barriers | 34 |
| 5.1 | Barrier microbenchmark . . . . .                                     | 61 |
| 5.2 | Prefix-scan (data parallel) . . . . .                                | 64 |
| 5.3 | Prefix-scan (naive) . . . . .  | 65 |
| 5.4 | Prefix-scan (optimized for clusters) . . . . .                       | 66 |
| 5.5 | Radix sort (data parallel) . . . . .                                 | 66 |
| 5.6 | Radix sort (message-passing) . . . . .                               | 68 |

# 1 Introduction

Clusters of personal computers are rapidly becoming a dominant platform for high-performance computing. Even former supercomputer-only shops such as NCSA are migrating users from parallel computers to PC clusters [78]. The reason for doing this is not only that PCs have an excellent price:performance ratio, but also that the latest generation of PC microprocessors and local-area networks is *performance-competitive* with supercomputers.<sup>1</sup>

However, PC clusters are a qualitatively different—and more complex—platform from parallel supercomputers. While parallel supercomputers generally either sport a small run-time system on each node or share a single heavyweight operating-system image across all the processors, PC clusters run an independent, heavyweight, commodity operating system on each node. Operating system independence implies that the operating systems are not designed for global, coordinated resource management and scheduling. And their heavyweight, commodity nature is an issue because it impacts predictability. User processes share the machine with each other and with operating system daemons, which run at unpredictable times for unpredictable durations. As the size of the cluster increases, it rapidly approaches certainty that *some* process in a program is descheduled at any given time. Memory hierarchies that extend out to disk cause memory access times to vary from a fraction of a microsecond to tens of milliseconds. In short, commodity operating systems make it virtually impossible to reason about response times. Another difference between parallel supercomputers and PC clusters is that the former are entirely homogeneous, while PC clusters frequently have different clock speeds, system architectures, and amounts of memory on each node.<sup>2</sup> This heterogeneity further diminishes predictability, because processes may observe different performance depending upon exactly which node they run on.

PC clusters are also qualitatively different from traditional distributed systems. Distributed systems are designed around wide-area communication—in which the

---

<sup>1</sup>SPEC CPU performance, from <http://www.spec.org/osg/cpu95/results> (2Q1999):

|   |                         |
|---|-------------------------|
| SGI Origin 2100 supercomputer (250 MHz MIPS R10000):    | 15.3 CINT95, 25.2 CFP95 |
| PC based on Intel MS440GX chipset (555 MHz Intel Xeon): | 23.6 CINT95, 16.9 CFP95 |

<sup>2</sup>Because of the rate of growth of PC performance, it is virtually impossible to obtain “old” PCs when incrementally expanding a cluster. Additional supercomputer nodes are available only from a single vendor, which is slower to introduce node upgrades.

network, not the endpoints, is the bottleneck—and, hence, give higher precedence to fault tolerance and security than to performance. Modern PC clusters, however, are interconnected with networks fast enough to shift the performance bottleneck from the network fabric to software running at the endpoints. These networks are fairly reliable—Myrinet, for instance, has a bit error rate of 1 bit error per  $1 \times 10^{15}$  bits [16]. In addition, security is less of an issue because a PC cluster resides in a single administrative domain. Hence, there is little concern that a malicious administrator will configure one node to sabotage the other nodes. The key difference from a programming perspective is that PC cluster software can do without costly checks for fault detection, dropped or corrupted messages, and protocol violations. Furthermore, modern PC clusters do not have to sacrifice CPU time to reduce network involvement, e.g., by aggregating messages.

The distinguishing characteristics of PC clusters are that they:

- are composed primarily of commodity hardware,
- may include a small number of non-commodity parts integrated with the rest of the system across standard interfaces (e.g., a special network card on a PCI bus),
- run unmodified, commodity operating systems, and
- execute interactive applications, not just batch-submitted.

The primary use of PC clusters is for large, resource-demanding parallel applications. To simplify coding and improve performance, a number of parallel applications utilize *collective-communication operations*. In contrast to point-to-point messages, which have a single sender and a single receiver, collective communication operations enable sending one-to-many, many-to-one, and many-to-many messages. By using operations in these basic categories, processes can replicate state and ensure that updates to one copy are correctly perceived by all the other extant copies. The following are examples of collective operations:

- multicast/broadcast — send a message from one process to many
- barrier — synchronize processes
- gather — concatenate data from each process into an aggregate on one process
- scatter — divide one process's message into many pieces and send each to a different process

- reduce — perform an associative operation across the elements of a distributed array and provide the result to a single process
- shift/circular shift — move data from each process to its neighbor, wrapping around in the case of circular shift

In addition, there are various combinations of the above, such as gather-to-all (a gather followed by a broadcast) and reduce-to-all (a reduction followed by a broadcast).

The key problem that occurs when performing collective operations in a PC cluster is that not all of the participants are actively servicing the network in unison. Because a collective operation cannot complete until all participants have contributed to it, the latency of the entire operation is determined by the slowest member. The highly-variable response times in PC clusters therefore bring the average collective-operation performance close to its worst-case value.

Collective operations utilize *group semantics* in many parallel-communication libraries [9, 41, 50, 73, 80]. That is, a group of processes collectively decides to perform a particular operation. These libraries assume that process groups are semi-permanent; processes do not spontaneously join or leave a group without consensus from the rest of the group. Processes that crash, are killed by an external entity, or that lose network connectivity generally terminate the entire application.<sup>3</sup> Group semantics are a reasonable practice for parallel applications because a single set of programmers produces the entire application and has *a priori* knowledge of the application's communication pattern. Distributed applications, however, are quite different from parallel applications. In a distributed application, there is comparatively little global coordination of activity. Processes contain different code, written by different (possibly mutually-hostile) sets of programmers. Furthermore, distributed applications take a more pessimistic view of the world than parallel applications. They assume that processes die sporadically, networks are severed and reconnected at random, and messages are frequently lost or corrupted. Hence, collective operations must be implemented with specially designed—and generally costly and non-scalable—algorithms such as CBCAST or ABCAST [12] to ensure global state consistency, even in the face of disaster.

As in a parallel computer, these collective operations are highly sensitive to unresponsiveness. However, the problem is exacerbated in a distributed system because a high degree of unresponsiveness is indistinguishable from failure. Failure-

---

<sup>3</sup>PVM [41] is a notable exception.

recovery code can require a significant amount of communication and time to complete. For example, Vogels, et al. discovered that under heavy load (read as: node unresponsiveness), the Microsoft Cluster Service could take as long as a *minute* to broadcast a single message to 32 nodes [103]. (In contrast, a 10-node broadcast typically took 2–5 seconds.)

The key untapped problem for both parallel and distributed systems is *endpoint responsiveness*. If a process does not participate in a collective operation promptly, it will delay the entire operation and, therefore, degrade overall application performance. The context of my thesis research is high-performance, latency-sensitive parallel applications that:

- rely on user-level collective communication and run on a large PC cluster,
- are interconnected with a high-speed network, and
- contain an insufficient number of threads to hide much communication latency.

As a motivating example of how unresponsiveness is manifested, consider a multicast operation. To multicast a message to a number of receivers, the root of the multicast tree must send the message to each of its children; those children, in turn, forward the message to each of their children; and so on, until all nodes have received the message. Once a node receives the message and forwards it onward, it is no longer needed for the multicast operation and can resume computing. This is illustrated in Figure 1.1.

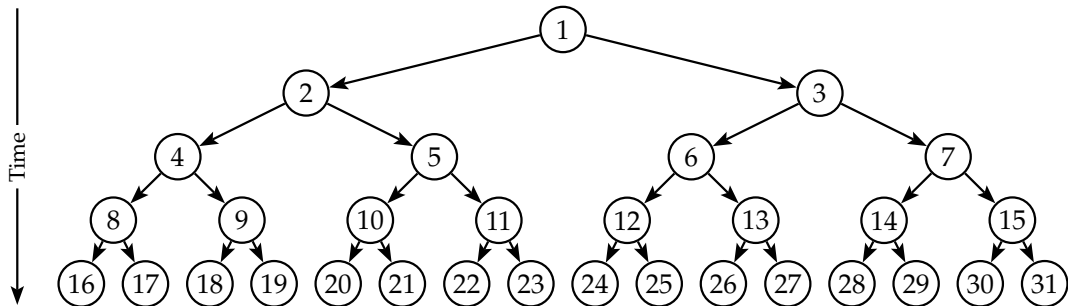


Figure 1.1: Sample 32-node multicast tree in which all nodes are responsive to the network

If a node high up in the multicast tree is busy computing and is not servicing the network, it will delay all the nodes downstream of it in the multicast tree. Figure 1.2 depicts the same tree as in Figure 1.1, except that node 3 is late to receive and forward the multicast message. As a result, nearly half the nodes in the network



are idled, waiting to receive the multicast. One would expect similar behavior regardless of whether the multicast is implemented in hardware or software. This is because hardware can buffer only a finite amount of data before requiring instruction from the host program.

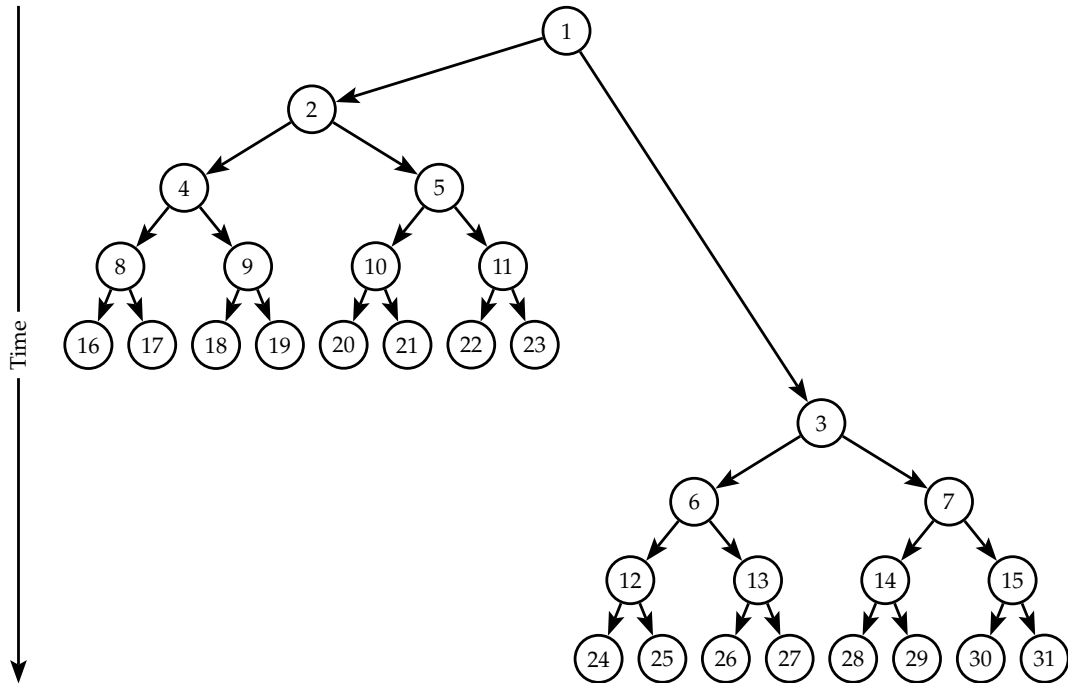


Figure 1.2: Sample 32-node multicast tree in which node 3 is temporarily unresponsive

There are three sources of node unresponsiveness in a PC cluster that can lead to the poor behavior illustrated in Figure 1.2:

1. Process descheduling — delays of tens to hundreds of milliseconds
2. OS interrupt handling — delays of hundreds of microseconds to a few milliseconds
3. Application load imbalance — application-dependent delays

Note that process descheduling can occur even if there are no other obvious user processes running on the node; daemons and services running on behalf of the operating system awake at unpredictable intervals and perform activities that need the CPU, memory, I/O, and other resources, taking those resources away from the parallel application. OS interrupt handling includes page fault handling, device driver scheduling/running time, complex kernel thread interactions caused by contention

for shared resources (such as locks and semaphores). Application load imbalance is sometimes a result of an application neglecting to service the network in a timely manner, possibly to avoid polluting the memory hierarchy.

While application load imbalance decreases performance on any system, the other forms of unresponsiveness are problematic primarily on high-performance PC clusters. Distributed system performance is frequently limited by the (slow) network or disk, so additional multi-millisecond latencies do not noticeably decrease overall performance. Parallel computers generally devote special-purpose hardware and/or custom or modified operating systems to reduce unresponsiveness. For instance, coordinated scheduling [82] eliminates process misscheduling, but is inappropriate for a commodity PC cluster environment because it generally requires modifications to the operating system scheduler or special synchronization hardware. Similarly, parallel computers frequently delegate most system services to service or I/O nodes and run only a vestigial run-time system on the compute nodes, while PC clusters run the latest version of some unmodified, commodity OS on each node.

For my thesis, I tackled the problem of performance loss due to node unresponsiveness in PC clusters. This is an important problem because:

1. PC clusters are rapidly becoming a popular high-performance platform,
2. performance degradation will increase with larger clusters and faster processors/networks, and
3. the problem has not previously been solved.

In point 3, I do not consider a solution to be one that merely pretends that PC clusters are parallel computers and loads them with custom operating systems and/or specialized hardware. Doing so strips PC clusters of their defining characteristics—low cost, ease of upgrading, simplicity of integrating or adapting to new technologies, availability of software, etc.—and is therefore an unsatisfying solution.

Coordinated thread scheduling (such as gang scheduling [82], dynamic coscheduling [95], and implicit coscheduling [5]) is not sufficient to eliminate the problem of unresponsiveness in PC clusters. Even if all the threads in a parallel job are coscheduled, communication operations within those threads may not be. While a single misscheduled thread can induce tens to hundreds of milliseconds of unresponsiveness at once, real-world occurrences such as cache misses, page faults (which may need to be satisfied by disk), intra-application load imbalance, resource contention (e.g., for a network interface), and hardware interrupts each introduce small amounts of unresponsiveness that can add up quickly. Hence, coordinated

thread scheduling needs to be augmented with new mechanisms and algorithms for tolerating individually-small, but collectively-large, sources of unresponsiveness.

The result of my thesis project is a demonstration that it is, in fact, possible to improve the performance of PC clusters by making collective-communication operations tolerant of endpoint unresponsiveness. The specific contributions of my work are the following:

1. a set of communication/network interface architecture features that enable latency-tolerant collective communication,
2. an implementation of various collective-communication operations that is robust to the forms of unresponsiveness listed on page 5,
3. a quantitative evaluation of the performance of a set of applications that rely on collective communication and run atop VIA [24, 35].<sup>4</sup>

The remainder of this dissertation is structured as follows. Chapter 2 presents the requisite background information needed to understand my thesis work and to put it in context. Chapter 3 provides the motivation for my work and states the precise problem I attempted to solve. Chapter 4 lists the constraints guiding my solution and describes the approach I eventually took. Performance measurements of my implementation and an analysis of the results are shown in Chapter 5. Chapter 6 covers projects and research results related to those presented in this dissertation. Finally, Chapter 7 draws some conclusions from my thesis work.

Perhaps an editor might begin a reformation in some such way as this. Divide his paper into four chapters, heading the 1st, *Truths*. 2nd, *Probabilities*. 3rd, *Possibilities*. 4th, *Lies*. The first chapter would be very short, as it would contain little more than authentic papers and information from such sources as the editor would be willing to risk his own reputation for their truth. The second would contain what, from a mature consideration of all circumstances, his judgment should conclude to be probably true. This, however, should rather contain too little than too much. The third and fourth should be professedly for those readers who would rather have lies for their money than the blank paper they would occupy.

*Thomas Jefferson*

LETTER TO JOHN NORVELL, 1807

---

<sup>4</sup>Specifically, I used Gigaset's VIA implementation [1].

# 2 Background

In order to understand the research presented in this dissertation, one must first become acquainted with some of the concepts behind both collective communication and PC clusters. Although collective communication was introduced in Chapter 1, Section 2.1 illustrates the differences between the parallel-computing view of collective communication and the distributed-computing view, to help clarify some of the decisions made in this thesis. Section 2.2 describes the hardware and software components that comprise a PC cluster, paying special attention to the communication subsystem, as this is central to unresponsiveness tolerance.

## 2.1 Collective communication

Collective communication has been studied extensively and in a variety of contexts [8, 20, 31, 49, 60, 73, e.g.]. Collective communication research is generally performed using one of two models: parallel computing or distributed computing. Table 2.1 contrasts the key assumptions made by programmers writing to each of those models. In summary, the parallel computing model assumes a more controlled environment and therefore takes a more “optimistic” view of the system. The distributed computing model utilizes collective communication primarily for fault tolerance.

Table 2.1: Assumptions of the parallel and distributed computing models

|                                       | Parallel computing assumption | Distributed computing assumption |
|---------------------------------------|-------------------------------|----------------------------------|
| Network speed                         | Fast                          | Slow                             |
| Host speed                            | Fast                          | Fast                             |
| Network reliability                   | High                          | Low                              |
| Host reliability                      | High                          | Low                              |
| <i>A priori</i> process coordination  | Yes                           | No                               |
| Network, host, or process homogeneity | Yes                           | No                               |

To demonstrate how a parallel program might use collective communication, Figure 2.1 lists a simple parallel program<sup>1</sup> written in Fortran [52] with the MPI messaging library [73]. The program, `pi`, computes  $\pi$  and utilizes a pair of collective communication operations in the process. The way the program works is as follows. First, the user starts up some number of copies of the executable. (This is external to the text of the program.) MPI assigns each process a rank in the computation, which stays fixed for the duration of the run; it is assumed that processes neither join nor leave the computation after the call to `MPI_INIT`. Second, process 0 reads a number of intervals, `n`, from the user and broadcasts that number to all `numprocs` processes with `MPI_BCAST`. `MPI_BCAST` is a collective operation. All processes make the call in unison, all processes agree that process 0 is producing the data that the remaining processes will consume, and all processes block until their participation in the broadcast is no longer needed. Third, each process computes  $\int \frac{4}{1+x^2}$  over a unique, nonoverlapping subset of  $0, \dots, 1$ . Fourth, process 0 sums the results of all processes using a (collective) reduction operation, `MPI_REDUCE`. Similar to `MPI_BCAST`, `MPI_REDUCE` requires all processes to participate, agree on a destination, and block until completion. Finally, process 0 outputs the sum as an approximation of  $\pi$ .

To demonstrate how, in contrast to the previous example, a *distributed* application might use collective communication, consider a distributed computational steering environment. The idea is that a large application runs on a collection of machines, and one or more users control the run-time behavior of that application in real time using a graphical or virtual-reality interface [90]. For example, consider a computationally-steered cardiac defibrillator electrode design program [86]. The goal is to find the best place to implant electrodes within a human body to defibrillate a patient suffering from, for example, cardiac arrhythmias. This is accomplished by simulating a human thorax and letting a user interactively place defibrillation devices at various places on the virtual thorax and letting a distributed set of machines simulate the result.<sup>2</sup>

Such a system might use collective communication in a variety of ways, mostly different from that in the `pi` program listed in Figure 2.1. First, it must deal with users who act spontaneously and therefore cause control messages to be sent at unpredictable times. Contrast this with the `pi` example, in which all processes simultaneously call `MPI_BCAST/MPI_REDUCE`. Second, the computational steering system must broadcast results to a number of users that varies over time. In contrast, MPI uses a static process model; no processes can join or leave a computation during

<sup>1</sup>The example in Figure 2.1 was taken almost verbatim from the book *Using MPI* [44].

<sup>2</sup>SCIRun, on which this cardiac defibrillator code runs [86] allows only a single user at a time. For the ensuing discussion in this dissertation, we assume that multiple people could use such a system simultaneously.

```

c*****
c pi.f - compute pi by integrating f(x) = 4/(1 + x**2)
c
c Each node:
c 1) receives the number of rectangles used in the
c approximation.
c 2) calculates the areas of it's rectangles.
c 3) Synchronizes for a global summation.
c Node 0 prints the result.
c
c Variables:
c
c pi the calculated result
c n number of points of integration.
c x midpoint of each rectangle's interval
c f function to integrate
c sum,pi area of rectangles
c tmp temporary scratch space for global summation
c i do loop index
c*****
program main

include 'mpif.h'

double precision PI25DT
parameter (PI25DT = 3.141592653589793238462643d0)

double precision mypi, pi, h, sum, x, f, a
integer n, myid, numprocs, i, rc
c function to integrate
f(a) = 4.d0 / (1.d0 + a*a)

call MPI_INIT( ierr )
call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
print *, "Process ", myid, " of ", numprocs, " is alive"

sizetype = 1
sumtype = 2

10 if ( myid .eq. 0 ) then
write(6,98)
98 format('Enter the number of intervals: (0 quits)')
read(5,99) n
99 format(i10)
endif

call MPI_BCAST(n,1,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)

c check for quit signal
if ( n .le. 0 ) goto 30

c calculate the interval size
h = 1.0d0/n

sum = 0.0d0
do 20 i = myid+1, n, numprocs
x = h * (dble(i) - 0.5d0)
sum = sum + f(x)
20 continue
mypi = h * sum

c collect all the partial sums
call MPI_REDUCE(mypi,pi,1,MPI_DOUBLE_PRECISION,MPI_SUM,
$ 0,MPI_COMM_WORLD,ierr)

c node 0 prints the answer.
if (myid .eq. 0) then
write(6, 97) pi, abs(pi - PI25DT)
97 format(' pi is approximately: ', F18.16,
+ ' Error is: ', F18.16)
endif

goto 10

30 call MPI_FINALIZE(rc)
stop
end

```

Figure 2.1: Sample parallel program that uses collective communication

its execution. Third, because nodes, which may be geographically distributed, can hang or become disconnected from the network, the computational steering system must prevent the entire application from aborting when a small piece (say, one of the users' interactive environments) stops. Fourth, and related to the previous point, the computational steering system must handle retries, timeouts, and out-of-order message delivery in a sensible way. Finally, the computational steering system must ensure a global ordering on user actions. Otherwise, two users might simultaneously choose to remove all defibrillators from the thorax and then place their defibrillator somewhere. Without total ordering, different users might see either of the two defibrillators. In the pi program, this is not an issue, because MPI\_BCAST and MPI\_REDUCE are effectively preceded by a barrier operation, which forces a total ordering on the collective-communication operations.

In summary, collective communication research is generally performed with either a parallel-computing mindset or a distributed-computing mindset. The two focus on completely different sets of issues. In parallel computing, the primary issue is performance; programs are expected to exist in a "best case" environment and merely need to run as fast as possible. In distributed computing, the primary issue is

maintaining consistent global state in the face of uncoordinated state modifications as well as processes and networks that sporadically produce erroneous data or fail altogether. Performance is of a secondary concern.

I performed my thesis research in the context of parallel computing and with the parallel-computing goal of maximizing performance. However, some of the mechanisms I developed should be readily applicable to distributed computing, as well. For example, a distributed computing application can maintain consistent global state by periodically introducing global barrier operations to ensure message causality. This is what was done for parallel discrete-event simulation with predictive barrier scheduling [67] and noncommittal barrier synchronization [79].

## 2.2 Cluster technology

A PC cluster (Figure 2.2) is composed of a number of *nodes*—independent computers each containing memory, CPU disk, and a network interface card (NIC)—interconnected by a network. The identifying characteristics of a PC cluster are:

1. Each node's CPU memory, disk, and network are based on COTS (commodity off-the-shelf) technology. A corollary is that the NIC resides on the I/O bus and there is no cache-coherent shared memory across nodes (although there is *within* a multiprocessor node).
2. There may be multiple CPUs and/or multiple disks per node, but only one OS kernel.
3. The network—including the NIC—can send and receive with both low latency (the time required to send a minimally-sized message from one node to another) and high bandwidth (the amount of data transferable per unit time).
4. The network has a **low bit-error rate**.
5. Nodes are mostly homogeneous. That is, the architectures are binary-compatible, but may sport different CPU speeds, numbers of CPUs, amounts of memory, numbers/sizes of disks, bus speeds, and possibly even types of network interfaces.

Note that the above characteristics classify a number of parallel computers as clusters. For example, the IBM SP2 [2] is composed of homogeneous RISC System/6000s, runs a single AIX kernel, and utilizes a custom, high-speed network with a low bit-error rate. It is therefore considered a cluster for the purpose of this dissertation.

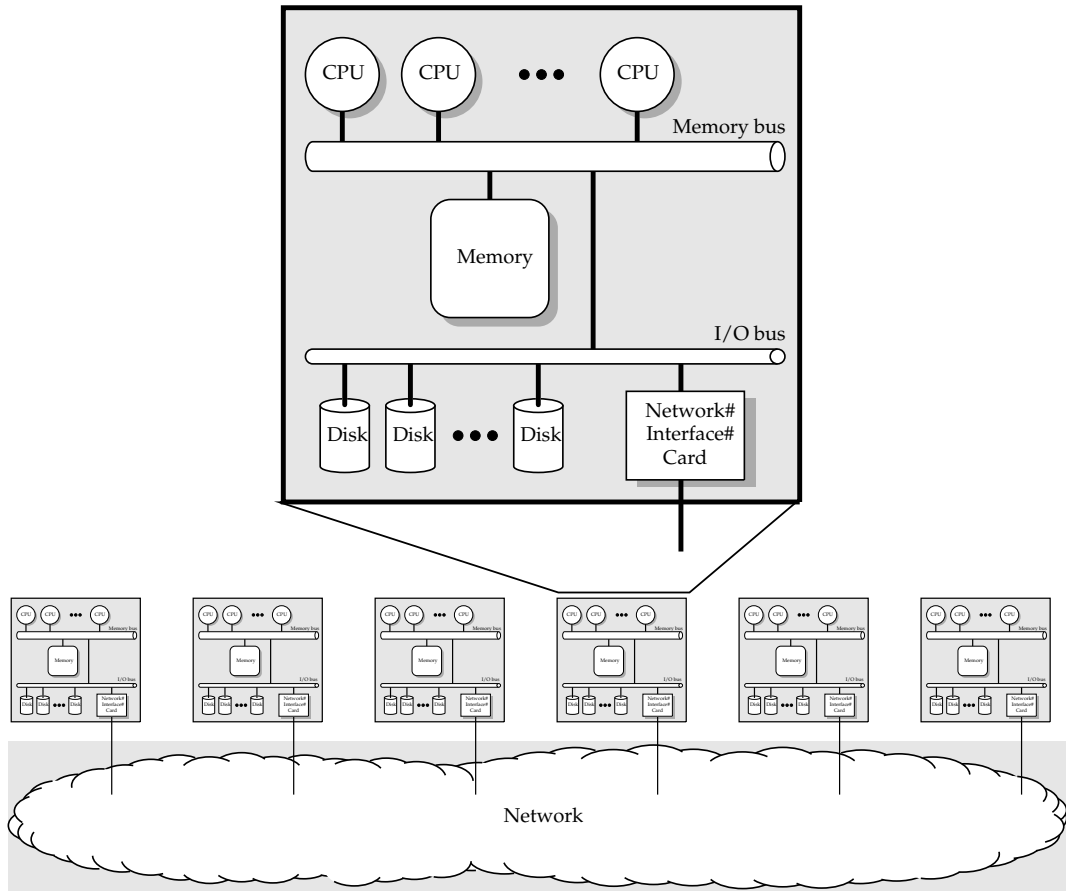


Figure 2.2: PC cluster architecture

### 2.2.1 Software

The identifying characteristics of a PC cluster's software environment are:

1. The operating system is a commodity workstation operating system, such as Windows NT [29] or Linux [91], as opposed to a small, custom run-time system or simple loader.
2. If the system software is modified, the modifications do not break existing applications (i.e., the interface remains the same).
3. Each node in the cluster runs a separate copy of the operating system.

The distinction in point 1 is that COTS OSES support features such as multiprocessing, virtual memory, demand paging, and multiple, concurrent users, while the special-purpose system software is designed to "stay out of the way" of applications, giving them direct access to most of the node's resources while providing only a minimal number of system services.



### 2.2.2 Network (VIA)

As a sweeping generalization, the performance bottlenecks in a cluster are the software and the I/O bus; the network and CPUs are generally “fast enough”. Industry has been gradually improving I/O speeds over time, introducing faster standard buses, such as PCI [94]. Academia has been studying ways to improve the operating system and communication run-time software. While there has been much work in improving the operating system’s protocol-handling code [17, 33, 87], the operating system as a whole remains a significant source of performance loss [23]. Hence, the real key to high-performance cluster communication is to remove the operating system from the critical path of communication. To this end, a number of researchers have been investigating user-level communication and custom, high-speed protocols [34, 71, 85, 89, 99, 104]. By bypassing the operating system, user-level messaging layers can improve bandwidth, latency, and overhead by up to several orders of magnitude. Four of these messaging layers—Illinois Fast Messages (FM) [85], Active Messages (AM) [71], U-Net [104], and VMMC-2 [34]—led to the development of Compaq, Intel, and Microsoft’s new network interface standard, the Virtual Interface Architecture (VIA) [24, 35]. VIA is an interesting architecture because of its industry support<sup>3</sup> and because it:

- contains a notion of reliability; endpoints can be notified if a packet is lost, dropped, or corrupted,
- was designed for high-speed,<sup>4</sup> user-level communication, and
- supports one-sided communication (PUT and GET), in addition to the more traditional two-sided communication (SEND and RECEIVE).

However, VIA has no built-in support for collective communication.

The VIA model is connection-oriented, with connections going between communication endpoints called *virtual interfaces* (VIs). Each process allocates one or more VIs from its local NIC and connects each of them to exactly one remote VI. There is a distributed-systems mentality to these connections: the “server” VI is allowed to accept or deny a connection request from the “client” VI. To send a message, a process registers (i.e., pins) a region of memory within its address space. It then enqueues a send descriptor onto a queue that is accessible by the NIC and notifies the NIC that it has data waiting to be sent. Similarly, the receiving process registers a region of

---

<sup>3</sup>Approximately 85 companies are named as contributors on [http://www.viarch.org/html/Contributors/vi\\_contributors.htm](http://www.viarch.org/html/Contributors/vi_contributors.htm).

<sup>4</sup>Giganet, Inc. claims their cLAN1000 VIA host adapter achieves latencies of 3.5  $\mu$ sec and bandwidths over 110 MB/s ([http://www.giganet.com/pdf/cLAN\\_HostAdapter.pdf](http://www.giganet.com/pdf/cLAN_HostAdapter.pdf))

memory into which to receive the message, enqueues a receive descriptor, and notifies the NIC of the descriptor's existence. The receiver is notified of message arrival by either polling or blocking, and it is possible to block on multiple connections at once.

One unique feature of VIA is its support of multiple *reliability levels*. When a process establishes a network connection, it specifies as its reliability level one of: unreliable delivery, reliable delivery, or reliable reception. In unreliable delivery, messages may be dropped or misordered with no notification to either VI. In reliable delivery, messages that are lost or corrupted in the network or arrive out of order not only notify the VI of the problem, but also tear down the connection. Reliable reception is just like reliable delivery, but does not transfer a new message until it knows that the previous message was delivered intact. Note that "reliable" in VIA terminology is somewhat of a misnomer; it implies only notification—not retransmission—of lost messages. Furthermore, if a receive descriptor has not been posted by the time a message arrives on a reliable connection, the message is dropped, the connection is torn down, and an error is signalled on the VI. In Chapter 4, we will see how VIA's reliable communication is used as a basis for unresponsiveness-tolerant collective communication.

Always design a thing by  
considering it in its next larger  
context—a chair in a room, a room  
in a house, a house in an  
environment, an environment in a  
city plan.

*Eliel Saarinen*

TIME, JUNE 2, 1977

(quoted by his son, Eero)

# 3 Problem Statement

The focus of my research is to improve the performance of parallel programs running on PC clusters. Section 3.1 characterizes the hardware and software context in which the thesis work was executed. Section 3.2 describes an important new problem that arises in that context. The space in which I attacked that problem is detailed in Section 3.3. This leads to the precise thesis statement in Section 3.4. Finally, Section 3.5 explains how to validate the success of my approach.

## 3.1 Context

The context of my work is performance-demanding, parallel applications. Such applications are traditionally designed for integrated parallel computers, recent examples being machines such as the T3E [93]. These computers tend to run only a simple run-time system on each node instead of a full operating system. Besides making more of the node's resources available to the application, this setup presents a simple resource model to the user:

1. All nodes running the user's job will do so uninterrupted.
2. Nothing external to the user's job will pollute the memory hierarchy.

In short, users can assume that the system is always responsive. This assumption simplifies programming. Applications need not handle the case in which a group of processes needs to synchronize, but is delayed by some processes being descheduled. They need not handle the case in which a group of processes plans to distribute data among the group, but one of the participants' working set is paged out to disk. And they need not handle the case in which periodic operating system activity causes processes to stall at regular intervals, delaying any process that needs to coordinate with a stalled process.

While integrated parallel computers simplify programming and reasoning about program performance, they have a crucial drawback. These systems tend to be expensive—beyond the budgets of many researchers and application scientists. A less-costly alternative that is rapidly gaining in popularity is PC clusters composed of commodity hardware and software. Because of fierce marketplace competition to sell node hardware, prices are low and performance is high. As a result, the

price:performance ratio of the cluster as a whole exceeds that of an integrated parallel computer. For example, the HPVM [21] cluster performs within a factor of two to four of the Origin 2000 [64] and T3E [93] supercomputers, at roughly a sixth of the cost [22].

PC clusters are an important, emerging new platform. Many supercomputer centers, such as the National Center for Supercomputing Applications and the Pittsburgh Supercomputing Center, are building large clusters instead of (or, occasionally, in addition to) buying new parallel computers [26, 43, 92]. Reasons frequently cited, in addition to low price and high performance, are that PC clusters can not only be scaled to arbitrary sizes, but they can also be scaled in arbitrary increments. The tightly integrated CPU, memory, and network used in parallel supercomputers often dictate constraints on scalability (e.g., a maximum of a few thousand processors) or increment (e.g., requiring that the total number of processors be a power of two). PC clusters, in contrast, can scale upward commensurate with what the organization's budget allows.

The following are some of the fundamental characteristics of PC clusters:

- PC clusters are inexpensive relative to more tightly-integrated supercomputers.
- The CPU is a commodity microprocessor and may not be altered.
- The operating system is a commodity and may not be altered.
- The network interface resides on the I/O bus and is neither integrated with nor contains any special channels to the memory system, CPU, etc.
- The microprocessor on the NIC (if any) is less powerful than the host CPU, and the NIC has little (if any) on-board memory.
- Interrupts from a device to the CPU are expensive.

Treating the above characteristics as constraints provides focus to my research and adds a degree of realism. These constraints prevent me from postulating solutions that alter the fundamental characteristics of PC clusters or that require unobtainable resources or attributes thereof. Some of the implications of my constraints, however, are that cluster-wide coordinated scheduling, if used, must be built atop the base operating system scheduler (as in Sobalvarro, et al. [95], for example). The NIC cannot buffer more than a small amount of data relative to the link speed. And NIC-to-host interrupts are not allowed on the critical path of communication.

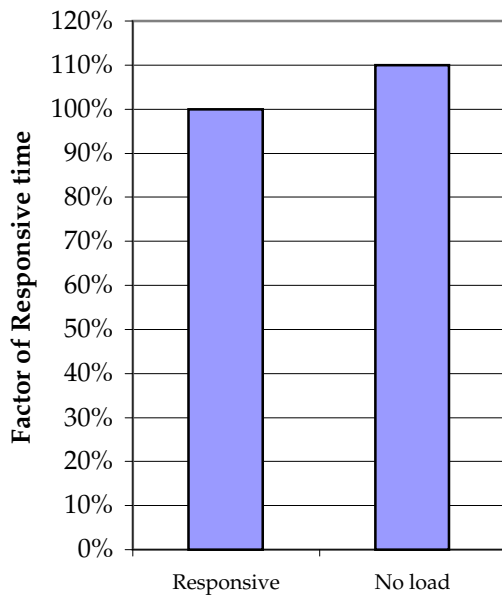
## 3.2 Problem

Although the scalability features and price:performance ratio of PC clusters are appealing, PC clusters present an important new problem: unresponsiveness. While parallel supercomputers tightly integrate the CPU, memory, and NIC, interprocess communication on a PC cluster must interact with various hardware and software layers. These layers vie for the system's resources, causing those resources occasionally to become inaccessible to a running application. The result is that processes become unresponsive to each other.

In the absence of communication, unresponsiveness is a nonissue. However, when a process requires a result produced by an unresponsiveness process in order to make progress, it must wait until that process becomes responsive again. Even worse, when a collective-communication operation is being performed, any number of processes may block waiting for an unresponsive peer. Hence, although PC clusters can exhibit comparable peak performance to integrated parallel supercomputers [22], unresponsiveness can impact the sustained performance.

Figure 3.1 evinces the severity of the problem. The figure shows the performance of a radix sort (which relies heavily on collective communication) running on an 8-node cluster. The first graph in the figure contains two bars. The Responsive bar represents the performance observed on a cluster specially configured to be as responsive as possible. The radix sort program was run at the highest OS priority that would not cause priority inversion problems with the network device driver. In addition, each node was given a spare processor for the sole purpose of absorbing operating system activity. This is not standard operating procedure for a cluster, as it wastes 50% of the processors and adversely affects scheduling fairness on each node. However, the Responsive configuration does show how the cluster would perform if it were used more like a dedicated parallel computer.

The No Load bar in the first graph of Figure 3.1 shows the performance when *radix sort* is run normally, with no additional load on the cluster. As the figure shows, the No Load run is 9.9% slower than the Responsive run. This is an important observation: Even on a cluster that is running a single user job and nothing else, unresponsiveness noticeably impacts performance. However, performance can get much worse. The second graph in Figure 3.1 additionally shows what happens when *radix sort* is run either with a competitor for the CPU (a simple spin-loop) running on each node (One competitor/node) or with two *radix sort* processes running on each node (Two processes/node). The One competitor/node bar is 5.3 times as tall as the Responsive bar, and the Two processes/node bar is 6.1 times as tall as the Responsive bar. Ideally, each of those bars should be only twice as tall as Responsive, as only two processes are sharing each CPU. The reason is that a single descheduled



The above graph depicts a detailed view of these two bars.

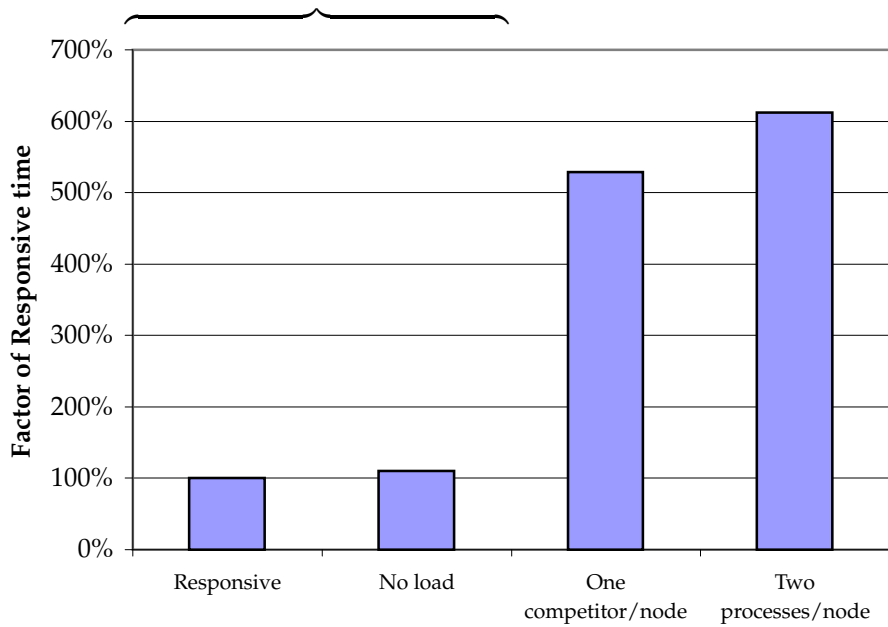


Figure 3.1: Unresponsiveness observed in a PC cluster

process delays even the processes that *are* scheduled, as they must block waiting for the unresponsive process to be rescheduled, so they can communicate with it. The conclusion to draw from Figure 3.1 is that unresponsiveness is a problem even on an unloaded system, and it becomes a serious problem in the presence of load.

### 3.3 Solution space

Unresponsiveness is a multifaceted problem, and there are many ways to approach solving it. This section describes the approach my thesis takes and justifies it as the most propitious tack to take. Phrases that distinguish the specific space in which my thesis work is performed are shown in bold.

At the top level, there are two possible ways to deal with unresponsiveness. Either it can be tolerated, or it can be removed altogether. While removing unresponsiveness may sound more favorable at first, doing so in the context of PC clusters has a number of drawbacks. Primarily, removing unresponsiveness generally requires a dedicated system, specially modified operating system, or wasted resources. (Recall that the measurements for the Responsive bar in Figure 3.1 were taken while running with abnormally high process priorities and on a dedicated system with half of the CPUs essentially wasted.) Those conditions degrade many of the benefits of using COTS components, such as the good price:performance ratio or the ability to take advantage of the timesharing features of a commodity operating system.

A second problem with removing unresponsiveness is that only unresponsiveness external to the user's application can be removed without requiring application modifications. If unresponsiveness is caused, for example, by load imbalance within the application, it can be removed only if the application is rewritten to schedule tasks in a more flexible manner. However, this is not always possible or practical.

Because removing unresponsiveness violates COTS properties and may require application modifications, my thesis will instead **tolerate unresponsiveness**.

There are two places to look for unresponsiveness: in the network or in the nodes. With modern high-speed networks such as Myrinet [16] and Giganet [1], the network is effectively infinitely fast. More precisely, a network link bandwidth of 2+2 gigabits/second (i.e., 2 Gbps in each direction) is roughly twice as fast as a 64-bit/33MHz PCI (I/O) bus. Because in a COTS-based PC cluster, all network traffic must cross the I/O bus, those numbers indicate that the I/O bus will become a performance bottleneck long before the network does. Figure 3.2 provides further evidence that the network is not the primary source of unresponsiveness. Instead, unresponsiveness lies within the endpoints and is highly correlated to

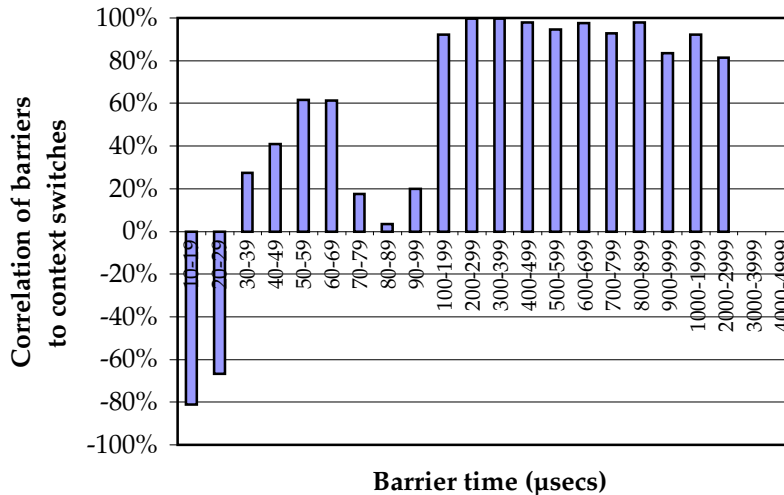


Figure 3.2: Correlation between the number of barriers in each time range and the number of context switches observed

context-switch activity. The data used for Figure 3.2 comes from the following experiment:

1. Perform a large number of back-to-back barrier operations, measuring each of them individually on each process.
2. Log the number of context switches that occurred during the entire run.
3. Tally the barrier times into a histogram (number of 10–19 µsec barriers, number of 20–29 µsec barriers, etc.).
4. Repeat with varying delays between barrier operations.

Figure 3.2 then plots the correlation of the vector of context switch tallies to the vector of 10–19 µsec barrier tallies, the correlation of the vector of context switch tallies to the vector of 20–29 µsec barrier tallies, and so forth up to the tally of 4–5 msec barriers.

The figure clearly shows that there is a large inverse correlation between the number of context switches observed on a given run and the number of barriers from that run that took the expected amount of time (10–29 µsecs) to complete. Similarly, there is an almost perfect correlation between the number of context switches observed on a given run and the number of barriers in that run that took 100–1000 µsec to complete. The conclusion we can draw is that context switches—or rather, the system services that the context switches to—are the likely cause of unresponsiveness, and that this unresponsiveness typically lasts for 100–1000 µsecs. The



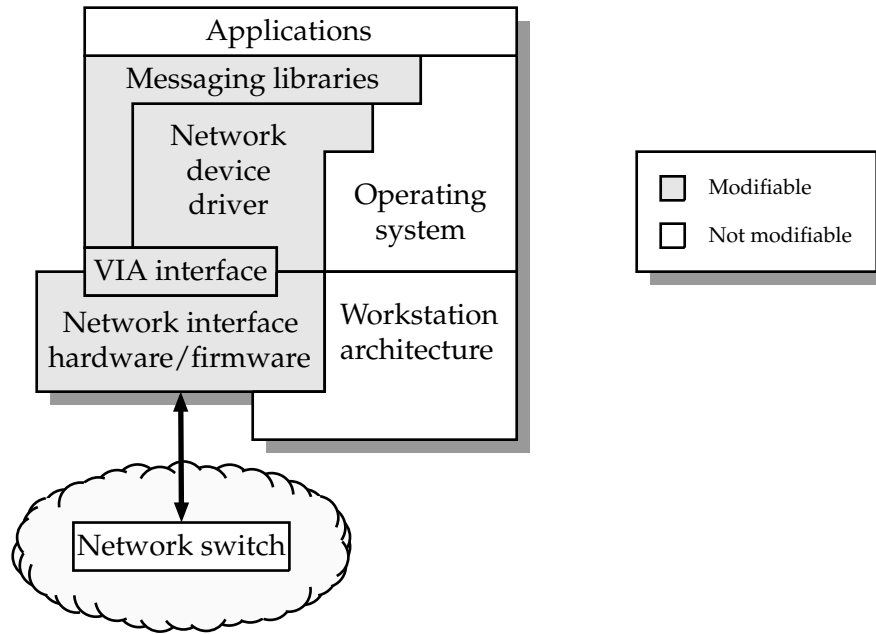


Figure 3.3: Components of an endpoint

strong correlation corroborates the claim that the best place to fight unresponsiveness is in the endpoints. Hence, my thesis will **tolerate unresponsiveness at the endpoints**.

Within the endpoints are a variety of hardware and software components that can potentially be modified to tolerate unresponsiveness. However, not all of these are appropriate to change. Figure 3.3 illustrates the components of an endpoint and differentiates between those that are reasonable to modify and those that are not. As the figure shows, I did not investigate any techniques to tolerate unresponsiveness that required changes to the following:

**Applications** Users are generally averse to modify their source code. In order to promote acceptance of my solution to the problem of endpoint unresponsiveness, my thesis must not require source-code modifications. Hence, approaches that require unique programming languages or new programming paradigms are ill-favored for my thesis.

**Operating system + workstation architecture** Because I am assuming a COTS environment, my work would have less impact if it required changes to the fundamental components of a system. Because PC clusters are quintessentially COTS, the use of COTS technology is a fundamental constraint in PC clusters.

Even while preserving the application source code, operating system internals, and the basic workstation architecture, there remain a number of endpoint components that are suitable for investigation:

- Messaging libraries
- Network device driver
- Network interface hardware/firmware
- Software interface to the NIC

Any of the above are reasonable to modify, as they all stand a chance of successfully counteracting endpoint unresponsiveness. Each of those components has access to both local node state and network state. This broad view can be exploited to make the system tolerant of unresponsiveness. Furthermore, any changes made to the components in the above list will not affect application source code, nor do they require turning a PC cluster into something it is not. Hence, it should be comparatively easy for my work to impact future PC cluster projects.

Communication is the key to tolerating unresponsiveness. That statement is actually tautologous; if there were no communication in an application, there would be no possibility of unresponsiveness, as unresponsiveness is defined in terms of communication. My thesis therefore focuses on **communication**. Communication can be categorized into two main divisions: point-to-point and collective. Both types of communication are important to parallel programs. However, collective communication is the better one for my thesis to focus on for two reasons, one technical, one not:

1. Collective communication provides more opportunity for optimization.
2. Collective communication is widely considered an important problem.

If one of the two processes involved in a point-to-point communication operation is unresponsive, the other process will be delayed. However, if one of the  $P$  processes involved in a collective-communication operation is unresponsive, *all* of the remaining processes—of which there could be any number—may be delayed. Hence, unresponsiveness has a greater potential impact on collective communication than on point-to-point communication. A second argument for why collective communication offers more opportunity for optimization is that collective communication is a higher-level way to reason about communication. Because the programmer makes

more information available to the system, the system is able to use that additional information to relax communication ordering, reschedule communication operations, and generally exploit the additional knowledge of future communication events.

Apart from the technical issues, collective communication is a better target than point-to-point communication, because improving collective-communication performance is an important concern to researchers and application scientists. According to an ongoing survey<sup>1</sup> of LAM/MPI users, 38% of the users surveyed wanted to see “faster/better optimized MPI collective functions.” 26% of users said they would use MPI-2’s extended collective-communication operations if implemented. (The LAM implementation of MPI does not currently support the MPI-2 specification.) And, on a question asking users to rank the features they would most like LAM/MPI to support, improved collective-communication performance was the most popular response. The conclusion to draw from this survey is that users do consider collective communication important and believe that their applications would run faster if collective-communication performance were improved.

Because of the opportunity for optimization and the importance of the issue, my thesis focuses on **collective communication**.

There are a large number of collective-communication operations, some of which were enumerated on page 2. For the purpose of tolerating unresponsiveness, however, barrier synchronization [53] is the most important collective-communication operation to study. First, barriers are important to performance. Proprietary meteorological codes studied at Cray Research were found to run 7% slower when, instead of using the T3E’s barrier hardware, they utilized a 15  $\mu$ sec-slower all-software barrier [93]. Second, barriers are the most challenging collective-communication operation in which to tolerate unresponsiveness. Because a barrier is a synchronization operation, it runs only as fast as the slowest (in this context, least responsive) participant. Hence, tolerating unresponsiveness in barriers may lead to understanding of the problem as a whole. Finally, barriers are a useful abstraction for parallel programming. They are commonly used to separate phases of an application, to ensure ordering and data consistency across processes. In fact, barriers are the central abstraction in Valiant’s bulk-synchronous model of computing [101]; no nontrivial bulk-synchronous program can exist without them. Barriers are also used extensively in single program, multiple data (SPMD) models, such as High-Performance Fortran (HPF); compilers generally insert a barrier after every parallel loop (FORALL in HPF). Because of the importance of barrier operations

---

<sup>1</sup>The survey is located at [http://www.mpi.nd.edu/lam/user\\_survey/](http://www.mpi.nd.edu/lam/user_survey/). The results highlighted in this dissertation correspond to survey results dated January 24, 2001 and earlier.

within the class of collective-communication operations, my thesis focuses on **barriers**.

As stated, the approach my thesis takes is to tolerate unresponsiveness in the endpoints by focusing specifically on barrier operations and without requiring application modifications or violating the cluster's COTS properties. Within this solution space, the key assumptions my thesis makes are:

- The communication subsystem has no *a priori* knowledge of how long each communication operation will take, nor of the expected time between operations.
- The operating system lacks support for real-time scheduling.<sup>2</sup>
- Applications are not heavily multithreaded. That is, they do not always have more work to do while waiting for a remote process to become responsive. (Other applications on the same node may have other work to do, however.)
- There is not necessarily any coordinated thread scheduling across nodes.

Although my thesis will not assume coordinated thread scheduling, its presence could probably still improve application performance. That is, coordinated thread scheduling is complementary to unresponsiveness tolerance. However, coordinated scheduling does not replace unresponsiveness tolerance because it eliminates only unresponsiveness external to the application, not unresponsiveness caused by the application structure itself.

### 3.4 Thesis statement

My thesis is stated as follows:

Parallel application performance can be improved by tolerating unresponsiveness at the endpoints. By altering the implementation of collective-communication operations, especially barriers, this goal can be achieved without requiring application modifications and without violating the COTS properties of PC clusters.

---

<sup>2</sup>Here, "real time scheduling" means using a real-time scheduling algorithm such as Earliest Deadline First [70], as opposed to merely supporting a fixed-priority scheduling class with a higher base priority than the timeshared class, as is common in modern commercial operating systems such as Windows NT [29] and Solaris [98].

### 3.5 Success criteria

The following are the criteria to use to determine whether I have successfully validated my thesis:

1. Have I developed a technique to tolerate unresponsiveness?
2. Does my technique improve performance?
3. Does my technique preserve the cluster's COTS characteristics?
4. Does my technique preserve the application's source code?

In short, I must demonstrate that a new technique that solves the performance penalty caused by unresponsiveness and does so without sacrificing the benefits of commodity-based PC clusters, such as the basic architecture or the ability to use an off-the-shelf operating system.

[W]e demand rigidly defined areas  
of doubt and uncertainty!

*Douglas Adams*

THE HITCHHIKER'S GUIDE TO THE  
GALAXY, 1979

If you think the problem is bad now,  
just wait until we've solved it.

*Arthur Kasspe*

# 4 Nonblocking Barriers

Whereas Chapter 3 framed the problem and the space in which my thesis devises a solution, Chapter 4 describes the specific approach I took to solve the problem. The primary technique I developed, and the key contribution of this thesis, is a new mechanism called a “nonblocking barrier.” Nonblocking barriers enable unresponsiveness that is observed during a barrier operation to be tolerated. Of course, nonblocking barriers meet the success criteria stated in Section 3.5.

Chapter 4 describes the basic concept underlying nonblocking barriers (Section 4.1). It presents the detailed nonblocking-barrier algorithm and provides an example of how the algorithm works (Sections 4.2 and 4.3). Section 4.4 delves into more detail regarding the nonblocking barriers’ ordering semantics and draws an analogy between those semantics and the semantics of release-consistent shared memory. The chapter then describes alternative implementations and their relative merits and shortcomings (Section 4.5). Section 4.6 presents alternatives to nonblocking barriers and explains how they fail to meet this dissertation’s stated success criteria. Finally, Section 4.7 elaborates further on some of the points made in this chapter.

## 4.1 Introduction

A barrier is a synchronization operation. Its semantics dictate that no process may leave the barrier until all processes have entered the barrier. One common way to implement a barrier is to have each participating process synchronize with each of its peers in a butterfly network (Figure 4.1).<sup>1</sup> More formally, in each stage  $s \in [0, \lceil \lg P \rceil - 1]$ , each process,  $p \in [0, P - 1]$ , exchanges a message with process  $p \oplus 2^s$ . Figure 4.2 illustrates the problem with this traditional approach: If a single process is unresponsive, the entire barrier operation’s completion is detained until the unresponsive process becomes responsive again. Even worse, it can take up to  $\lg P$  additional steps to complete the barrier (during which time, any other process may become unresponsive, as well).

The key insight my thesis makes is that barrier semantics are actually stricter than necessary. The crucial observation is that, in the absence of communication, it is not possible to detect whether any of the participants have entered or left the bar-

---

<sup>1</sup>An alternative is to perform a null reduction followed by a null broadcast.

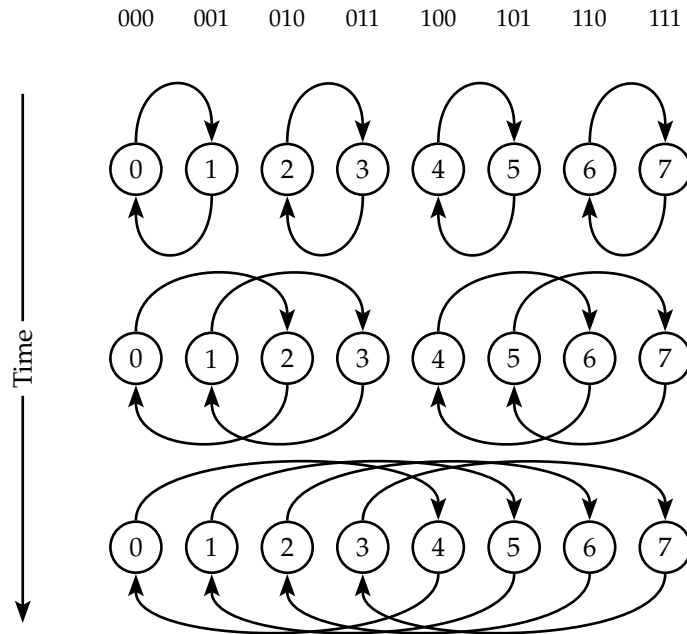


Figure 4.1: Basic, logarithmic-time, 8-process barrier

rier or whether synchronization has occurred. It is therefore permissible to allow processes to run ahead of the barrier, as long as there is no telltale communication that would enable other processes to detect that. As will be described shortly, non-blocking barriers relax the normally strict barrier semantics, but do so in such a way that they are indistinguishable from traditional barriers, in the absence of hidden channels [61].

The novelty of altering barrier semantics surreptitiously is that it requires no application modifications. Barrier semantics are changed only conservatively. In a sense, nonblocking barriers provide “covert multithreading”. Like ordinary multithreading, they enable work that would normally be performed after a barrier completes to overlap barrier idle time induced by an unresponsive peer. However, the programmer does not realize that his application is being multithreaded automatically behind the scenes. He can therefore reason about his program as if it still used its original, simpler, single-threaded semantics.

The closest piece of related work to my thesis work is Nicol’s noncommittal barrier synchronization work [79]. Nicol also observes that a single slow participant can greatly degrade barrier performance. In his solution, a process can exit a barrier and continue computing as soon as it has completed its pre-barrier computation. If the process later determines that barrier semantics were violated—by pre-barrier work arriving after exiting the barrier—the process handles its pre-barrier work, rolls the

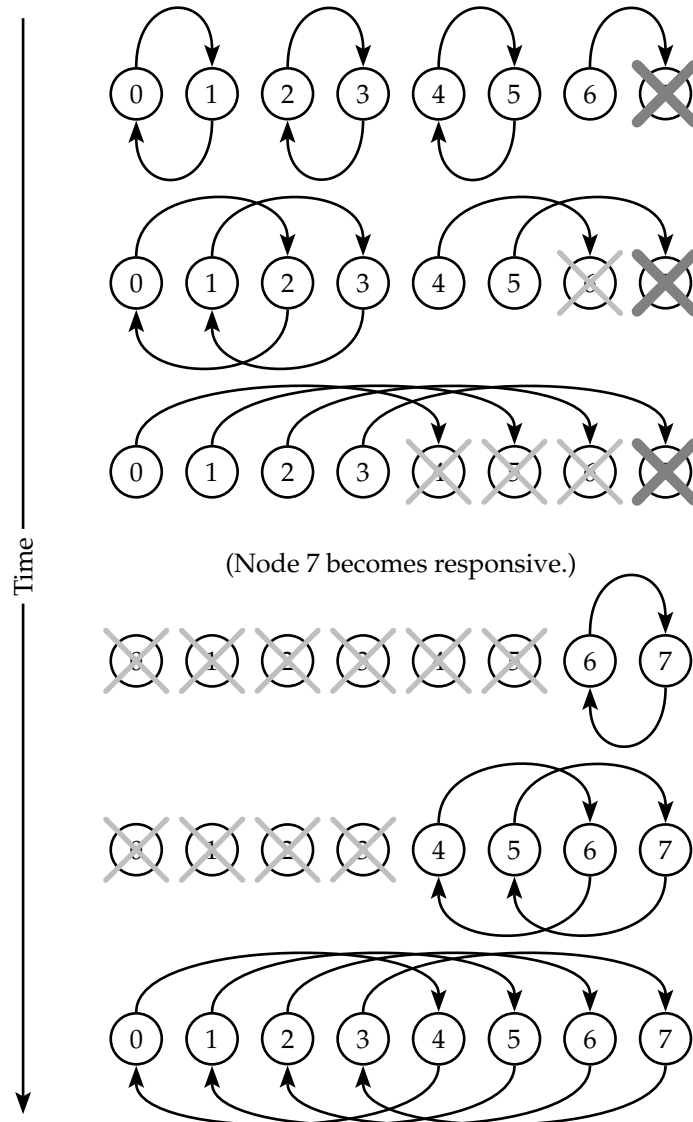


Figure 4.2: Basic, logarithmic-time, 8-process barrier, in which process 7 is temporarily unresponsive



entire application back to its pre-barrier state, and retries the barrier. This process repeats as many times as necessary until no pre-barrier work arrives after the barrier. The crucial difference between my thesis work and noncommittal barrier synchronization is that the latter requires application modifications, while my work does not. As iterated repeatedly in Chapter 3, a requirement for community acceptance of a new technique is that it not require source-code modifications. Noncommittal barrier synchronization generally requires extensive modifications. Applications must be rewritten to periodically checkpoint their state, roll the entire distributed computation state back to a previous checkpoint, and possibly even garbage-collect states that are known never to be needed again. None of these tasks are simple, and not all applications can be written in such a style.

A second piece of related work is MAGPIE [58], from Vrije Universiteit. MAGPIE examines supporting collective communication efficiently across both clusters and wide-area networks. It uses one set of communication algorithms within a cluster, to maximize parallelism, and a different set between clusters, to minimize bandwidth demands. This is similar to my thesis work if one draws an analogy between “local cluster” and “responsive” and between “remote cluster” and “unresponsive.” The difference is that MAGPIE treats unresponsiveness as a static characteristic: some processes are *always* unresponsive; others are *always* responsive. In contrast, my thesis proposes that unresponsiveness is a transitory condition; any process can become unresponsive at any time and for an arbitrary length of time.

A final example of related work is the broad category of load balancing. Charm++/Converse [55, 56] is one instance of a system with load balancing integrated into the runtime system. The user writes fine-grained object-oriented code in Charm++, a distributed, C++-like language in which objects communicate with explicit message passing. The runtime system then exploits global system state, observations of objects’ communication patterns, and application-provided triggers to periodically load balance the application across processors. While load-balancing schemes such as Charm++/Converse and my thesis work all try to solve the problem of unresponsiveness, the crucial difference is that the former tackles unresponsiveness by *preventing* it, while my thesis is designed to *tolerate* it.

Before describing how nonblocking barriers work (in the following sections), it is important to discuss the communication properties needed for their implementation. The following are the requisite properties:

- the ability to intercept and process a message before delivery to the application or after transmission from the application,
- the ability to attach a small amount of metadata to a message,

- the ability of a receiver to identify a message’s sender and peer group,
- FIFO message ordering between pairs of processes, and
- reliable message delivery.

The first property is the most important, as it can be used as a primitive to fabricate all of the remaining properties. Because PC clusters are composed of commodity components, it is common for a message to pass through a few layers of software and firmware before being injected into the network. By modifying any of these layers, nonblocking barriers can intercept a message for additional processing before or after it passes through the network. This property is somewhat unique to clusters. Parallel computers such as the CM-5 [100] and T3E [93], in contrast, integrate the network high up in the memory hierarchy, giving applications direct access to it. While this improves latency and possibly, bandwidth, it implies that the only way to intervene between two communicating processes on such a system is to modify the custom, vendor-specific communication hardware.

Once intervention is possible, the remaining communication properties shown in the above list can all be supplied, if they are not already present. Nonblocking barriers use message metadata to store information about the sender of a message and about the other processes involved in the same collective-communication operation. Inserting metadata into a message is trivial if the underlying layer supports variable-sized messages. Even if it does not, the higher-level messages can be segmented into sufficiently small pieces on the sending side and recombined on the receiving side. This is the mechanism by which UDP datagrams [88], which can be up to 65,535 bytes long, can be sent over an Ethernet [74] network, in which the maximum frame size is only 1536 bytes.

Nonblocking barriers assume that all messages that are sent will eventually be received. Furthermore, reception order must be FIFO between a given sender and a given receiver. Most modern high-speed networks support FIFO delivery and have a sufficiently low bit error rate that they can be considered reliable for most purposes. The VIA interface lets the systems programmer decide the tradeoff between message overhead and prompt—or any—notification of bit errors [24]. If guaranteed ordered, reliable delivery is essential, it is possible to fabricate it even on networks that support neither by sending sequence information as message metadata [63].

As my thesis work was carried out in the context of VIA, it is instructive to note which of the communication requirements shown in the above list are supported by VIA and which are not. First, there is ample opportunity to intercept an application’s communication before it reaches the network. Because VIA is merely an

interface specification, it is left to the implementation as to which components are implemented in hardware and which in software. However, few programs access the VIA interface directly; most utilize a higher-level API, such as MPI [73] or the Berkeley sockets interface [66]. Hence, messages can be intercepted in hardware, firmware, or the network device driver. Second, VIA supports gather and scatter of message data between the network and local memory, which makes metadata attachment easy. Third, the VIA specifications dictate that message order is FIFO. And finally, my thesis work uses the VIA dropped-packet notification to ensure that all messages are delivered.

## 4.2 Algorithm

First, we present some terminology:

**send** Specify that a message is to be sent at some arbitrary time in the future (i.e., *post* a send without necessarily *completing* it).

**receive** Accept a message at the destination node, without yet making the message available to the destination process.

**deliver** Make a received message available to the destination process.

Note that the above are the standard distributed-system definitions of *receive* and *deliver*—as used, for example, by Babaoğlu and Marzullo [83]—as opposed to the permuted definitions used by the VIA committee [24].

To preserve traditional collective-communication semantics while still being able to execute collective-communication operations optimistically, we must impose the following invariant on any mechanism we develop:

**Invariant 1** *All sends posted after a barrier is must be delivered after the barrier.*<sup>2</sup>

The above implies that messages can be sent and received at any time. However, a message can be delivered only in the same or a later phase<sup>3</sup> of the program than the one in which it was sent.

We can now state the key concept underlying this work: a process can continue computing and communicating after entering a barrier, but it is forbidden from *delivering* messages until all other processes have entered the barrier as well. In a traditional barrier, no process may leave the barrier until all processes have entered

---

<sup>2</sup>Also true, but less useful in the context of this dissertation, is the invariant that all sends delivered *before* a barrier must have been posted *before* the barrier.

<sup>3</sup>Here, “phase” means a barrier-delimited sequence of operations, akin to a superstep in Valiant’s bulk-synchronous model [101].

the barrier. Nonblocking barriers merely relax this constraint, while still honoring the standard barrier semantics by maintaining the invariant stated on the previous page.

We can now turn to a specific algorithm for implementing nonblocking barriers. One way to implement nonblocking barriers is for each process to keep track of the current time with something similar to an array of logical clocks [61], with “barrier” being the event that increments the logical time. Each process holds one “send” and one “receive” clock per peer. The send clock is incremented every time a process starts a barrier that involves the given peer. The send time is included in every message sent to that particular peer. The receive clock is incremented every time a process completes a barrier that involves the given peer. No point-to-point message can be delivered until its timestamp is less than or equal to the receive clock corresponding to the sender.

To tolerate unresponsiveness, the nonblocking-barrier scheme relies on a *message-driven* style of communication [54]. When an application program invokes the `Barrier()` function, the communication library merely initiates the first stage of the barrier operation and returns immediately. When a process receives a barrier message, the message contains enough information for that process to continue with the next stage. Algorithms 4.1 and 4.2 list, respectively, the pseudocode for nonblocking barriers proper and for the associated message-receive function.

The algorithms work as follows. The barrier function proper (Algorithm 4.1) starts by filling in all the fields of a barrier message (shown in Figure 4.3(b)). It then sends the message to its first peer, and increments the tally of the number of barriers started that involve each peer.<sup>4</sup> The **final if statement** in Algorithm 4.1 may be a bit counterintuitive. It increments each peer’s receive timestamp if the barrier that was just *started* had previously *finished*. This is necessary because nonblocking barriers enable an unresponsive process to receive notification that some number of its peers have completed a barrier even before the unresponsive process entered the barrier. (As will be described on the following page, the message receive function contains the opposite test: It increments each peer’s receive timestamp if a barrier that just *finished* had previously *started*.) As soon as the barrier function sends the first message and increments the various counters, it returns to the caller, enabling it to make progress in its computation.

Algorithm 4.2 is the nonblocking receive function that corresponds to the barrier function in Algorithm 4.1.<sup>5</sup> The algorithm starts by receiving a message from the network, returning NULL if none is available. There are then two cases: Either

---

<sup>4</sup>The send timestamp,  $TS_{\text{send}}$ , is an alias for *barriers\_started*.

<sup>5</sup>A blocking version of the receive function, not shown in this dissertation, is almost identical.

---

**Algorithm 4.1** Unresponsiveness-tolerant barrier

---

**Given:** *Participants* (list of IDs)

▷ Find our and our first peer's offsets into *Participants*, and fill in all the fields in an outgoing message (*m*).

```
for  $p \leftarrow 0$  to  $|Participants| - 1$  do  
   $m.participants[p] \leftarrow Participants[p]$   
  if  $Participants[p] = selfID$  then  
     $self\_ofs \leftarrow p$   
  end if
```

```
end for
```

```
 $first\_peer \leftarrow self\_ofs \oplus 1$ 
```

```
if  $first\_peer \geq |Participants|$  then
```

```
  return
```

```
end if
```

```
 $m.type \leftarrow BARRIER$ 
```

```
 $m.stage \leftarrow 0$ 
```

```
 $m.source \leftarrow self\_ofs$ 
```

```
 $m.dest \leftarrow first\_peer$ 
```

```
 $m.numparticipants \leftarrow |Participants|$ 
```

```
 $m.TS \leftarrow barriers\_started[peerID]$ 
```

▷ Send the message, and update the appropriate timestamps.

```
Send  $m$  to  $Participants[first\_peer]$ 
```

```
for  $p \leftarrow 0$  to  $|Participants| - 1$  do
```

```
   $peerID \leftarrow Participants[p]$ 
```

```
   $barriers\_started[peerID] \leftarrow barriers\_started[peerID] + 1$ 
```

```
  if  $barriers\_started[peerID] \leq barriers\_finished[peerID]$  then
```

```
     $TS_{recv}[peerID] \leftarrow TS_{recv}[peerID] + 1$ 
```

```
  end if
```

```
end for
```

---

the message is a point-to-point message or a barrier message. In the former case, the receive function enqueues the new message and then returns the oldest message in the queue, but only if its timestamp is sufficiently recent. This queueing is necessary because one of the goals of my thesis work is to improve performance while preserve existing communication semantics—both collective and point-to-point. Because VIA guarantees ordered delivery, returning a newer message when an older one is available would violate those semantics.

If the incoming message is a barrier message, Algorithm 4.2 increments (in place) the stage number. If there are more stages remaining in the barrier, the receive function forwards the barrier message onto the next peer. If this is the final stage of the barrier (i.e., stage  $\lceil \lg(m.numparticipants) \rceil$ ), the receive function increments its tally of the number of barriers that finished for each peer. For each finished barrier that previously started, the peer's receive timestamp is incremented.

---

**Algorithm 4.2** Nonblocking receive that supports unresponsiveness-tolerant barriers

---

```

if message is available from network then
   $m \leftarrow$  message from network
else
  return NULL
end if

if  $m.type = PT2PT$  then
  ENQUEUE( $Q, m$ )
   $m' \leftarrow$  PEEK( $Q$ )
  if  $m'.TS > TS_{recv}[m.src]$  then
    return NULL
  else
    return POP( $Q$ )
  end if

else if  $m.type = BARRIER$  then
   $m.stage \leftarrow m.stage + 1$ 
  if  $2^{m.stage} < m.numparticipants$  then
    ▷ Internal barrier stage
     $m.src \leftarrow m.dest$ 
     $m.dest \leftarrow m.dest \oplus 2^{m.stage}$ 
    if  $m.dest < m.numparticipants$  then
      Send  $m$  to  $m.participants[m.dest]$ 
    end if
  else
    ▷ Final stage of the barrier
    for  $p \leftarrow 0$  to  $numparticipants$  do
       $peerID \leftarrow Participants[p]$ 
       $barriers\_finished[peerID] \leftarrow barriers\_finished[peerID] + 1$ 
      if  $barriers\_started[peerID] \leq barriers\_finished[peerID]$  then
         $TS_{recv}[m.peerID] \leftarrow TS_{recv}[m.peerID] + 1$ 
      end if
    end for
  end if
end if

```

---

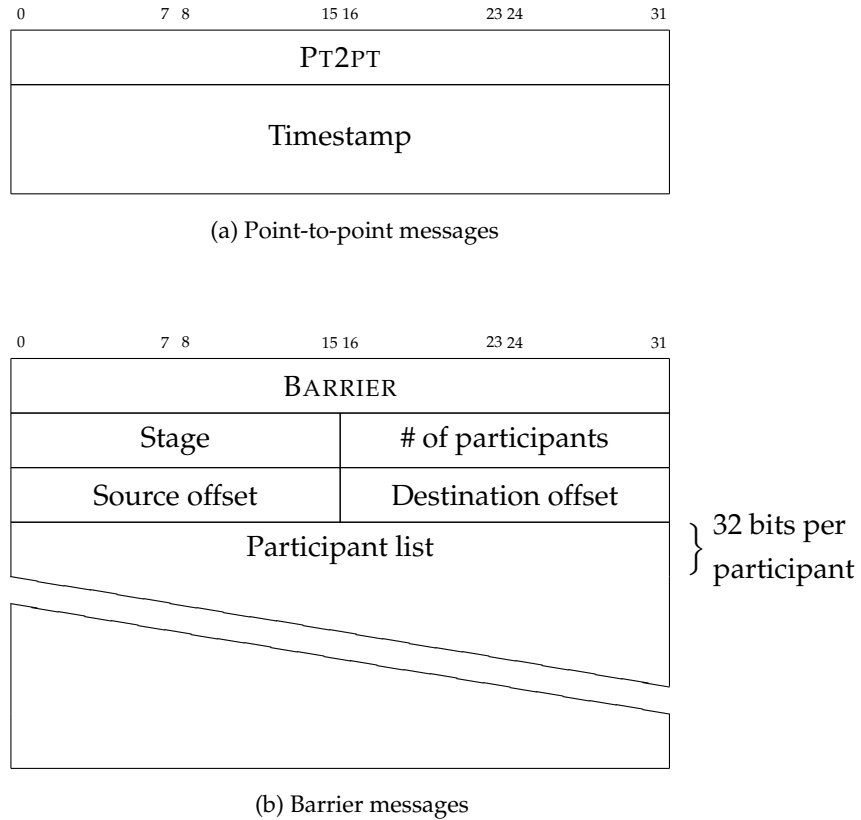


Figure 4.3: Packet formats used for nonblocking barriers

### 4.3 Example

Consider the following sequence of events, performed among processes A, B, C, and D:

1. Program initializes.
2. Processes A and B synchronize.
3. Processes C and D synchronize.
4. Processes B and C synchronize.

Figure 4.4 illustrates the bookkeeping that is performed when those events are performed using nonblocking barriers. In the figure, the last synchronization time between each pair of processes (i.e.,  $TS_{\text{send}}$  and  $TS_{\text{recv}}$ ) is shown to the right of each step. When the program initializes, all logical clocks are reset to 0. After processes A

and B synchronize, they each increment their logical clocks to indicate that the last time the two of them synchronized, it was before logical time 1. Processes C and D do likewise after the two of them synchronize. Finally, processes B and C increment their logical clocks again after they synchronize with each other.

With the bookkeeping illustrated in Figure 4.4, if process B sends a point-to-point message to process A after the sequence completes, the message will be timestamped at logical time 1—the logical time at which processes A and B last synchronized—and will be received no earlier than logical time 1. Therefore, process A can correctly deliver process B’s message, even though process B has been involved in two barriers total, versus process A’s one. The significance of process A’s being able to send a point-to-point message to process B in the context of this example will become apparent in Section 4.5.1.

## 4.4 Ordering semantics

The insight behind nonblocking barriers is that they relax some of the strict ordering constraints imposed by traditional barriers. This section details the specific rules that nonblocking barriers use to tolerate unresponsiveness. First, we present some definitions. Then, we state the reordering rules and argue that the rules are analogous to those used by release-consistent shared memory [42]. Finally, we discuss the implications of these rules on user applications.

### 4.4.1 Definitions

In the context of shared-memory consistency models, it is common to describe when a memory access is *performed*. The analogue in the context of my research is when a collective-communication operation—specifically, a barrier, reduction, or multicast—has *initiated* or *completed*. It is important to separate initiation and completion when discussing nonblocking barriers, because nonblocking barriers, unlike traditional barriers, are a split-phase operation; they allow other work to proceed between initiation and completion. However, only initiation is application-visible. Nonblocking barrier completion occurs asynchronously and is triggered by message arrival. The following are the definitions for “initiated” and “completed” that will be used on subsequent pages to describe the rules for ordering semantics:

- A process considers a multicast to be completed if:
  1. it is the root of the multicast, and it has sent the message to its immediate children, or



1. Program initializes.

|   | A | B | C | D |
|---|---|---|---|---|
| A | 0 | 0 | 0 | 0 |
| B | 0 | 0 | 0 | 0 |
| C | 0 | 0 | 0 | 0 |
| D | 0 | 0 | 0 | 0 |

2. Processes A and B synchronize.

|   | A | B | C | D |
|---|---|---|---|---|
| A | 1 | 1 | 0 | 0 |
| B | 1 | 1 | 0 | 0 |
| C | 0 | 0 | 0 | 0 |
| D | 0 | 0 | 0 | 0 |

3. Processes C and D synchronize.

|   | A | B | C | D |
|---|---|---|---|---|
| A | 1 | 1 | 0 | 0 |
| B | 1 | 1 | 0 | 0 |
| C | 0 | 0 | 1 | 1 |
| D | 0 | 0 | 1 | 1 |

4. Processes B and C synchronize.

|   | A | B | C | D |
|---|---|---|---|---|
| A | 1 | 1 | 0 | 0 |
| B | 1 | 2 | 1 | 0 |
| C | 0 | 1 | 2 | 1 |
| D | 0 | 0 | 1 | 1 |

Figure 4.4: Bookkeeping for a sample sequence of operations

2. it is not the root of the multicast, and it has received and delivered the multicasted message.
- A process considers a reduction to be completed if:
    1. it is the root of the reduction, and it has received and delivered the reduced value(s), or
    2. it is not the root of the reduction, and it has sent its data to its parent.
  - A process considers a barrier to be initiated after the first barrier message is sent to the process' first peer.
  - A process considers a barrier to be completed after the final barrier message is received and delivered from the process' final peer.

In the above, "sent" means that a message transmission was posted, although not necessarily received by the destination. While the preceding definitions make no mention of point-to-point operations, "send"+"receive" can be considered a single, paired operation that is a special case of either multicast or reduce.

#### 4.4.2 Reordering rules

The reordering rules for nonblocking barriers can be mapped directly onto the reordering rules for release-consistent shared memory. This is an important result, because it shows that nonblocking barriers make "intuitive" sense. That is, reasoning about nonblocking barriers' ordering semantics is not substantially different from reasoning about release consistency's ordering semantics.

Table 4.1 illustrates the correspondence between the operations used in release-consistent shared memory and those used in the context of nonblocking barriers. Table 4.2 then does the same thing for the reordering rules in the two contexts.

Table 4.1: Analogous operations

| Release consistency | Nonblocking barriers          |
|---------------------|-------------------------------|
| LOAD                | REDUCE                        |
| STORE               | MULTICAST                     |
| ACQUIRE             | BARRIER completion (implicit) |
| RELEASE             | BARRIER initiation            |

Table 4.2: Ordering semantics for nonblocking barriers

| Release consistency   | Nonblocking barriers  |
|---|---|
| Before an ordinary LOAD or STORE access is allowed to perform with respect to any other processor, all previous ACQUIRE accesses must be performed. | Before an ordinary MULTICAST or REDUCE operation is allowed to complete, all previous BARRIER operations must have completed. |
| Before a RELEASE access is allowed to perform with respect to any other processor, all previous ordinary LOAD and STORE accesses must be performed. | Before a BARRIER operation is allowed to initiate, all previous MULTICAST and REDUCE operations must have completed.          |
| Special accesses (ACQUIRES, RELEASES, and chaotic accesses) are processor consistent with respect to one another.                                   | BARRIER initiations and completions are processor consistent with respect to one another.                                     |

While the first two rules in Table 4.2 are straightforward, the final rule needs some additional explanation. The definition of processor consistency is that all processors agree on the order of STORE accesses from a given processor, but may disagree on the order of STORE accesses from different processors [42, 77]. Figure 4.5 illustrates how the same concept applies to nonblocking barriers. The figure shows a timeline of two concurrent barrier operations. Barrier 1 involves processes 0–3; barrier 2 involves processes 2–5. Barrier messages are labeled  $b_1$  and  $b_2$  to indicate which barrier they correspond to. Barrier initiations are labeled “(I1)” or “(I2)”, and barrier completions are labeled “(C1)” or “(C2)”.

The important observation to make from Figure 4.5 is that various processes disagree on the ordering of certain events. For instance, process 2 sees barrier 1 complete before barrier 2, while process 3 sees barrier 2 complete before barrier 1. (Recall that a process increments its logical clock only when it has both initiated and completed a particular barrier. The initiation and completion operations can occur in either order, however.) Furthermore, process 2 completes barrier 2 before initiating barrier 1, while process 3 initiates barrier 1 before completing barrier 2. Because of those discrepancies, nonblocking barriers are clearly not sequentially consistent [62], as that requires global agreement on the order that the barrier initiations and completions occur. However, all processes do agree on the order of barrier initiations from any given process. For instance, processes 2 and 3 agree that process 3 initiated barrier 1 before barrier 2.

We can therefore say that the components of a barrier operation—initiation and completion—are effectively processor consistent. And together with the remaining nonblocking-barrier rules in Table 4.2—essentially the release-consistency rules

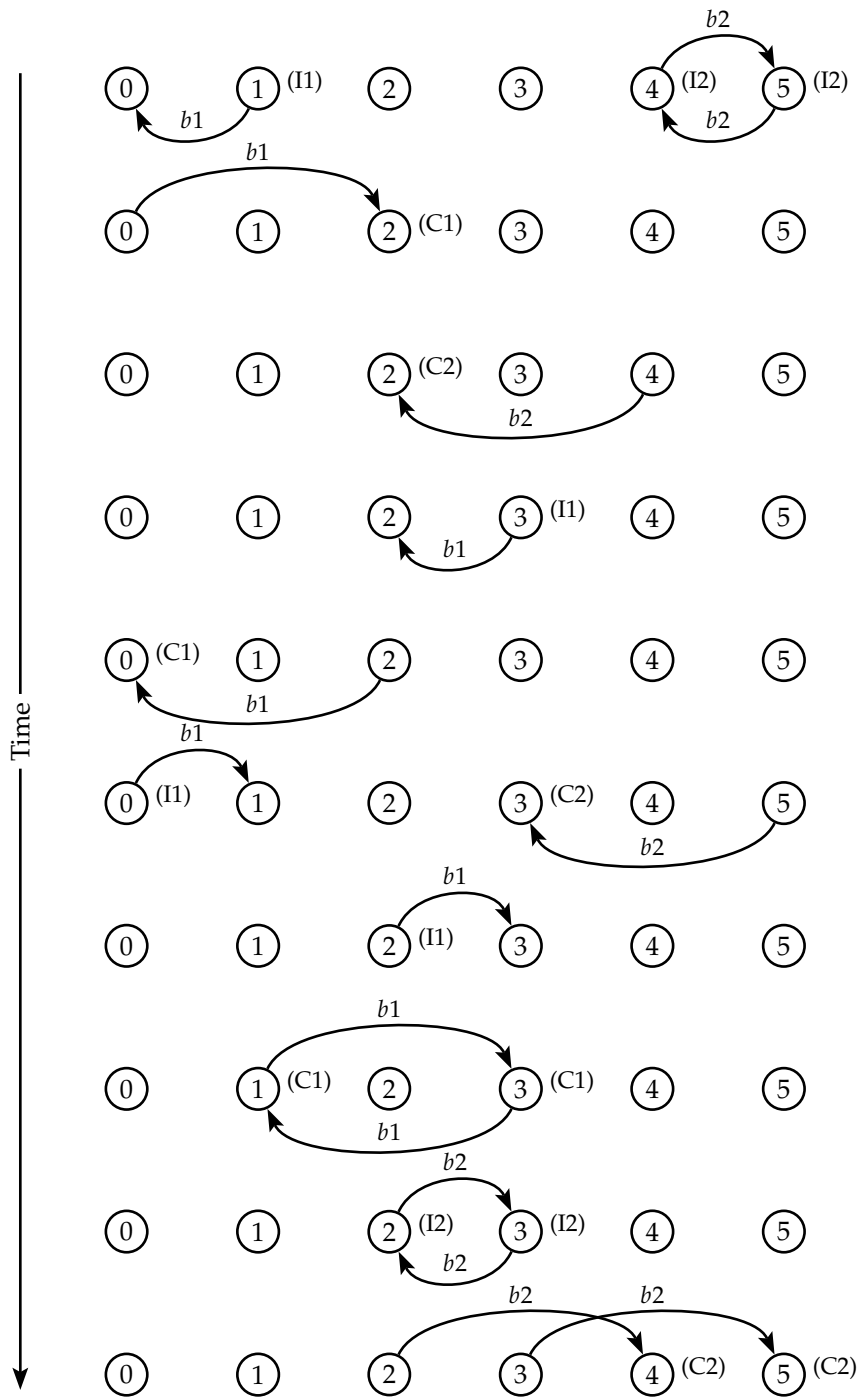


Figure 4.5: Processor consistency in the context of nonblocking barriers

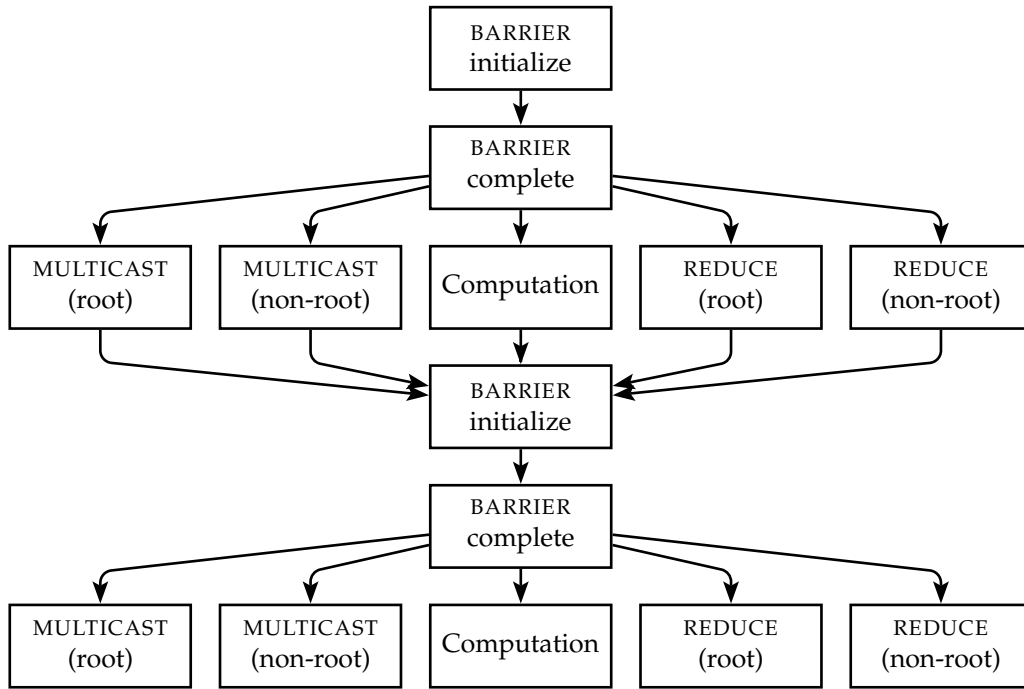
with the terms from Table 4.1 substituted in—we can conclude that a system using collective communication operations along with nonblocking barriers is effectively release consistent.

### 4.4.3 Implications

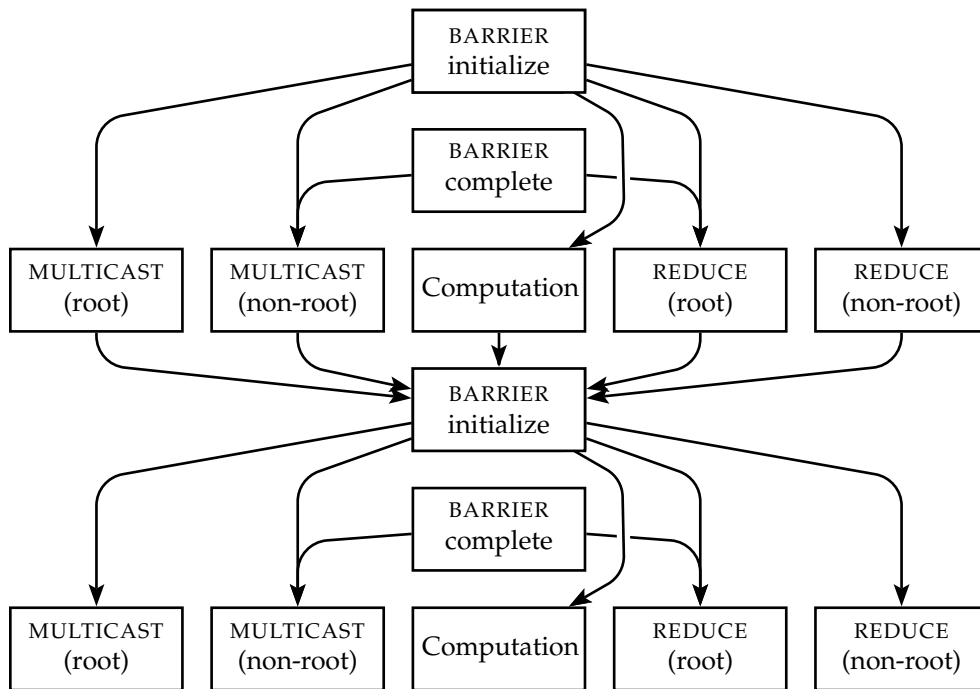
The result of the ordering semantics listed in Table 4.2 is that certain operations can execute between barrier initiation and barrier completion. Specifically, computation and fan-out collective-communication operations—multicasts, from the root’s perspective, and reductions, from the perspective of all processes except the root—can execute as soon as a barrier is initiated, without having to wait for the barrier to complete. Figure 4.6 contrasts the ordering requirements for traditional and nonblocking barriers. When traditional barriers are used (Figure 4.6(a)), processes make no progress between barrier initiation and completion. Once a barrier completes, then other communication and computation can proceed (in program order). However, when nonblocking barriers are used (Figure 4.6(b)), barrier completions are decoupled from barrier initiations and execute asynchronously. The root of a reduction must still wait for a barrier to complete before the reduction can complete. However, all the other processes in the reduction can send their data and immediately go on. Similarly, the root of a multicast does not have to wait for a preceding barrier to complete, although the non-roots do.

One implication of the ordering semantics for nonblocking barriers is that the applications that stand the most to gain from nonblocking barriers are those that do the most work between a barrier and the subsequent fan-in operation. Figure 4.7 shows a sample timeline illustrating a “good” sequence of operations for nonblocking barriers: BARRIER, SEND (representing any fan-out operation), COMPUTE, and RECEIVE (representing any fan-in operation). In Figure 4.7(a), the system is responsive. Hence, barriers reach completion quickly, so there is minimal (but still some) opportunity for operations to overlap with the barrier. In Figure 4.7(b), some set of processes is unresponsive. Traditional barriers must therefore block until the unresponsive processes become responsive again. In contrast, nonblocking barriers can execute SEND operations and perform computation during the otherwise idle time. RECEIVE operations must still block until the barrier has completed.

The *worst-case* situation for nonblocking barriers, based on their ordering semantics, is when messages are received immediately after a barrier. This is, unfortunately, a common occurrence in a number of applications, which use barriers as a fence operation for messages—essentially a promise that no more messages will be



(a) Traditional barriers



(b) Nonblocking barriers

Figure 4.6: Dependencies between barrier and non-barrier operations

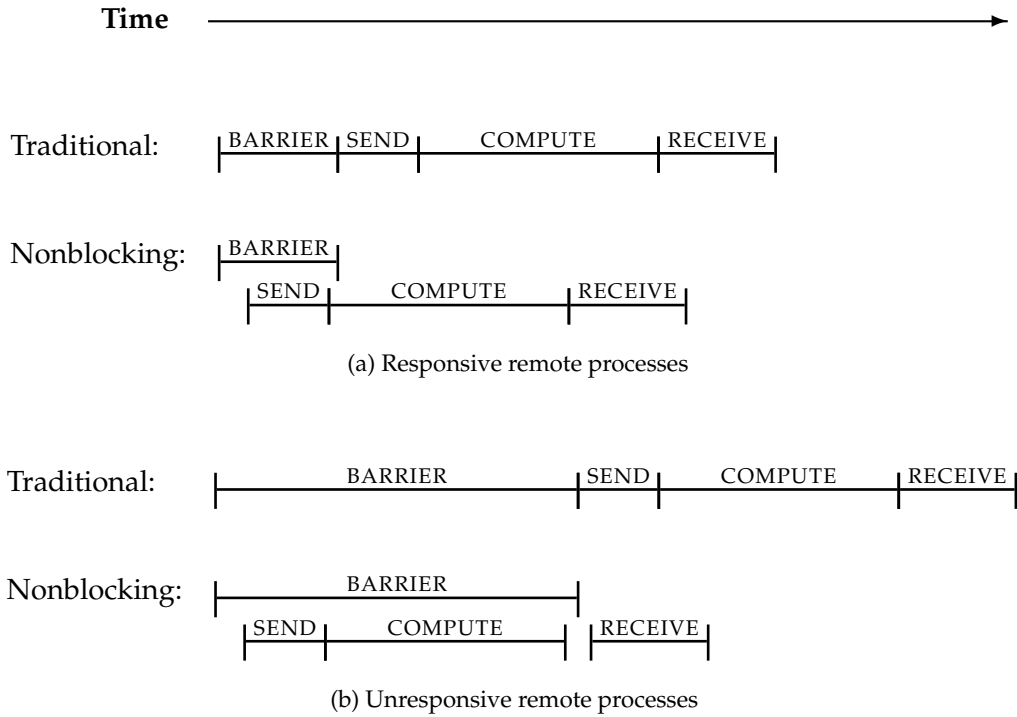


Figure 4.7: Timelines for traditional and optimistic collective communication

forthcoming in that iteration of the program. The main loop of such applications is as follows:

1. Send a number of point-to-point messages
2. Synchronize all processes with a barrier
3. Receive the point-to-point messages
4. Compute based on the data just received

Nonblocking barriers are unable to improve performance in that case, because there are no fan-out operations between the barrier and the subsequent receives.

As indicated by Table 4.2 on page 39, when writing an application, a programmer ought to be able to reason about nonblocking barriers on a PC cluster in much the same way that he can reason about release consistency in a shared-memory machine. By extension, the same types of applications designed to perform well (not to mention, run correctly) in a release-consistent shared-memory system are likely to observe better performance in a PC cluster with nonblocking barriers than with traditional barriers.

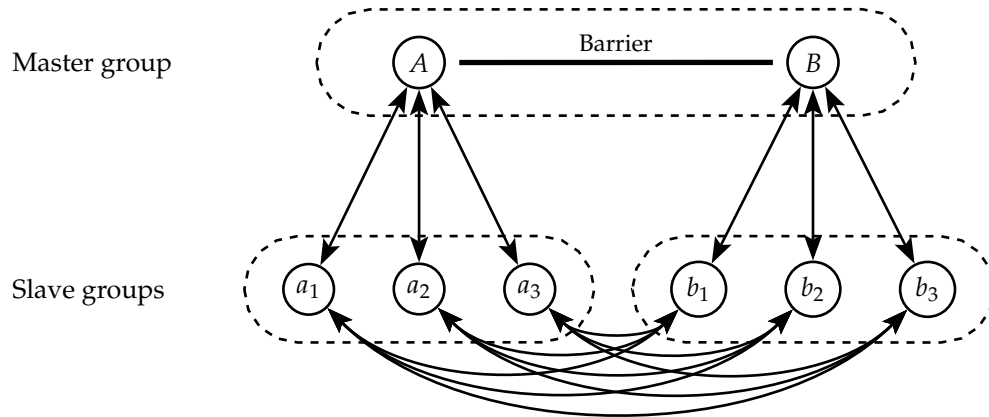


Figure 4.8: Prohibited group-communication pattern

#### 4.4.4 Restrictions

There are two restrictions on programs that utilize nonblocking barriers:

1. There can be no hidden communication channels [61].
2. Synchronization must be explicit if processes communicate across branches of a hierarchical group [13].

If a program violates either of those constraints, it may observe incorrect behavior when traditional barriers are replaced with nonblocking barriers. Communication over hidden channels (e.g., the filesystem) is invisible to the nonblocking-barrier algorithm. As stated on page 29, the ability to intercept, modify, and possibly, delay messages is fundamental to nonblocking barriers. Without that ability, the algorithm cannot maintain Invariant 1 (page 31), so messages sent after a barrier might, incorrectly, be delivered before the barrier.

The second restriction, that synchronization must be explicit if processes communicate across branches of a hierarchical group, is more complicated and requires some explanation. Consider a program in which processes  $A$  and  $B$  form a master group, processes  $a_1$ ,  $a_2$ , and  $a_3$  are slaves to  $A$ , and processes  $b_1$ ,  $b_2$ , and  $b_3$  are slaves to  $B$ . Suppose that the program performs the following communication operations, which are illustrated in Figure 4.8:

- At the start of a phase, each master multicasts work to its slaves.
- The work assigned to a slave may involve point-to-point communication across slave groups (e.g., from one of  $A$ 's slaves to one of  $B$ 's).
- Each master waits until all of its slaves have completed their tasks (e.g., by using a reduction operation).



- The masters perform a barrier to separate one phase from the next.

Given those communication operations, it is possible when using nonblocking barriers for a point-to-point message sent in one phase to arrive, incorrectly, while the recipient is still in an earlier phase. The following sequence of events illustrates how this missequencing can occur:

1.  $A$  and  $B$  multicast Phase 1 work to their slaves.
2.  $b_1$  sends a Phase 1 message to slave  $a_1$ , but the message is delayed in the network.
3.  $B$ 's slaves notify  $B$  that they have completed their Phase 1 work.
4.  $A$  and  $B$  perform a barrier, but  $A$  is unresponsive. (It is blocked waiting for  $a_1$  to acknowledge receipt of  $b_1$ 's message.)
5. Because nonblocking barriers enable  $B$  to continue without waiting for  $A$ ,  $B$  multicasts Phase 2 work to its slaves.
6.  $b_2$  sends a Phase 2 message to  $a_1$ , and it arrives before  $b_1$ 's Phase 1 message.

Had traditional barriers been used in the preceding sequence,  $B$  would not have been able to prematurely exit the barrier in step 5.  $B$  therefore would not have multicast Phase 2 work to its slaves, and  $b_2$  would not have been able to send a message to  $a_1$ . All messages would hence be delivered in the same phase in which they were sent.

The reason that nonblocking barriers permit incorrect behavior to occur in this situation is that synchronization among processes in separate slave groups is implicit, not explicit. That is,  $a_1$ ,  $b_1$ , and  $b_2$  never participate in a barrier, yet they still expect to be synchronized with each other. When traditional barriers are used, the slaves are, in fact, synchronized with each other, because they each synchronize with a master (using a reduction), and the masters synchronize with each other (using a barrier). When nonblocking barriers are used, however, logical, not physical, time coordinates message delivery. Because  $a_1$  has no knowledge of the logical time at  $A$ , it has no way of knowing that Invariant 1 will be violated if it delivers  $b_2$ 's message.

A simple workaround is to have all processes—slaves and masters—participate in each barrier. Because processes entering a nonblocking barrier can exit immediately, there is no performance penalty associated with having a larger number of processes involved in a barrier operation. A programmer does need to realize, however, that nonblocking barriers may be inappropriate for certain complex communication patterns.

## 4.5 Alternative implementations

Section 4.2 described one way to implement nonblocking barriers. However, that is certainly not the only way that nonblocking barriers can be implemented. We now present a couple of alternatives. Section 4.5.1 examines the consequences of using a simpler scheme than what was previously presented, and Section 4.5.2 shows what changes would need to be made to implement nonblocking barriers entirely in hardware.

### 4.5.1 Single logical clock

Nonblocking barriers, as presented in Section 4.2, use an array of logical clocks, one for each peer process. It is therefore instructive to ask: Would a single logical clock suffice? That is, can nonblocking barriers be made simpler than what was previously presented?

The single logical clock scheme works as follows. Each process has a single logical clock, which acts as a barrier counter. A process increments its barrier counter each time the process exits a barrier. And messages are timestamped with the value of the counter and are not delivered until the value of the receiver's counter is no less than the value included in the message. The advantage of this scheme over the original nonblocking barrier algorithm is that its resource usage scales better with the number of processes. While Section 4.2 calls for one logical clock per peer, the single logical clock scheme requires only one logical clock in toto. However, in practice, this is not a significant savings. In the original nonblocking barrier scheme, even a fairly large parallel program would require only a few kilobytes of memory—an amount that can easily fit even in most NICs' limited memory space.

The single logical clock scheme works well when all processes in the computation participate in each barrier.<sup>6</sup> However, the drawback of using a single logical clock is that deadlock can ensue if not all of the application's processes are involved in every barrier. The barrier counter at a receiving process may never advance enough for it to receive a particular sender's messages. As an illustration of the problem, consider the sequence of barrier operations shown in Figure 4.9, which was the same sequence used in Figure 4.4 on page 37.

In Figure 4.9, the sets of boxes to the right of each line show each process' barrier counter. The single logical clock scheme uses fewer resources than the original nonblocking-barrier algorithm, and appears correct at first glance. However, consider what would happen if process B were to send a message to process A after the above sequence completes. Because a process' barrier counter is incremented

---

<sup>6</sup>With MPI [73] terminology, one would say that `MPI_Barrier()` is always passed `MPI_COMM_WORLD` as a communicator.

1. Program initializes.

| A | B | C | D |
|---|---|---|---|
| 0 | 0 | 0 | 0 |

2. Processes A and B synchronize.

| A | B | C | D |
|---|---|---|---|
| 1 | 1 | 0 | 0 |

3. Processes C and D synchronize.

| A | B | C | D |
|---|---|---|---|
| 1 | 1 | 1 | 1 |

4. Processes B and C synchronize.

| A | B | C | D |
|---|---|---|---|
| 1 | 2 | 2 | 1 |

Figure 4.9: Bookkeeping with a single logical clock

after each barrier completes, the message is timestamped with “2”, the time on process B’s barrier counter. However, process A will receive the message at time 1 on *its* logical clock. Because  $1 < 2$ , process A will not be able to deliver process B’s message. If there are no more barriers in the program, process A will *never* be able to deliver the message.

It is arguable whether that deadlock situation is a serious impediment to using a single logical clock. If it is known *a priori* that every process in the application is involved in every barrier, and if memory utilization is a serious concern, then the single logical clock scheme is a viable alternative implementation to that described in Section 4.2.

#### 4.5.2 All-hardware implementation

Because nonblocking barriers, as described, preserve collective-communication semantics, and because the bookkeeping is fairly minimal in terms of space and time requirements, these nonblocking barriers could potentially be implemented in hardware. In the context of VIA, for example, the (implementation-specific) per-process state maintained by the VI Provider could be extended to include the appropriate vectors of logical clocks, with one entry per VI. Whenever the NIC transmits a message, it can include the appropriate timestamp in the message header. On the receive side, the NIC can compare the message’s timestamp to the current reading of the logical clock and, only if the message is deliverable, update the receive descriptor’s Status field and—at the application’s request—raise an interrupt.

Figures 4.10–4.12 are a sketch of what the hardware to implement the nonblocking barrier function might look like. They are a fairly faithful, but unoptimized, translation of Algorithms 4.1 and 4.2. Figure 4.10 represents the hardware needed to implement the second stanza of Algorithm 4.1. However, the hardware version can update all the *barriers\_started* and  $TS_{recv}$  counters in parallel, for greater efficiency. Figure 4.11 is essentially the same as Figure 4.10, but it corresponds to the receive side instead of the send side. Specifically, it implements the processing of messages in the final stage of a barrier, and is the hardware equivalent of the second half of the third stanza of Algorithm 4.2. The receive hardware receives from the receive buffer instead of the transmit buffer, and updates *barriers\_started*. Because both the send side and the receive side increment counters in  $TS_{recv}$ , the adders at the bottom of each figure would actually be combined in practice. Finally, Figure 4.12 handles the delivery of point-to-point messages. The top half of the figure corresponds to the second stanza of Algorithm 4.2, and the bottom half of the figure corresponds to the first half of Algorithm 4.2’s third stanza.

While many details, such as error handling and the handling of numeric wraparound, have been omitted, the purpose of Figures 4.10–4.12 is to make the following key points:

1. The bookkeeping needed for nonblocking barriers exhibits a lot of parallelism. Only one clock cycle and a number of gate delays is added to the critical path.
2. Nonblocking barriers require little hardware to implement, the hardware is straightforward to design, and various logic blocks can be reused in multiple locations.

The first point implies that the performance of the common-case, point-to-point communication, will not be degraded noticeably by adding unresponsiveness-tolerant hardware. The second point implies that the hardware shown in Figures 4.10–4.12 is apt to be inexpensive in terms of materials and nonrecurring engineering costs. Hence, the hardware could reasonably be added to an existing hardware VIA implementation. The main hardware cost is the storage for the the receive FIFO and the various counters (*barriers\_started*, *barriers\_finished*, and  $TS_{recv}$ ), whose size is proportional to the number of VIs. To put the number of VIs in perspective, the Gigaset cLAN1000 adapter used in this thesis supports 1024 VIs. A particularly hardware-stingy implementation of nonblocking barriers—or one that supports an immense number of VIs—could save on hardware at the expense of I/O bus crossings by keeping only a small cache of VIs on the NIC, and paging them in and out of main memory over the I/O bus. This sort of caching technique

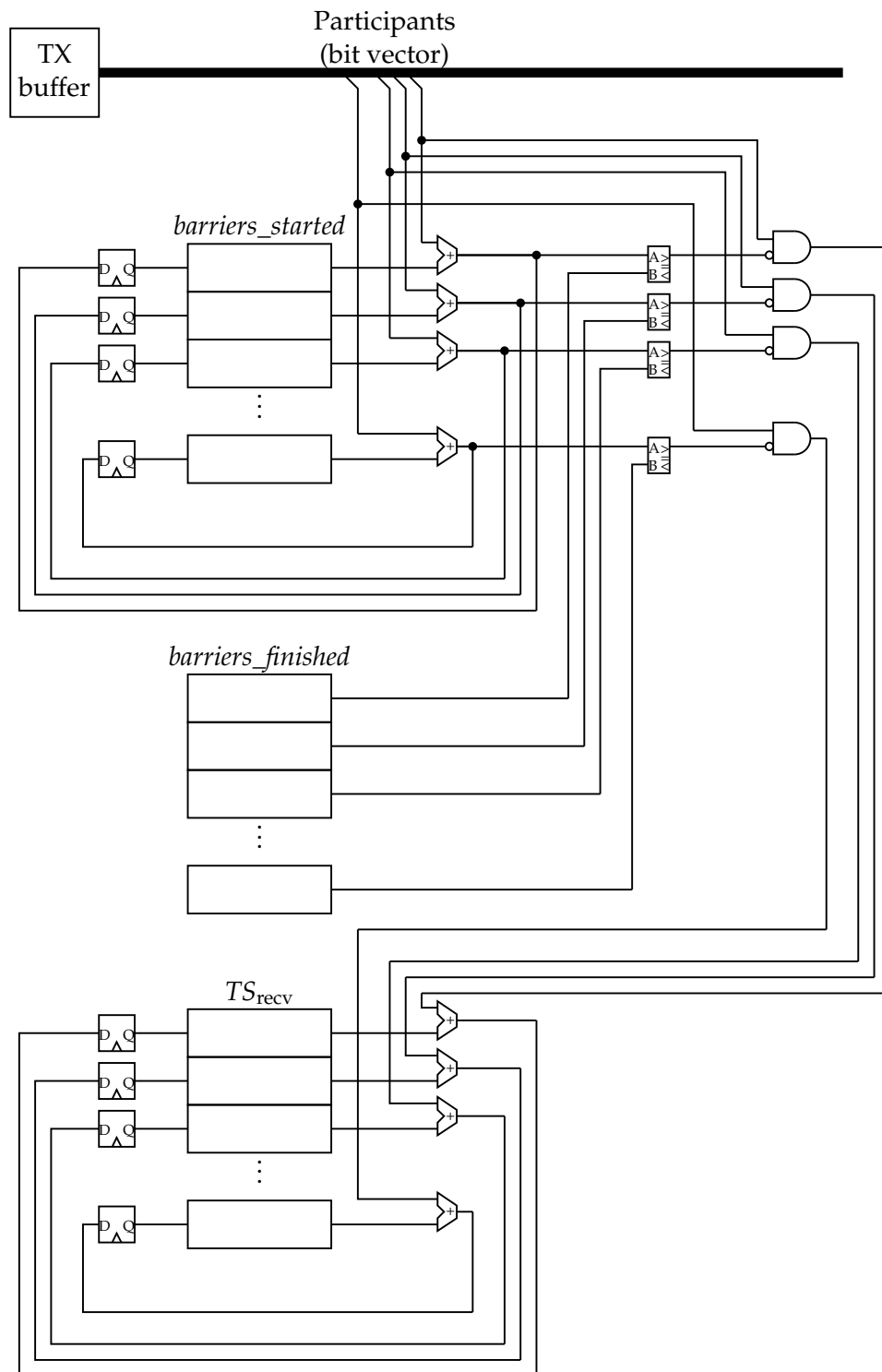


Figure 4.10: Hardware for an unresponsiveness-tolerant barrier (send side)

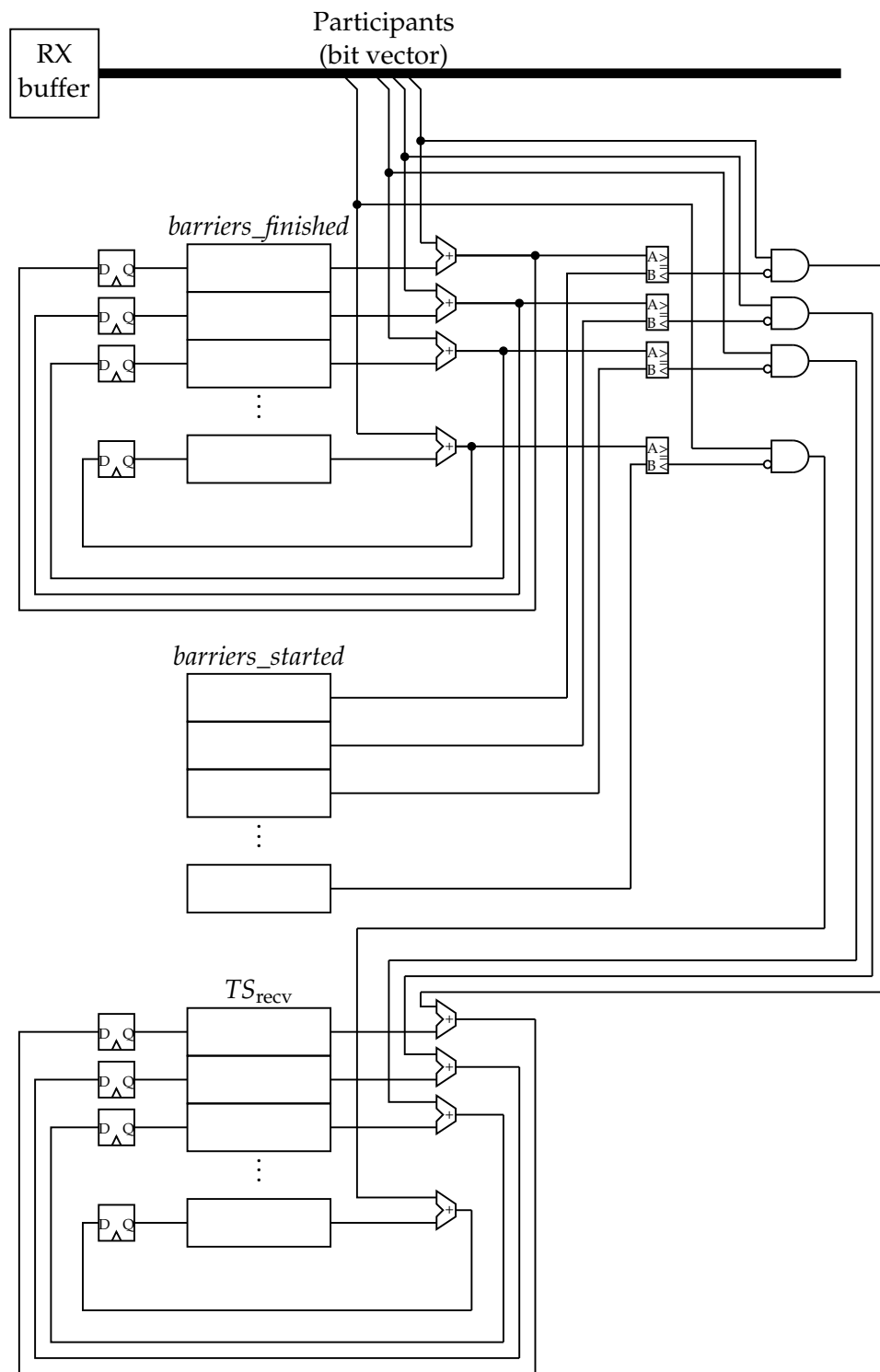


Figure 4.11: Hardware for an unresponsiveness-tolerant barrier (receive side, final stage)

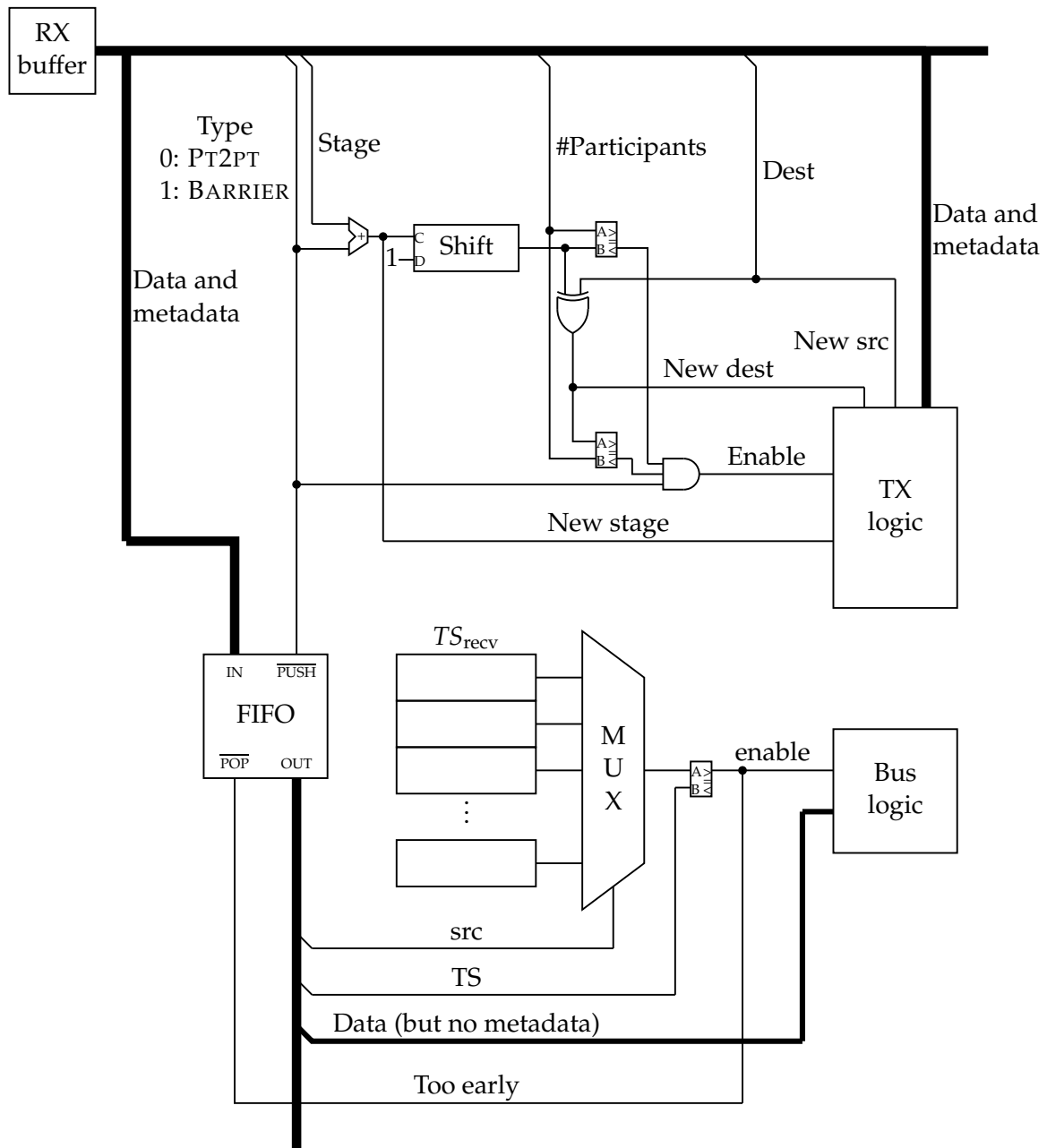


Figure 4.12: Hardware for an unresponsiveness-tolerant barrier (receive side, intermediate stages)

has been implemented previously in network interface firmware by, e.g., AM-II [71]. The advantages of a hardware approach are:

- Better scaling with the number of VIs per NIC
- Better CPU utilization
- Improved unresponsiveness tolerance
- Better buffer utilization

Scaling is improved, because hardware can take advantage of the inherent parallelism in the barrier bookkeeping to reduce bookkeeping from an  $O(n)$  operation to an essentially  $O(1)$  operation. As the hardware shown in Figure 4.12 does not notify the host of message arrival if the message is too early (i.e., arrives before a preceding barrier completes), processes that block on message arrival will not awake from blocking by false positives and will therefore yield better CPU utilization. Hardware-implemented barriers can make the system more tolerant of unresponsiveness, because host involvement is necessary only to initiate the barrier operation. The hardware handles the remainder of the work, even if the the host process subsequently becomes unresponsive. Finally, a hardware implementation of nonblocking barriers reduces some of the VIA buffer resources needed by the host, specifically, message descriptors and registered (i.e., pinned) message buffers. It can do this because barriers require only temporary buffering. Once the hardware has processed an incoming barrier message and either forwarded it onwards or updated the *barriers finished* and  $TS_{recv}$  counters, it can immediately and automatically recycle the message state. In a software implementation of nonblocking barriers, the communication library must provide a message buffer and message descriptor for each potential outstanding barrier, and these resources persist until the process services the network and consumes them.

To support a hardware implementation of nonblocking barriers, the VIA API (VIPL) must be augmented with a barrier function, which will notify the NIC that it needs to perform the appropriate bookkeeping. Figure 4.13 shows what the prototype of such a function might look like. Note that VIA applications that do not use barriers can run unmodified; only those that wish to take advantage of nonblocking collective communication must use the new barrier function.

## 4.6 Alternative techniques

The previous section presented two additional ways to implement nonblocking barriers. In this section, alternatives to nonblocking barriers are discussed. First, the



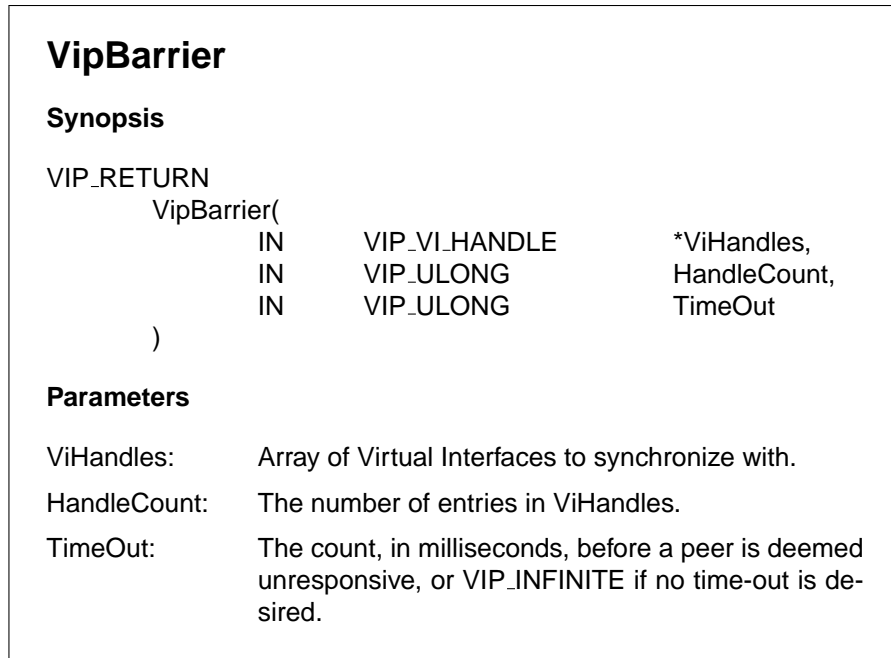


Figure 4.13: Sample prototype of a VIA barrier function

idea of tolerating unresponsiveness *within* each barrier operation is considered (Section 4.6.1). Next, Section 4.6.2 discusses a mechanism for explicitly detecting unresponsiveness. This mechanism can be used to explicitly reschedule communication events, in order to minimize idle time. Finally, in Section 4.6.3, we describe how an OS-centric approach to tolerating unresponsiveness would differ from the communication-centric approach taken by nonblocking barriers. For each approach presented in Sections 4.6.1–4.6.3, we argue that nonblocking barriers are a superior (or, at least, complementary) technique for tolerating unresponsiveness.

#### 4.6.1 IntrabARRIER unresponsiveness tolerance

While nonblocking barriers tolerate unresponsiveness *between* barriers, it is also possible to tolerate unresponsiveness *within* each barrier operation. One way to do this is to have each process that detects an unresponsive peer (e.g., with an expired timeout) notify all subsequent peers of the unresponsiveness. When the unresponsive process becomes responsive again, it multicasts its presence to the rest of the processes in the barrier. Figure 4.14 illustrates the execution of this idea applied to an 8-process barrier, in which process 7 is initially unresponsive. In the first stage, process 6 does not receive a message from process 7 within some specified timeout period, so it concludes that process 7 is unresponsive. In the second stage, process 6

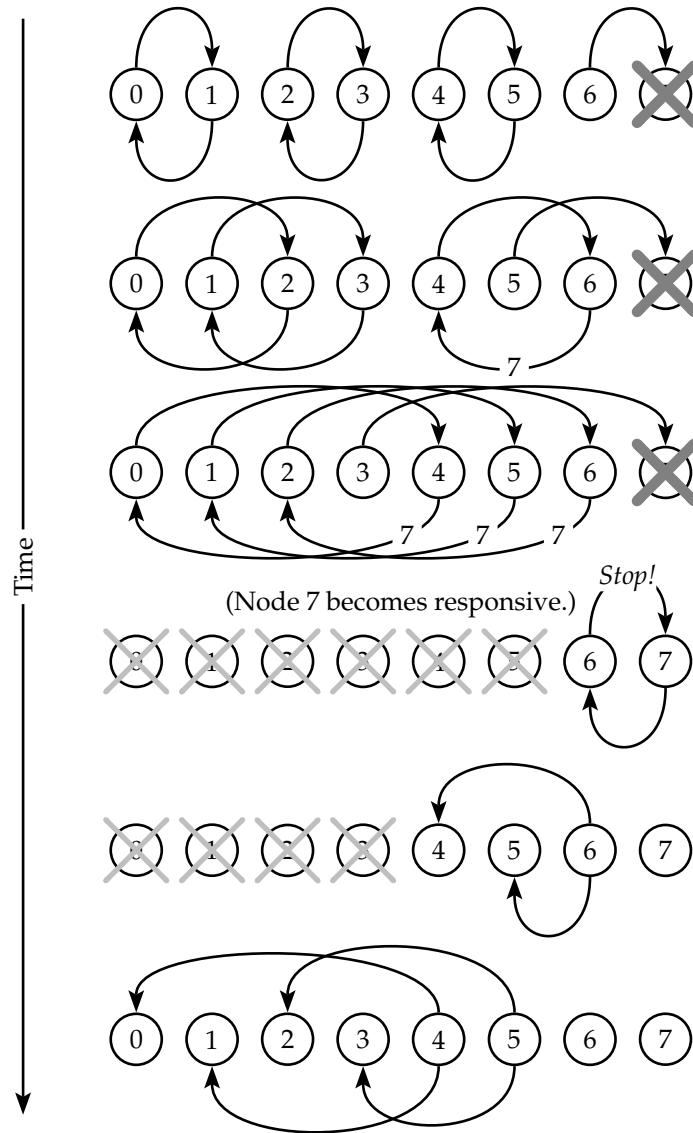


Figure 4.14: A nonblocking barrier

tells its peer, process 4, that the barrier cannot complete until it hears from process 7. Meanwhile, process 5 discovers for itself that process 7 is unresponsive. In the third and final stage of the barrier, processes 4, 5, and 6 tell, respectively, processes 0, 1, and 2 that process 7 is unresponsive, while process 3 discovers process 7's unresponsiveness for itself.

When process 7 becomes responsive again, it initially tries to continue with the barrier, oblivious to its own prior unresponsiveness. Its stage 1 peer, process 6, tells it that all the other processes have finished their participation in the barrier, and hence, process 7 can leave the barrier immediately. Process 6 then multicasts to the

rest of the processes that process 7 is once again responsive and that all participants can complete the barrier.

While this approach does not prevent an unresponsive process from delaying barrier completion, it does ensure that a barrier operation is penalized only once. By effectively converting the barrier into a multicast during otherwise idle time, the barrier becomes more robust to future unresponsiveness. However, there are two limitations to this approach:

1. It has limited applicability.
2. It does not enable processes to make progress while waiting for an unresponsive peer.

Because barriers generally take comparatively less time than the surrounding computation, it is unlikely that many processes will become unresponsive within the small window of time the barrier is executing. And because processes can make no other progress during a long segment of idle time, only minimal unresponsiveness can be tolerated. Due to those limitations, the idea of tolerating unresponsiveness within a barrier was abandoned in favor of tolerating unresponsiveness between barriers, as is done in the nonblocking barrier approach.

#### 4.6.2 Explicit unresponsiveness detection

Another approach to tolerating unresponsiveness is to have processes explicitly detect unresponsiveness and reschedule communication around it. That is, within a collective-communication operation, each process has a number of peers it needs to communicate with. If a process starts communicating with the peers that are responsive at the time, the hope is that the unresponsive peers will once again become responsive by the time their turn to communicate comes around.

One way to implement explicit unresponsiveness detection is with timeouts in the point-to-point communication protocol. Many messaging layers are implemented using a three-way handshake to transmit data (Figure 4.15). In the first step of a three-way handshake, the sender notifies the receiver of its intent to send a message. In the second step, the receiver allocates local buffer space and alerts the sender when it is ready. And in the third step, the sender transmits the data. A three-way handshake is a useful tool, because it is an easy way to implement flow control; the sender cannot transmit a message until the receiver has space to store it. But a three-way handshake can additionally be used to detect unresponsiveness. If a sender does not receive the receiver's acknowledgement within some timeout period (i.e.,  $2(L + 2o)$  plus a small amount of compute time in the LogP model [28]), it can conclude that the receiver is probably unresponsive. The sender can then move

on to another receiver that it needs to communicate with, and later return to the unresponsive process (which, one hopes, is no longer unresponsive). This explicit unresponsiveness detection mechanism can be used for intrabARRIER unresponsiveness tolerance (Section 4.6.1).

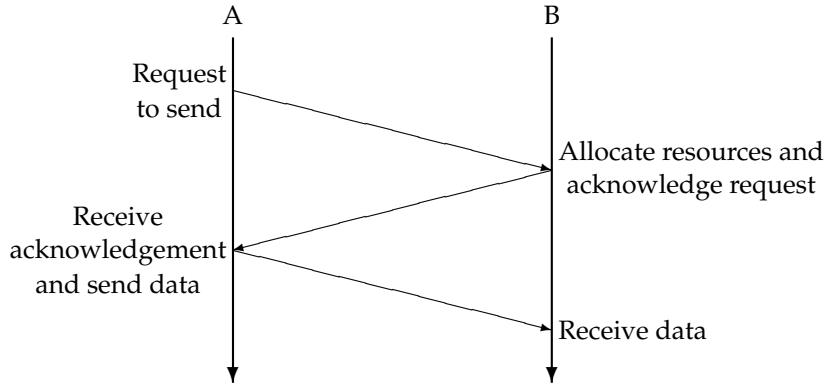


Figure 4.15: A three-way handshake

The primary disadvantage of explicit unresponsiveness detection is that performance is sensitive to the timeout value. If the timeout value is set too large, then an unresponsive receive will introduce excessive idle time to the sender, thereby decreasing the application’s efficiency. If, on the other hand, the timeout value is set too small, then receivers will frequently appear unresponsive when they are, in fact, responsive. These false positives will add extra overhead to the sender, and may cause a livelock situation, if the sender cycles among the possible receivers indefinitely in search of a responsive one.

### 4.6.3 Operating system support

Rather than tolerate unresponsiveness at the communications level, an alternative is to enlist the operating system in unresponsiveness tolerance. This generally involves rescheduling kernel-level threads to hide unresponsiveness behind useful work. For example, the operating system can support scheduler activations [3], which provide callbacks to applications, granting to their (fast) user-level threads many of the benefits of (slow) kernel threads, such as preemption and the ability to block on I/O. A parallel application could use these facilities to give higher priority to threads involved in a collective-communication operation, but to schedule additional worker threads that make progress while the threads performing collective-communication operations are blocked. The benefit of this approach is that it provides applications with considerable flexibility to tolerate unresponsiveness. The problem is that it violates the properties stated on page 21: it requires

not only application modifications, but also operating system modifications.<sup>7</sup> These modifications hamper acceptance of an unresponsiveness-tolerating technique that is centered around something like scheduler activations. Programmers may be unwilling or unable to convert their applications to a specially multithreaded version. And cluster administrators may be unwilling to run a specialized operating system, because specialized operating systems are unable to keep pace with their COTS counterparts.

A second approach that relies on operating system support is to use coordinated thread scheduling [82] across nodes to ensure that all threads involved in a collective-communication operation are granted the CPU coincidentally. The goal in this approach is to prevent unresponsiveness, rather than tolerate it. The advantage of using coordinated thread scheduling to combat unresponsiveness is that, like my thesis work, it does not require application modifications. In addition, some forms of coordinated scheduling, such as dynamic coscheduling [96], can be implemented without OS modifications [95]; rather, they use device drivers and NIC firmware to *influence* the operating system's thread scheduler. The disadvantage of using coordinated scheduling techniques to make collective communication responsive is that coscheduling threads does not necessarily imply that collective communication is responsive. There may be load imbalance *within* the threads that coordinated scheduling can do nothing about. Nevertheless, coordinated thread scheduling can complement nonblocking barriers. The coordinated thread scheduler can prevent some unresponsiveness by keeping all of an application's threads running in parallel,<sup>8</sup> while nonblocking barriers can tolerate whatever unresponsiveness remains.

## 4.7 Discussion

This chapter presented nonblocking barriers, a novel approach to tolerating endpoint unresponsiveness in PC clusters. Section 4.2 provided a detailed algorithm for implementing nonblocking barriers. While the algorithm represents a natural expression of an all-software implementation, Section 4.5.2 outlined how it can be adapted to produce an all-hardware implementation. Nonblocking barriers are a sufficiently flexible mechanism that they can be implemented at either extreme of the hardware-software spectrum. Table 4.3 summarizes some of the tradeoffs involved in implementing nonblocking barriers in hardware versus firmware versus software. In the context of tolerating unresponsiveness, a real strength of a hardware or firmware implementation is responsiveness, while software is prone to un-

---

<sup>7</sup>No current commodity operating system supports scheduler activations.

<sup>8</sup>In emergent coscheduling schemes, such as dynamic coscheduling [96] and implicit coscheduling [5], it is possible that some threads will occasionally not be coscheduled with the rest.

responsiveness, due to operating system and memory hierarchy behavior. Software and firmware are more flexible than hardware in terms of the ability to alter the implementation; if an implementation of nonblocking barriers needs to support larger clusters, it would be much quicker to change the data structures used by software/firmware than it would to fabricate new hardware. Software has access to the host CPU and system memory, which are assumed to be faster and larger, respectively, than anything available on a device (Section 3.1). Hardware can achieve better performance by exploiting the parallelism in the nonblocking-barrier algorithm, while software is limited to instruction-level parallelism or the sacrifice of additional CPUs. Firmware could conceivably exploit parallelism, but no current, commodity NIC contains firmware that is structured in a manner favorable to nonblocking barriers. Finally, the additional performance achievable by hardware is offset by its higher cost relative to firmware and software.

Table 4.3: Hardware/firmware/software implementation tradeoffs

| Attribute                        | Hardware | Firmware | Software |
|----------------------------------|----------|----------|----------|
| Responsive                       | ✓        | ✓        | ✗        |
| Flexible                         | ✗        | ✓        | ✓        |
| Fast processor/large memory      | ✗        | ✗        | ✓        |
| Efficiently exploits parallelism | ✓        | ?        | ✗        |
| Low cost                         | ✗        | ✓        | ✓        |

It is worthwhile to consider an implementation of nonblocking barriers that is implemented as a combination of hardware, software, and firmware. The way to think about a combined approach is in terms of the various responsibilities assigned to each of hardware and software. The hardware’s responsibility should be to keep the software from being interrupted by premature message arrivals, i.e., messages sent after a barrier, but received before it. In the all-software scheme, every time a message arrives, the software must process it, determining if it is deliverable, and buffering it if it is not. The software’s responsibilities in a combined hardware/software scheme are to distinguish communication operations as being collective and to handle establishment and tear-down of peer groups. In addition, software can be used to cache hardware state if the NIC’s memory is insufficient to hold all of it itself. By assigning the software the management responsibilities and the hardware the responsibility for most of nonblocking-barrier work, further enhanced performance may be realizable.

A second discussion point is that while this dissertation has so far focused on collective communication itself, collective-communication operations are implemented in terms of point-to-point messages, and there are design decisions involving those.

The most crucial design decision concerns message-notification semantics. Two common ways to detect message arrival are *blocking* and *polling*.<sup>9</sup> With polling notification, a thread checks periodically and explicitly for message arrival. With blocking notification, a thread sleeps until a message arrives, allowing other threads access to the CPU in the meantime. The tradeoff, in the context of this dissertation, is that blocking increases the time that other threads are responsive, while polling decreases the time that the current thread is unresponsive. It is important to investigate the effects of both in any dissertation that investigates unresponsiveness, as neither notification mechanism is apt to be better in all circumstances. Many of the graphs in Chapter 5 present measurement data taken both when polling and blocking are used for the underlying point-to-point communication.

For every problem, there is one  
solution which is simple, neat, and  
wrong.

*H. L. Mencken*

---

<sup>9</sup>This leaves aside techniques such as asynchronous message handlers and combined blocking/polling schemes such as the Polling Watchdog [72].

# 5 Experiments

Chapter 4 introduced nonblocking barriers, a new mechanism designed to tolerate unresponsiveness. In Chapter 5, we evaluate the performance gained by applying nonblocking barriers. The goal is to prove the thesis statement made in Chapter 3, namely that utilizing nonblocking barriers to tolerate unresponsiveness at communication endpoints will result in a noticeable improvement in application performance.

The rest of Chapter 5 is structured as follows. Section 5.1 describes the experimental setup used for the experiments in this chapter. Section 5.2 presents results that demonstrate that unresponsiveness is indeed a problem for collective-communication performance. The core performance results are presented in Section 5.3, and, in Section 5.4, these results are shown to be compatible with other unresponsiveness-tolerating techniques and robust to cluster scale.

## 5.1 Experimental setup

Except where noted, all of the experiments described in this chapter were performed on a cluster with the characteristics shown in Table 5.1.

Table 5.1: Platform characteristics

| Component     | Characteristic                                 |
|---------------|--|
| Node          |  |
| Type          | Hewlett-Packard NetServer LPr                  |
| CPU           | Dual 450 MHz Pentium IIs                       |
| Memory        | 1 GB SDRAM                                     |
| OS            | Windows NT 4.0, Terminal Server Edition        |
| Network       |  |
| Type          | Giganet cLAN (VIA)                             |
| Size          | 32 nodes                                       |
| Topology      | Multistage                                     |
| Communication | Custom user-level messaging layer (VIA++ [84]) |



Table 5.2: Benchmarks used in Chapter 5

| Benchmark          | Description                                     |
|--------------------|---|
| <i>barrier</i>     | Perform a large number of back-to-back barriers |
| <i>mg</i>          | 3-D multigrid solver                            |
| <i>cholesky</i>    | Cholesky factorization                          |
| <i>prefix scan</i> | Prefix-scan of a large vector                   |
| <i>radix sort</i>  | Sort the elements in an array                   |

A number of graphs shown in this chapter contain error bars. Unless otherwise indicated, these error bars correspond to plus-or-minus one standard deviation in performance over 11 trials.

### 5.1.1 Applications

The experiments discussed in this chapter draw from a pool of five microbenchmarks and application kernels (Table 5.2): *barrier*, *mg*, *cholesky*, *prefix scan*, and *radix sort*.

**barrier** The *barrier* microbenchmark (Procedure 5.1) models a program with alternating computation and synchronization phases. For each of 10,000 iterations, it performs (and times) a barrier operation and then idles in an empty loop to consume CPU time and thereby simulate computation.

---

#### Procedure 5.1 Barrier microbenchmark

---

```

for all nodes (in parallel) do
  repeat 10,000 do
     $t_0 \leftarrow$  current time
    Barrier with all the other nodes (using a  $\log N$  time algorithm).
     $t_1 \leftarrow$  current time
    Write  $(t_1 - t_0)$  to a memory-mapped file
    Spin for a given number of iterations (while touching an external variable to
    prevent dead code elimination)
  end repeat
end for

```

---

**mg** The *mg* program is a 3-D multigrid solver that is one of the NAS Parallel Benchmarks (NPB) [6]. While the original NPB version uses MPI for communication, I used a version of the benchmark that Oxford Parallel ported to BSPlib to showcase BSP as a competitive alternative to MPI and similar messaging layers [65]. I instrumented BSPlib so that process 0 records the time spent in the computation

component of each superstep. More precisely, I modified the BSPLib source code to record the time at the beginning and ending of the `bsp_sync()` function (i.e., the ending and beginning, respectively, of a local computation).

**cholesky** *cholesky* is a Cholesky factorization code from the [Center for Simulation of Advanced Rockets](#). *cholesky* is written in C with the MPI communication interface [73] and is configured to factor a  $10,000 \times 10,000$  matrix. The program has the communication structure shown in Figure 5.1.

```

Initialization:
  MPI_Barrier()

Cholesky factorization:
  for k in 1, 2, ..., 10,000 do
    MPI_Reduce((10,000 - k)-element vector)

Forward substitution:
  repeat 10,000 do
    MPI_Reduce(1-element vector)

Backward substitution:
  repeat 9,999 do
    MPI_Bcast(1-element vector)

```

Figure 5.1: Communication structure of the Cholesky code

The vast majority of *cholesky*'s time is spent in the factorization phase. Table 5.3 shows a somewhat arbitrary selection of data points, intended to provide a feel for the ratios in execution time across *cholesky*'s three phases. In the table, "naive" means a flat (two-level) communication tree is used for the collective-communication operations. "Binary" means a binary tree is used.

**prefix scan** A prefix scan operation<sup>1</sup>, implemented in the *prefix scan* program, takes a vector  $x$  and an associative operator,  $\odot$ , and maps from  $\{x_0, x_1, x_2, \dots, x_{N-1}\}$  to  $\{x_0, x_0 \odot x_1, x_0 \odot x_1 \odot x_2, \dots, x_0 \odot x_1 \odot x_2 \odot \dots \odot x_{N-1}\}$ . With  $O(N)$  processes, *prefix scan* can be executed in  $O(\lg N)$  time [81].

The data-parallel expression of *prefix scan* is fairly straightforward (Algorithm 5.2). Even a naive translation of this to node code for a workstation cluster using message-passing communication is noticeably more involved (Algorithm 5.3). Algorithm 5.3 is, in fact, simplified from what I actually implemented. An important omission is flow control, which is necessary because there is no flow control in

<sup>1</sup>"Prefix scan" also goes by a number of other names, including "parallel prefix" and—when addition is the operator used—"sum prefix" and "plus scan."

Table 5.3: Selected *cholesky* performance numbers

| Experiment   | Time spent in each phase (H:MM:SS) |                      |                       |
|--|------------------------------------|----------------------|-----------------------|
|  | Factorization                      | Forward substitution | Backward substitution |
| 16 processes, 16 nodes, 1 CPU/node, naive reductions, binary multicasts, blocking notification                                     | 0:07:23                            | 0:00:01              | 0:00:01               |
| 4 processes, 1 node, 2 CPUs/node, naive reductions, binary multicasts, polling notification  | 1:25:36                            | 0:03:33              | 0:03:43               |
| 8 processes, 8 nodes, 1 CPU/node, binary reductions, binary multicasts, polling notification, competition for the CPU on all nodes | 0:31:18                            | 0:01:37              | 0:02:06               |

VIA, and without flow control, the large number of messages ( $O(N \lg N)$ ) would overflow the message buffers, causing dropped connections and message loss. My implementation of Algorithm 5.3 uses the same static window-based flow control scheme that Fast Messages [85] uses. The added synchronization due to flow control will be shown in Section 5.3 to have a deleterious effect on the ability of nonblocking barriers to tolerate unresponsiveness.

The next implementation of *prefix scan* I implemented is more tuned for workstation clusters and message passing. Algorithm 5.4 describes this optimized implementation. The key optimization is that only the last element in each process' local subset of the data array needs to be sent, and only the first element needs to be received. Table 5.4 lists the variables used in Algorithms 5.2–5.4 and their assigned meanings.

Table 5.4: Variables used in Algorithms 5.2–5.4

| Variable | Description  |
|----------|--|
| $N$      | Total number of elements to prefix-scan                      |
| $P$      | Number of processes involved in the computation              |
| $n$      | Number of elements on each processor                         |
| $p$      | Process ID of "self"   |
| $x$      | Local slice of the global array (indexed from 0 to $n - 1$ ) |
| $m$      | Message received from the network                            |
| $i, j$   | Loop variables   |

---

**Algorithm 5.2** Prefix-scan (data parallel)

---

```
for  $j \leftarrow 0$  to  $\lceil \lg N \rceil - 1$  do
  for  $i \leftarrow 0$  to  $N - 1$  parallel do
    if  $i \geq 2^j$  then
       $x[i] \leftarrow x[i - 2^{j-1}] \odot x[i]$ 
    end if
  end for
end for
```

---

The purpose of experimenting with two different implementations of prefix scan is as follows. By examining the performance of the naive implementation, we can determine if a program originally targeted for a SIMD [39] machine (data-parallel, tightly coupled, no unresponsiveness) can be naively retargeted to a workstation cluster (message-passing, loosely coupled, much unresponsiveness) without suffering from the violated assumption of no unresponsiveness. That is, do nonblocking barriers provide a sufficient illusion of responsiveness to satisfy programs that rely on complete responsiveness? In contrast, by examining the performance of the cluster-optimized implementation of prefix scan, we can determine if program designed with clusters in mind—and therefore much less tightly coupled—can benefit from unresponsiveness tolerance.

**radix sort** *radix sort* is a data-parallel-style radix sort routine, based on the one described by Hillis and Steele [47] and implemented on the Connection Machine [46]. Algorithm 5.5 shows a data-parallel radix sort of this type. The `COUNT()` function tallies the number of “active” processes—data-parallel terminology for processes whose `if` test took the `TRUE` branch. `COUNT()` is implemented with a reduction operation, using “+” as the operator. The `ENUMERATE()` function returns “1” to the first active process, “0” to the second, “3” to the third, and so forth. It is implemented as follows: Each active process, sets its  $y[k]$  to 1. Each inactive process sets its  $y[k]$  to 0. Then, all processes, both active and inactive, do a prefix scan with “+” as the operator.

Some other things to note about Algorithm 5.5 are that all processes see the value of  $c$  after the first `end if`, even though only those with bit  $j$  of  $x[k]$  equal to 0 contribute to its value. This is why two `if` statements are used, instead of a single `if...else`. Also, the final assignment is an all-to-all exchange, another collective-communication operation. Each process receives as many messages as it sends, but this number is not known at compile time.

Algorithm 5.6 is a high-level summary of the node code used in the message-passing version of *radix sort*. The key observation to make from Algorithm 5.6 is that radix sort employs a wealth of collective-communication operations: pre-

---

**Algorithm 5.3** Prefix-scan (naive)

---

```
for  $i \leftarrow n - 1$  downto 0 do
   $x'[i] \leftarrow \text{INVALID}$ 
end for
for  $j \leftarrow 0$  to  $\lceil \lg N \rceil - 1$  do
  for  $i \leftarrow n - 1$  downto 0 do
     $i' \leftarrow np + i + 2^j$ 
     $p' \leftarrow \lfloor i'/n \rfloor$ 
    if  $p' = p$  then
       $\triangleright$  If one message goes to self, the rest will, too.
      break out of inner loop
    else
      if  $p' < P$  then
        Send {value =  $x[i]$ , offset =  $i'$ } to  $p'$ 
      end if
    end if
  end for

  for  $i \leftarrow n - 1$  downto 0 do
     $i' \leftarrow i - 2^j$ 
     $p' \leftarrow \lfloor (np + i')/n \rfloor$ 
    if  $p = p'$  then
       $x[i] \leftarrow x[i'] \odot x[i]$ 
    end if
  end for

  for  $i \leftarrow n - 1$  downto 0 do
     $i' \leftarrow i - 2^j$ 
     $p' \leftarrow \lfloor (np + i')/n \rfloor$ 
    if  $p' < p$  then
      while  $x'[i] = \text{INVALID}$  do
        Receive  $m$ 
         $x'[m.value] \leftarrow m.offset$ 
      end while
       $x[i] \leftarrow x'[i] \odot x[i]$ 
       $x'[i] \leftarrow \text{INVALID}$ 
    end if
  end for
  BARRIER()
end for
```

---

---

**Algorithm 5.4** Prefix-scan (optimized for clusters)

---

```
 $n \leftarrow \lfloor N/P \rfloor$ 
for  $j \leftarrow 1$  to  $n - 1$  do
   $x[j] \leftarrow x[j - 1] + x[j]$ 
end for

if  $p < N - 1$  then
  Send  $x[n - 1]$  to process  $p + 1$ 
end if
BARRIER()
for  $j \leftarrow 1$  to  $\lceil \lg N \rceil - 1$  do
  if  $p \geq 2^j$  then
    Receive  $m$ 
     $x[n - 1] \leftarrow m \odot x[n - 1]$ 
    if  $p < N - 2^j$  then
      Send  $x[n - 1]$  to process  $p + 2^j$ 
      BARRIER()
      for  $i \leftarrow n - 2$  downto  $0$  do
         $x[i] \leftarrow m \odot x[i]$ 
      end for
    end if
  end if
end for
if  $p \geq 2^{\lceil \lg N \rceil - 1}$  then
  Receive  $m$ 
  for  $i \leftarrow n - 1$  downto  $0$  do
     $x[i] \leftarrow m \odot x[i]$ 
  end for
end if
```

---

---

**Algorithm 5.5** Radix sort (data parallel)

---

```
for  $j \leftarrow 0$  to  $\lceil \lg \text{MAXINT} \rceil - 1$  do
  for all  $k \in [0, N - 1]$  parallel do
    if bit  $j$  of  $x[k]$  is  $0$  then
       $y[k] \leftarrow \text{ENUMERATE}() - 1$ 
       $c \leftarrow \text{COUNT}()$ 
    end if
    if bit  $j$  of  $x[k]$  is  $1$  then
       $y[k] \leftarrow \text{ENUMERATE}() + c - 1$ 
    end if
     $x[y[k]] \leftarrow x[k]$ 
  end for
end for
```

---

fix scan, reduction, multicast, and barrier, as well as an (implicit) all-to-all exchange. Figure 5.2 illustrates the communication dependencies in each iteration of Algorithm 5.6's main loop. Once the *enum\_even*[] and *enum\_odd*[] arrays and the *count\_even* variable are initialized, the two prefix scans and the reduction can proceed. Process 0 must wait for the reduction to complete before it can multicast the results to the rest of the processes. Once a process has a valid version of the prefix-scanned *enum\_even*[] and *enum\_odd*[] and of the global sum of *count\_even*, it can create and sort its *exchangeinfo*[] array and perform its local exchanges. The *P* reductions, in which each process learns how many exchanges it will be told to perform, can all be performed concurrently. The all-to-all exchange requires the result of the exchange. And a barrier is performed at the end of each iteration to ensure message consistency.

*barrier*, *mg*, and *cholesky* are used in Section 5.2 to determine the extent of the problem of unresponsiveness. These benchmarks are suited for that task because they are easy to tune (*barrier*), exhibit realistic ranges of interbarrier work (*mg*), and exhibit linear speedup (*cholesky*). *prefix scan* and *radix sort* are used in Sections 5.3 and 5.4 to examine the performance improvement garnered by nonblocking barriers, because these benchmarks have a suitable structure for nonblocking barriers to exploit, as was presented in Section 4.4.3. *prefix scan* is of interest because it is internally load-imbalanced, while *radix sort* is of interest because it uses a variety of collective-communication operations, many of which can run concurrently.

When the benchmarks run, all files—the executable and, when applicable, measurement logs—reside on the local hard disk. This eliminates performance artifacts that would otherwise be caused by paging code to and from a remote server and the concomitant interaction with the local operating system.

### 5.1.2 Workloads

Table 5.5 lists the independent variables used throughout Chapter 5. The number of nodes and processes range from 1 to 16. If the number of processes is larger than the number of nodes,<sup>2</sup> we say that there is *internal contention* in the system. That is, the processes of an application compete with each other for the CPU. It is also possible to have external contention in the system. *External contention* occurs when processes from other applications share a CPU with the application in question. In the experiments presented in this chapter, external contention is introduced in the form of simple “`while(1);`” spin loops. A process executing a spin loop may run on none of the nodes, one of the nodes, or all of the nodes. Collective-communication operations are composed of point-to-point primitives. A process can be notified

---

<sup>2</sup>The number of nodes is never larger than the number of processes (i.e., there are no idle nodes).

---

**Algorithm 5.6** Radix sort (message-passing)

---

```
for  $j \leftarrow 0$  to  $\lfloor \lg \text{MAXINT} \rfloor - 1$  do
  for  $i \leftarrow 0$  to  $n - 1$  do
     $\text{enum\_odd}[i] \leftarrow$  if  $x[i] \wedge 2^j \neq 0$  then 1 else 0 end if
     $\text{enum\_even}[i] \leftarrow 1 - \text{enum\_odd}$ 
  end for
   $\text{count\_even} = \sum \text{enum\_even}$ 
  PREFIX-SCAN( $\text{enum\_even}, +$ )
  PREFIX-SCAN( $\text{enum\_odd}, +$ )
  REDUCE( $\text{count\_even}, 0, +$ )
  MULTICAST( $\text{count\_even}, 0$ )
  for  $i \leftarrow 0$  to  $n - 1$  do
    Determine which process and global array index  $x[i]$  should go to. Store this
    information in  $\text{exchangeinfo}[]$ .
  end for

  Counting-sort  $\text{exchangeinfo}$  by target process. As a side effect, set each element
  of  $\text{tally}[]$  to the number of exchanges destined for the corresponding process.

  Perform all our local exchanges, copying from  $x[]$  to  $x'[]$ .

  ▷ Tell each process how many messages to expect.
  for  $i \leftarrow 0$  to  $P - 1$  do
     $\text{expected} \leftarrow$  REDUCE( $\text{tally}[i], i, +$ )
     $\text{received}[i] \leftarrow 0$ 
  end for

  while ( $\exists p'$  such that  $\text{tally}[p'] \neq 0$ ) or ( $\text{received} < \text{expected}$ ) do
    if  $\exists p'$  such that  $\text{tally}[p'] \neq 0$  then
      Send as many exchanges to process  $p'$  in a single message as the NIC al-
      lows. Decrement  $\text{tally}[p']$  by that number.
    end if
    if  $\text{received} < \text{expected}$  then
       $m \leftarrow$  RECEIVE()
      Store the data from  $m$  at the appropriate offsets into  $x'[]$ . Increment  $\text{received}$ 
      by the number of exchanges performed.
    end if
  end while
   $x[] \leftarrow x'[]$ 
  BARRIER()
end for
```

---



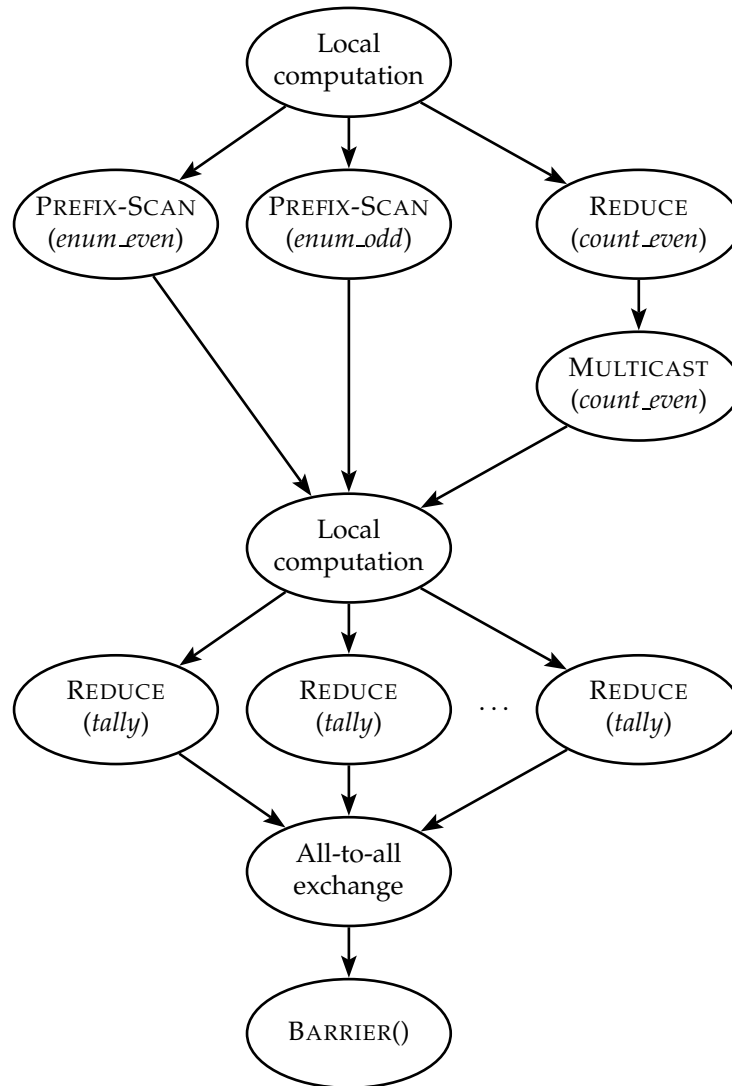


Figure 5.2: Data dependencies in *radix sort*

Table 5.5: Independent variables

| Variable                  | Values utilized  |
|---------------------------|--|
| Number of nodes           | 1, 2, 4, 8, 16   |
| Number of processes       | 1, 2, 4, 8, 16   |
| External competition      | none<br>spin loop on all nodes<br>spin loop only on node 1 |
| Message waiting mechanism | blocking or polling  |
| Multicast algorithm       | binary or flat   |
| Reduction algorithm       | binary or flat   |

of a completed send or receive by either blocking—relinquishing the CPU until a message is sent/received—or polling—repeatedly querying the network for completion. Finally, multicasts and reductions perform their composite point-to-point operations in either a binary- or flat-tree topology. Figure 5.3 shows the multicast case; reductions are analogous, but with the data flowing in the opposite direction.

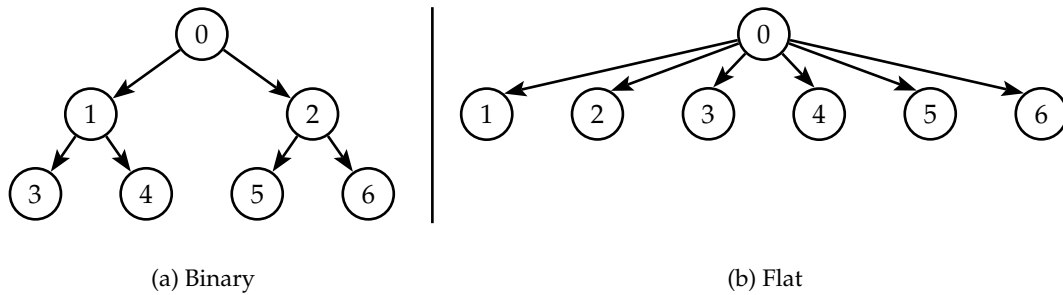


Figure 5.3: Binary versus flat trees

## 5.2 Preliminary experiments

The experiments in this section provide some background on the nature and extent of unresponsiveness in PC clusters. Section 5.2.1 presents experimental data confirming that unresponsiveness is a problem for performance. Section 5.2.2 demonstrates that the primary sources of unresponsiveness are context switches and internal and external CPU contention. Section 5.2.3 summarizes the findings of these preliminary experiments.

### 5.2.1 Total unresponsiveness

The first experiment I performed was intended to determine how much unresponsiveness occurs “naturally” in a workstation cluster. That is, how much unresponsiveness is present on an unloaded system running a single user job with no explicit competition for resources? (The operating system and system services are the only potential source of competition.) To answer that question, I used the *barrier* program, shown in Procedure 5.1 on page 61. Procedure 5.1 models a program with alternating computation and synchronization phases. For each of 10,000 iterations, it performs (and times) a barrier operation and then idles in an empty loop to consume CPU time and thereby simulate computation. The system is perfectly load-balanced; all nodes do the same amount of “work” between barriers. Also, there is no steady background load on the machines, only the normally-running system services.

Ideally, the barrier time should be independent of the spin time. Figure 5.4 clearly shows that this is not the case. In Figure 5.4, each bar represents a different amount of “computation” (i.e., spin) time used in Procedure 5.1, from 0  $\mu$ sec (i.e., back-to-back barriers) up to 10,000  $\mu$ sec (10 msec).<sup>3</sup> The height of each bar represents the total amount of time spent performing barriers. Each bar is partitioned into the contribution to the total by barriers of 10–99  $\mu$ sec, 100–999  $\mu$ sec, . . . , 1,000,000–9,999,999  $\mu$ sec and represents the mean of three runs. Observations in the 100,000  $\mu$ sec-and-up range are statistical noise and can be ignored. The experiment was performed on a four-node cluster, and the mode barrier time was in the 20–29  $\mu$ sec range. The key observation in Figure 5.4 is that barrier time increases with computation time. Section 5.2.2 investigates the source of this performance loss.

Having observed the somewhat counter-intuitive result that barrier time is not independent of inter-barrier time, a logical next step is to formulate a mathematical model of the relation between barrier time and inter-barrier time. There are two reasons that a mathematical model is useful:

1. It simplifies reasoning about unresponsiveness.
2. It makes it possible to estimate the performance impact of unresponsiveness on arbitrarily large clusters.

Before modeling slow barriers, we first need a definition of “slow.” I define a “slow”  $P$ -process barrier as one that takes longer than  $15 \lg P$   $\mu$ sec to complete. This corresponds to the mean barrier time in the no-computation case plus one standard

---

<sup>3</sup>As implied by Procedure 5.1, these timings were actually expressed in an empirically-determined number of spins.

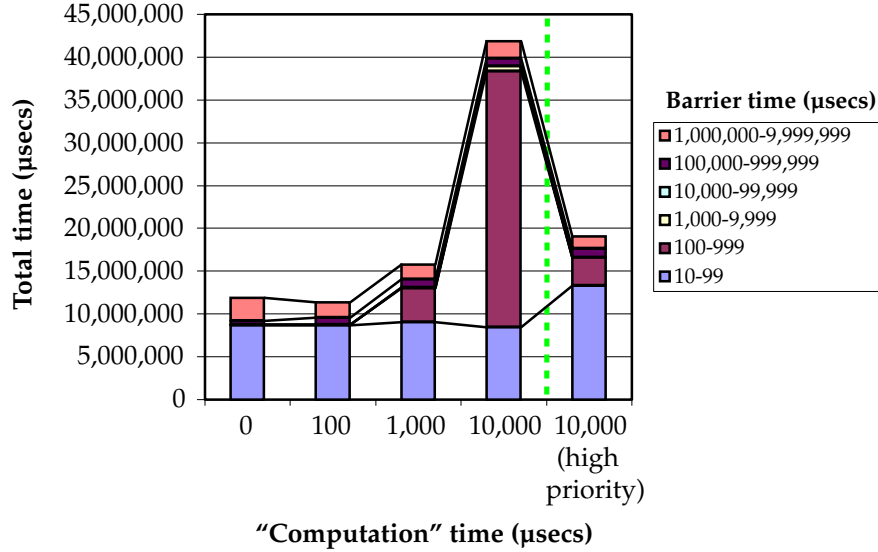


Figure 5.4: Time spent in barriers as a function of “computation” time

deviation. Clearly, when using Procedure 5.1, the total fraction of slow barrier measurements can be at most  $\frac{P-1}{P}$ , because the last process to enter a barrier will observe no delay time.<sup>4</sup>

Figure 5.5 plots the fraction of slow barriers as a function of the computation time. To model this function—and thereby predict the performance for larger numbers of processes—we first make the following observations:

- The fraction of slow barriers is a function of the computation time.  $\implies$  We let  $d$  represent computation time, measured in spins (for precision).
- The curves in Figure 5.5 are S-shaped.  $\implies$  We propose the basic shape of the curve is proportional to  $\frac{d}{d+1}$ .
- The function will reach an asymptote at  $\frac{P-1}{P}$  (all processes except the tardiest observe a slow barrier on every iteration)  $\implies$  The  $d$  in the denominator of  $\frac{d}{d+1}$  should be divided by  $\frac{P-1}{P}$ .

Hence, we arrive at the following model:

$$s(d, P) \approx \frac{d}{\frac{d}{(P-1)/P} + \frac{1}{P}} \quad (5.1)$$

<sup>4</sup>Note that in Procedure 5.1, there are  $P$  measurements per barrier operation, because each process independently records its observed barrier latencies.

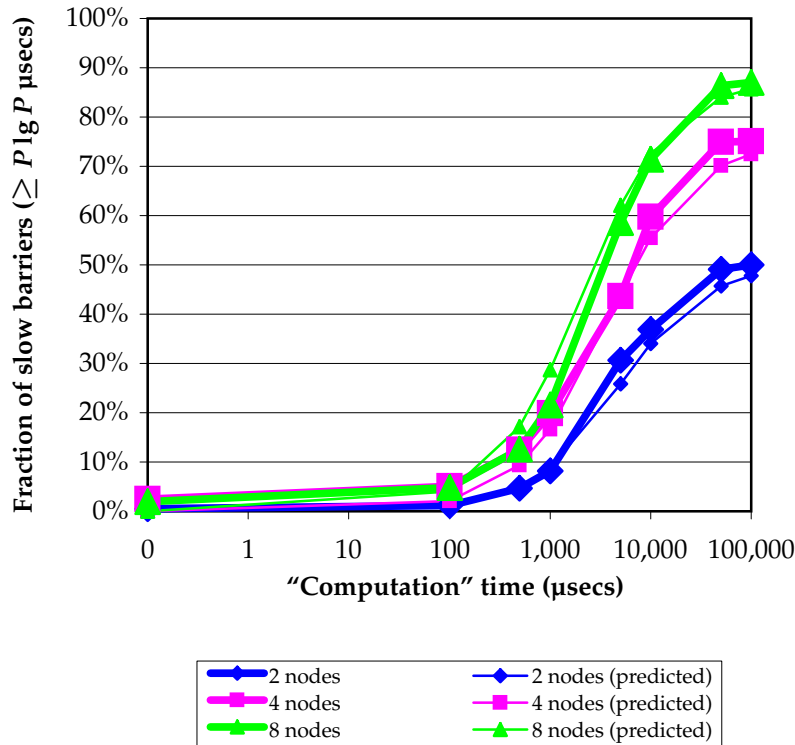


Figure 5.5: Measured vs. predicted tally of slow barriers

in which  $s$  is the fraction of slow barriers ( $s \in [0, 1]$ ),  $d$  is the number of spins (100 spins  $\approx 1 \mu\text{sec}$  in this experiment),  $P$  is the number of processes, and  $\alpha$  is a scaling constant. With the help of *Mathematica*, I determined that  $\alpha \approx 1,876,505$  gives the best fit to my data. Figure 5.5 illustrates that the fit is qualitatively quite good.

Recalling the asymptote at  $\frac{N-1}{N}$ , we can extrapolate that for a sufficiently coarse-grained application, 95% of the barriers in a 20-node cluster will be slow, as will 99% of the barriers in a 100-node cluster. In short, as PC clusters get larger, unresponsiveness will become an increasingly important obstacle to achieving good performance.

**Efficiency** Another way of examining the data from the *barrier* program is to plot it in terms of efficiency. Efficiency is a useful metric because an efficiency graph shows the minimum performance lost to unresponsiveness as an asymptote. If the asymptote is at 100%, that shows that unresponsiveness ceases to be a problem at some “computation” size. If the asymptote is less than 100%, that indicates that unresponsiveness is always present and needs to be dealt with. I define efficiency as:

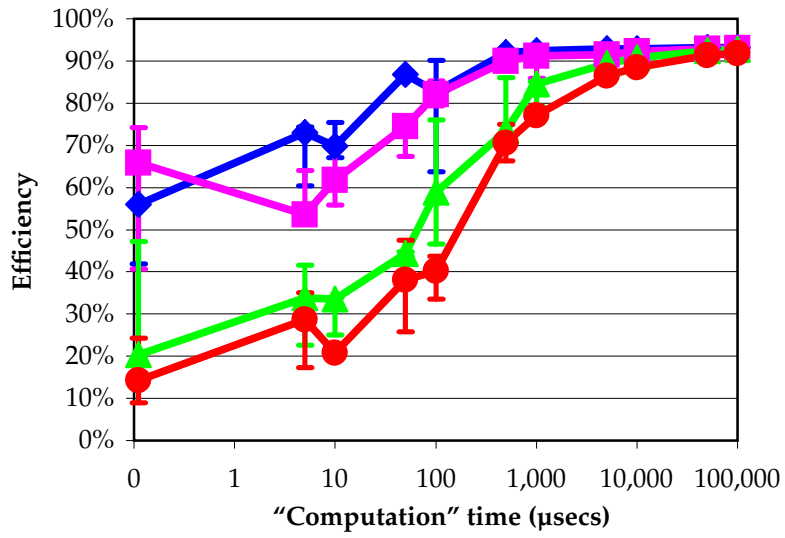
$$\frac{\text{ideal barrier time} + \text{ideal computation time}}{\text{measured barrier time} + \text{measured computation time}}$$

Ideal barrier time is defined as  $9 \lg P$   $\mu\text{sec}$  per barrier, which is the mean barrier time in the no-computation case. This is also somewhat intuitive—9  $\mu\text{sec}$  per message exchange multiplied by  $\lg P$  sets of exchanges. Ideal computation time is defined as  $s/150$   $\mu\text{sec}$ , where  $s$  is the number of spins and dividing by 150 converts from spins to microseconds. ( $\frac{1 \text{ spin}}{3 \text{ cycles}} \times \frac{450 \text{ cycles}}{1 \mu\text{sec}} = 150 \text{ spins}/\mu\text{sec}$ .) Measured computation time was calculated as total elapsed time minus the measured barrier time. Figure 5.6 shows the efficiency of the barrier program as a function of the amount of inter-barrier “computation,” with both one and two CPUs per node (Figures 5.6(a) and (b), respectively). For the one-CPU runs, Windows NT was booted with the /ONECPU option. Error bars are included to show the minimum and maximum efficiency over three runs.

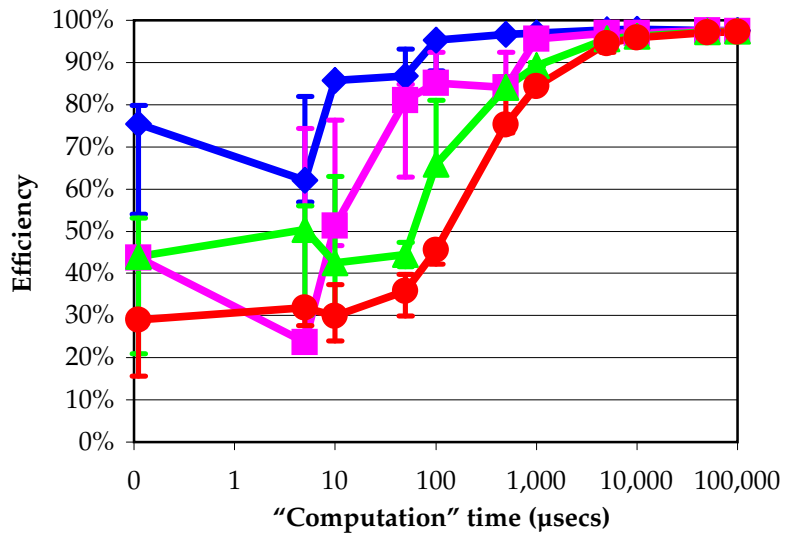
The key observations one should make when considering Figure 5.6 are:

- An increase in the number of processes leads to a decrease in efficiency.
- Fine-grained programs are more susceptible to performance loss than coarse-grained programs. (1,000  $\mu\text{sec}$  can be considered the threshold between “fine-grained” and “coarse-grained” in this context.)
- Not only is efficiency low for fine-grained programs, but—as implied by the large error bars—the variance in efficiency is high.
- A second (idle) CPU absorbs some of the cost of system services that cause unresponsiveness and loss of efficiency. As a result, the single-CPU runs (Figure 5.6(a)) consistently observe less efficiency than the dual-CPU runs (Figure 5.6(b)).
- In both the single- and dual-CPU cases, efficiency reaches an asymptote when computation time is sufficiently high, but that asymptote is below the 100% efficiency mark ( $\approx 93\%$  in the single-CPU case).

**Application interbarrier times** Having characterized the relationship between interbarrier times and barrier times (and barrier efficiency), it is worthwhile to examine the actual times observed between barriers in a sample application. We use *mg* as this application, because it uses barriers extensively and performs a variety of work between barriers. The intention is to plot a histogram of *mg*’s interbarrier times and observe the number of barriers in the “fine-grained” range (which see greater delays) and those in the “coarse-grained” range (which see lesser delays).



(a) One CPU



(b) Two CPUs



Figure 5.6: Barrier program efficiency

Table 5.6: Experimental setup for *mg*

| Component     | Characteristic                       |
|---------------|--------------------------------------|
| Node          |                                      |
| Type          | Compaq Professional Workstation 6000 |
| CPU           | Dual 300 MHz Pentium IIs             |
| Memory        | 128 MB EDO DRAM                      |
| OS            | Red Hat Linux 5.2 (kernel v2.0.36)   |
| Network       |                                      |
| Type          | 10 Mbit Ethernet                     |
| Size          | 8 nodes                              |
| Topology      | Bus                                  |
| Communication | BSPlib over UDP/IP                   |

Recall from Section 5.1.1 that *mg* is written in BSP [101]. In a BSP program, processes alternate computation, communication, and barrier synchronization phases. (In BSP terminology, these are collectively called a *superstep*.) During computation phases, processes compute on local data and initiate point-to-point communication operations (SEND, RECEIVE, PUT, and GET), which are non-blocking and do not *complete* (i.e., change any process' state) until after the next synchronization phase. *mg* performs a sufficient number of barriers to be a worthwhile application to study for its interbarrier times.

Due to software availability, *mg* was run on a somewhat different workstation cluster (described in Table 5.6) than the other experiments in this dissertation. However, the experimental results are still valid. Because the cluster used for *mg* is slower than the one described in Table 5.1, all reported interbarrier timings will, in fact, be *upper* bounds on what could be expected from the more up-to-date equipment described in that table.

Figure 5.7 shows the distribution of interbarrier computation times for an 8-node, class A run of *mg*: four iterations over a  $256 \times 256 \times 256$  matrix. The  $x$  axis represents the time between barriers in microseconds. The  $y$  axis represents a tally of how often each interbarrier time occurred, shown as a cumulative percentage of the total number of barriers. (100% = 700 barriers.) What is evident from the figure is the bimodal distribution of interbarrier times. There is a dense set of tallies in the 10–50  $\mu$ sec range, no data points from 3,000–6,000  $\mu$ sec, and another dense set from 8,000–50,000  $\mu$ sec. Again, Figure 5.6 suggests that the 0-1,000  $\mu$ sec range supports my thesis that there is performance to be had by tolerating unresponsiveness; the 1,000  $\mu$ sec-and-up range contradicts it.



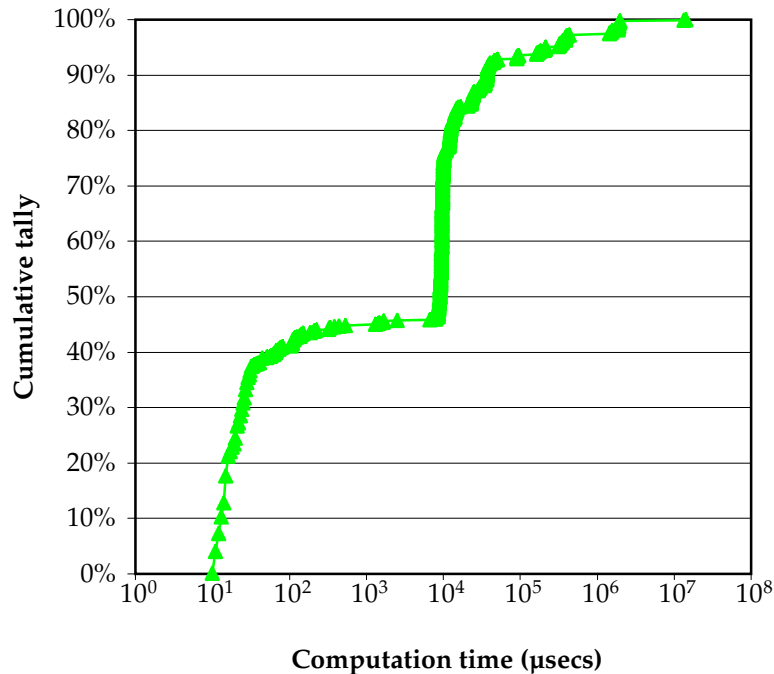


Figure 5.7: Computation time for MG class A, 8 nodes

## 5.2.2 Characterizing unresponsiveness

Figure 5.4, which was described on page 71, showed that barrier time increases with computation time. The question, then, is what is causing this time increase? The number of page faults was fairly constant across all the runs. However, running the barrier program’s processes at high priority<sup>5</sup> does cause a drastic drop in the total barrier time, as shown by the rightmost bar in Figure 5.4. Hypothesizing that the increased barrier time is caused by background OS processes awaking, performing brief tasks, and polluting the cache, I analyzed the correlation between the number of barriers that occurred in each time range and the total number of context switches that occurred during a run of the barrier program. Figure 5.8 depicts the results, which encompass all the data used in Figure 5.4.

Figure 5.8 shows that the number of barriers in the “fast” range (< 30 µsec) is negatively correlated to the number of context switches, while the number of barriers in the “slow” range (30 µsec and up) is positively correlated. In fact, correlation is nearly perfect in the 100–999 µsec range—the same range in which most of the additional overhead occurs in Figure 5.4. This implies that context switches (or, rather, the system processes that are switched to) are causing unresponsiveness even in an

<sup>5</sup>Specifically, this means Windows NT thread priority 15 (THREAD\_PRIORITY\_TIME\_CRITICAL in the HIGH\_PRIORITY\_CLASS).

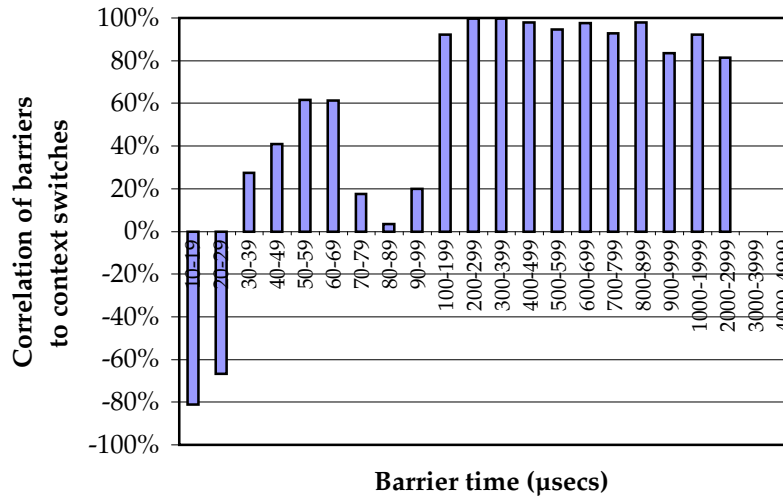


Figure 5.8: Correlation between the number of barriers in each time range and the number of context switches observed

“ideal” setting: perfectly load-balanced computation and only a single (non-system) process per machine.

Having shown the correlation between context switches and slow barriers, the final step in the analysis is to show that these context switches lead to degraded performance. To do so, we run a program (*radix sort*) both on an ordinary, unloaded cluster and on a cluster specially configured to minimize disruption caused by context switches. The latter cluster has two characteristics that make it resource-wasteful and unrealistic for general computation but a reasonable vehicle for studying the impact of unresponsiveness on performance:

1. All of *radix sort*'s processes run at the highest priority that did not lead to priority inversion with the network device driver.
2. Every node in the cluster contains an additional CPU, which is used solely for absorbing system services and other operating system activity that otherwise would have introduced context switches to *radix sort*.

Other than those differences, the two clusters are identical in every way.

Figure 5.9 shows the result of running *radix sort* with 4, 8, and 16 processes (and the same number of nodes) on the “normal” and “responsive” clusters. The *x* axis represents the number of processes (and nodes), and the *y* axis represents the execution time relative to what was measured on the “responsive” cluster. The Responsive bar is therefore fixed at 100%. The No load (blocking) and No load (polling) bars show the normalized performance measurements from the “normal” cluster

with no additional load on the system. No load (blocking) corresponds to *radix sort*'s using blocking notification for the point-to-point messages that comprise its collective-communication operations, and No load (polling) is same, but using polling notification.

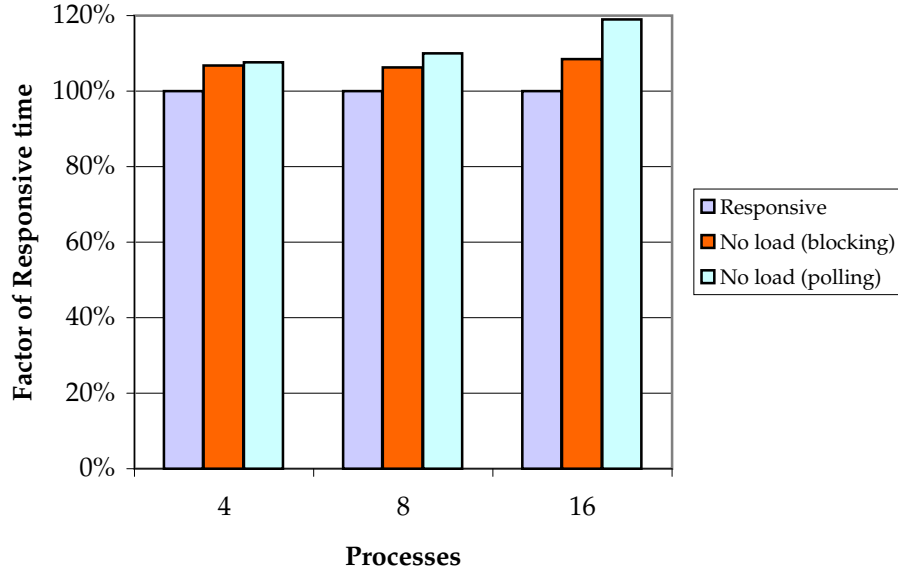


Figure 5.9: *radix sort* performance lost to unresponsiveness

The important observation from Figure 5.9 is that unresponsiveness causes performance loss even on an idle cluster. With 16 processes, *radix sort* runs 8.4% slower than ideal when blocking notification is used and 18.9% slower than ideal when polling notification is used. Furthermore, as the number of processes increases, the performance lost to unresponsiveness increases. This confirms what was stated in Chapter 3: A collective-communication operation runs only as fast as its slowest participant. As the number of participants increases, the likelihood that at least one is unresponsive increases, as well. Extrapolating from the data used in Figure 5.9 out to 12,288 nodes, the proposed size of the forthcoming ASCI Q supercomputer [25], one would expect *radix sort* running on a cluster of that size to be 22.8 times slower than a dedicated supercomputer when blocking is used or 122.3 times slower when polling is used. In short, unresponsiveness is a serious problem for a cluster, even when only a single application is running on it and CPU contention is, at most, sporadic.

One would expect unresponsiveness to be an even more serious problem when there *is* continuous contention for the CPU. This sort of contention comes in two forms, internal and external, as described in Section 5.1.2. We use *cholesky* as the application with which to measure the impact of internal and external conten-

tion on unresponsiveness, because on an idle system, *cholesky* exhibits near-perfect speedup (Figure 5.10). This is convenient for studying performance loss, because we know *a priori* that doubling the number of processes should halve the execution time. If it does not, then we know that unresponsiveness is the culprit.

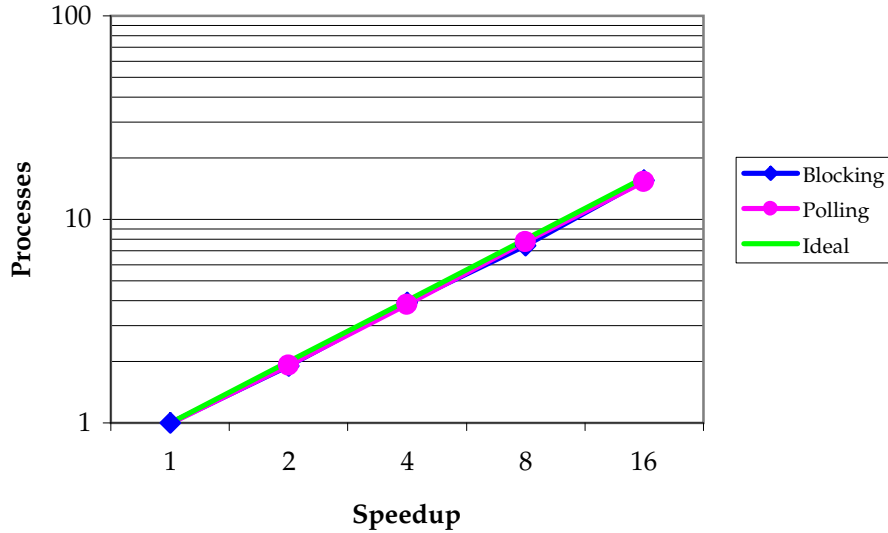


Figure 5.10: Speedup of the *cholesky* code

Internal CPU contention arises when there is only one job running on the cluster, but that job contains more processes than there are available CPUs. As a result, the processes cannot all be coscheduled, which implies that an all-process collective-communication operation cannot complete until each process has been given enough CPU time to complete its participation in the operation.

Figure 5.11 quantifies the impact of internal contention on collective-communication performance. It shows the results for both blocking and polling notification, and with the “naive” reduction algorithm. In each graph, the “P nodes” curve represents the case in which there is only one process per node (i.e., the no-contention case); “P/2 nodes” represents the performance when there are half as many nodes as processes (or, alternatively, two processes per node); and so on for the other curves. Additionally, the “1 node” line highlights the data points in which there is one process on one node, two processes on one node, four processes on one node, and so forth.

The most striking observation about the graphs in Figure 5.11 is the qualitative difference in performance between the blocking and polling versions of *cholesky* when the number of nodes is held constant but the number of processes varies. In the blocking version (Figure 5.11(a)), *cholesky* runs only 9.4% slower when there are 16 processes on a single node than when there is only one process on that node. In

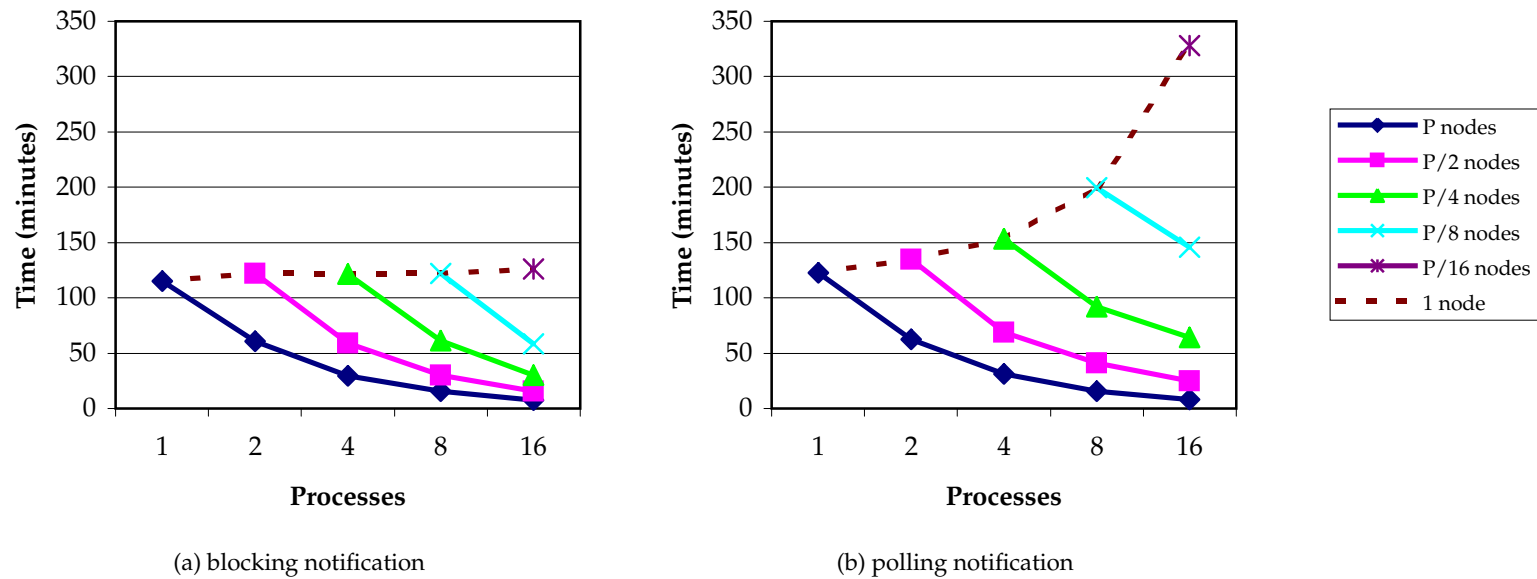


Figure 5.11: Impact of internal contention (naive reductions)

contrast, in the polling version (Figure 5.11(b)), *cholesky* runs 167.6% slower when there is 16 times as much parallelism, but 16 times the number of processes per node. The reason that polling performance deteriorates with increased resource contention is that processes may waste an entire time quantum (20 milliseconds on Windows NT 4.0 TSE) waiting for a message to arrive that *can't* arrive until the waiting process (or another waiting process on another node) relinquishes the CPU. Blocking performance, on the other hand, stays mostly constant when the number of nodes is fixed but the number of processes increases. This is because the increase in parallelism is countered by an equal increase in contention for the CPU. It also indicates that the overhead due to blocking (i.e., the context-switch overhead) adds only a small amount to the total performance loss, at least for large increases in the number of processes + contention. We empirically determined that it takes  $\approx 4 \mu\text{sec}$  to context switch to another thread and back on the **machines used in this dissertation**.

Figure 5.12 shows the same data as Figure 5.11, but with each data point normalized to the “ideal” time. The ideal time for a run of *cholesky* with  $P$  processes and  $N$  nodes is defined as  $I(P, N) \equiv I(1, 1)/N$ . In other words, doubling the number of nodes doubles the performance, while doubling the number of processes does not affect the performance.  $I(1, 1)$  is defined to match the empirically measured time. According to Figure 5.12, *cholesky* performs close to ideal when it uses blocking communication. However, when it uses polling notification, the performance diverges from ideal as contention increases. The execution time of the the polling version of *cholesky* can be approximated with the formula  $T_p(P, N) \approx I(P, N) \cdot (1 + \frac{P}{10} - \frac{N}{10})$ , with an  $R^2$  value of 0.960406.<sup>6</sup> In other words, when polling notification is used, increasing the number of processes or decreasing the number of nodes increases the difference between the ideal time and the observed time.

Having shown that internal contention is a source of significant unresponsiveness, we now turn our attention to external contention. External contention arises when the application under investigation is running on the cluster, with only one process per node, but there are a varying number of other processes in the system. Figure 5.13 graphs the results of using the “naive” reduction algorithm for both blocking and polling notification. The graph showing the results of using the “binary” reduction algorithm is virtually identical to Figure 5.13 and is therefore omitted from this dissertation. Figure 5.13(a) plots absolute time (in minutes) on the  $z$  axis, and Figure 5.13(b) plots the same data, but with the  $z$  axis representing the factor slower than the base (no contention) case. In each graph, there are three pairs

---

<sup>6</sup>This formula is a simplification for clarity of exposition. The actual result of the linear regression is  $T_p(P, N) \approx I(P, N) \cdot (1.04129 + 0.105949P - 0.106744N)$  and has an  $R^2$  value of 0.977693.

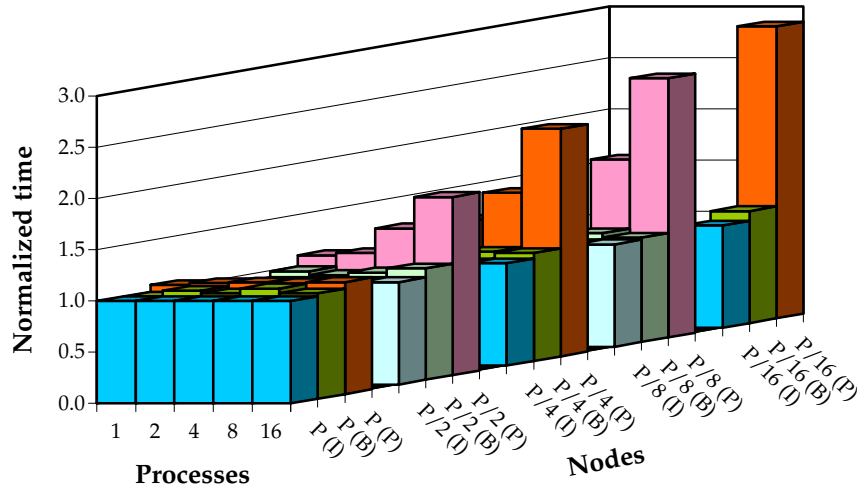


Figure 5.12: *cholesky* performance relative to ideal

of bars. The first pair is the base case: no competitors. The second pair, “(1)”, represents the case in which a CPU-intensive process is running on node 1.<sup>7</sup> And the final pair, “(1..N)”, represents the case in which a CPU-intensive process is running on each node in the cluster.

The interesting observations from Figure 5.13 are:

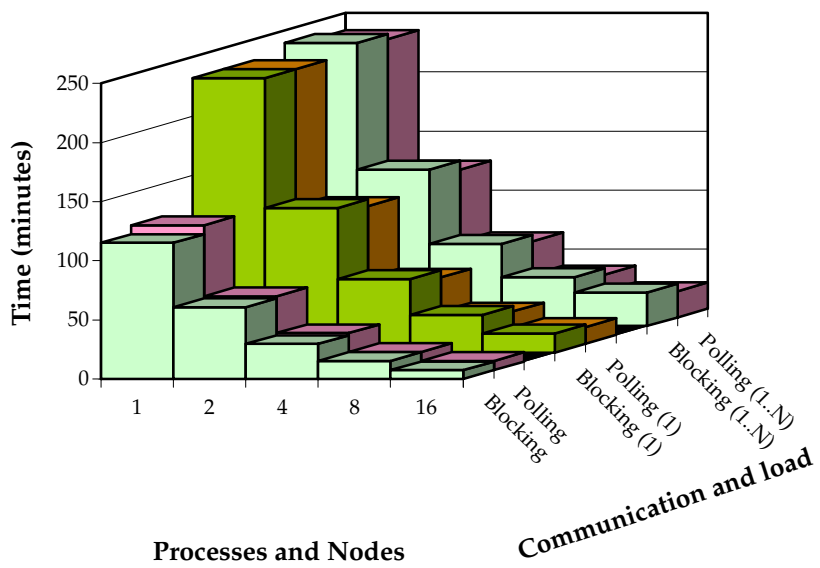
- A single slow node roughly doubles the execution time for any number of processes. (The height of the “(1)” curves in Figure 5.13(b) is approximately equal to 2.0.)
- As can be seen by comparing the Blocking and Polling bars in Figure 5.13(b), a competitor on every node causes the execution time to increase proportionally to the number of nodes when blocking is used ( $T_b(P, 1..N) \approx T_b(P, 0) \cdot (2 + \frac{P}{10})$ ), but to roughly double when polling is used. Actually, the Polling curves do increase linearly, but with a much smaller constant than Blocking.
- *cholesky* is less sensitive to external contention when using polling notification than when using blocking communication—the Polling bars in Figure 5.13(b) are generally shorter than the corresponding Blocking bars.

### 5.2.3 Summary

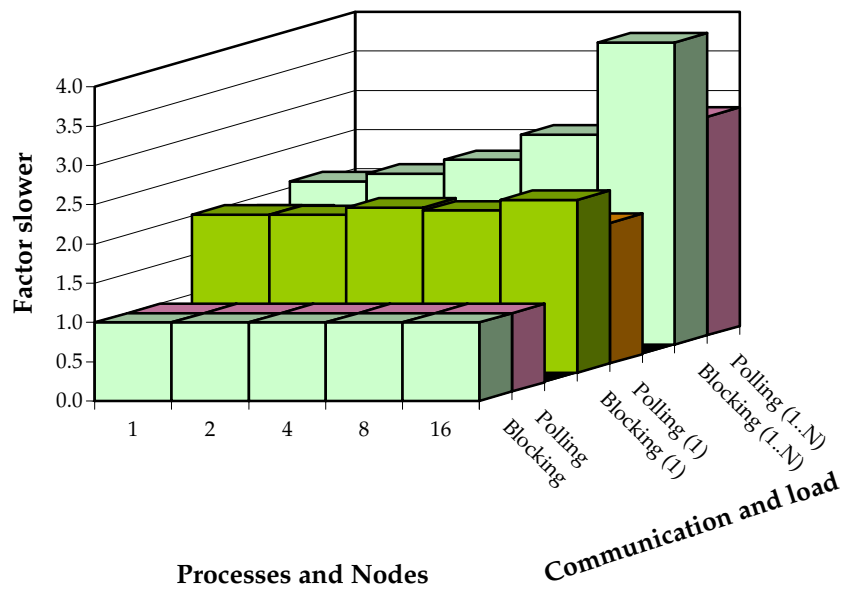
The following are the key points made in Section 5.2:

1. Unresponsiveness is a problem, even when the cluster is essentially idle.

<sup>7</sup>Note that the root of the reduction operation rotates round-robin among all the processes.



(a) Absolute



(b) Relative

Figure 5.13: *cholesky* performance in light of a CPU-intensive competitor



2. The performance penalty is expected to increase with the number of processes involved in the computation.
3. The key source of unresponsiveness is context switches to either a competing application (external contention) or the “wrong” process in the same application (internal contention).

We additionally observed that nonblocking barriers must be robust to both polling and blocking notification. Without unresponsiveness tolerance, polling gives better performance in the presence of external contention, while blocking gives better performance in the presence of internal contention. In the next few sections, we characterize the performance gain from nonblocking barriers when each notification mechanism is used for the underlying point-to-point communication.

## 5.3 Nonblocking barrier performance

To evaluate the performance of nonblocking barriers, we first examine the additional overhead they introduce for their bookkeeping (Section 5.3.1). We then evaluate the performance gain achievable by nonblocking barriers in the presence and absence of internal load imbalance (Section 5.3.2).

### 5.3.1 Bookkeeping overhead

As can be inferred from the nonblocking-barrier implementations presented in Chapter 4, nonblocking barriers require more bookkeeping than traditional barriers. A valid question to ask, therefore, is: What is the cost of this additional bookkeeping? An experiment to answer that question can be set up as follows. We configure the communication layer (VIA++ [84]) to use traditional barriers, but to attach the metadata required by nonblocking barriers to each message. Because this metadata is unused by traditional barriers, it constitutes pure overhead. Therefore, the difference in performance between the extraneous-metadata run and the no-extraneous-metadata run is exactly the overhead incurred by nonblocking barriers. Additionally, we can examine the performance regained from tolerating unresponsiveness and see what fraction of the overhead can be eliminated.

We use the naive implementation of *prefix scan* as the benchmark to use in this experiment. *Naive prefix scan* is an idea choice, because it represents a worst-case scenario for nonblocking barriers. Flow control of *naive prefix scan*'s abundant fine-grained communication introduces communication dependencies, which prevent nonblocking barriers from overlapping idle time with useful work.

Figure 5.14 shows the results of this experiment for both blocking and polling notification. The Traditional bars illustrate the time (in minutes) to run *naive prefix scan* using traditional barriers. The Traditional + metadata bars illustrate the time taken when using traditional barriers which carry the overhead introduced by nonblocking barriers. And the Nonblocking bars illustrate the time using true nonblocking barriers. The difference between the height of the Traditional and Traditional + metadata bars is the amount of overhead added by the software implementation of nonblocking barriers. The difference between the height of the Traditional + metadata and Nonblocking bars is the performance regained purely by tolerating unresponsiveness.

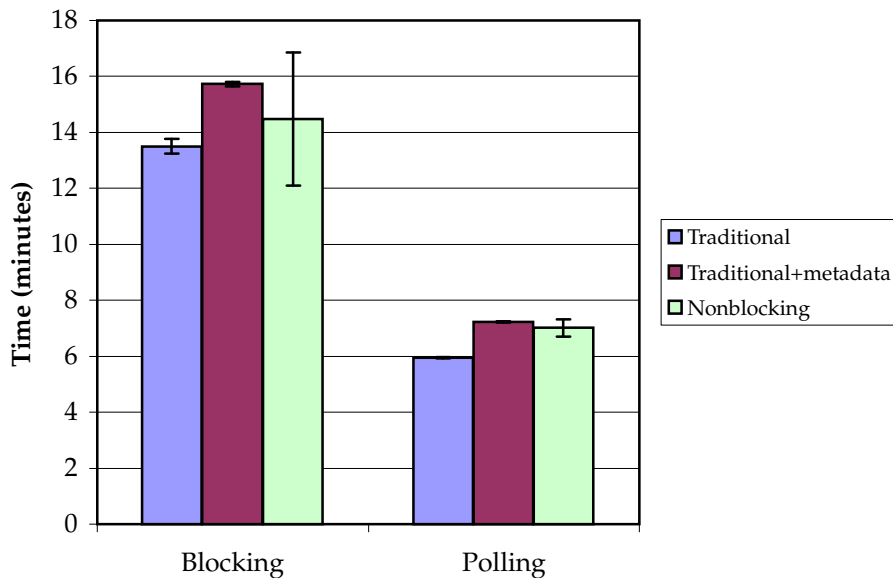


Figure 5.14: Performance of the “naive” *prefix scan*

The conclusions one can draw from Figure 5.14 are:

- There is unresponsiveness in an idle system running the *naive prefix scan*.
- Some of this unresponsiveness can be tolerated.
- The overhead due to carrying barrier metadata dominates the time regained by tolerating unresponsiveness in the *naive prefix scan* benchmark.

Regarding that last conclusion, each message carries with it up to 40 bytes of metadata, while the payload itself is merely 24 bytes. Furthermore, each process receives between 1 million and 10 million messages during the course of its execution, which adds up to a significant cost in total execution time: an extra 16% when blocking notification is used and an extra 21% when polling notification is used.

Note that communication in the naive implementation of *prefix scan* is tightly coupled. This is not due to data dependencies in Algorithm 5.3, but rather to the flow control induced by limited communication resources in the NIC. Table 5.7 lists some of the relevant limitations of the Gigaset cLAN1000, which were determined by invoking the VIPL routine `VipQueryNic()`. In particular, there can be at most 1023 outstanding messages on any VI. While additional VIs per process can be used (at the expense of scalability), there can be no more than 1024 VIs per node. Even if all 1024 VIs were used, messages after the 65,535th would still be dropped, because the completion queue would overflow. If all 1024 completion queues were used—which would substantially increase the cost of notification—only 896 MB of memory can be registered at once, which is insufficient for that number of descriptors. And while the communication layer used by the naive *prefix scan* could be modified to take advantage of more NIC resources and thereby lessen the interprocess coupling, one of the **quintessential characteristics of a PC cluster** is that NICs will always have limited resources relative to the host computer, and therefore, flow control will always be required to prevent data loss due to oversubscribed resources.

Table 5.7: Gigaset cLAN1000 NIC parameters

| NIC attribute (maximum)              | cLAN1000 value  |
|--------------------------------------|---|
| Message size                         | 65,519 bytes  |
| Number of VIs                        | 1024  |
| Number of pending descriptors per VI | 1023  |
| Entries per completion queue         | 65,535  |
| Number of completion queues          | 1024  |
| Amount of registerable memory        | 229,376 regions, each with up to 229,376 contiguous bytes, up to a maximum of 939,524,096 bytes total |

### 5.3.2 Performance gain from nonblocking barriers

While flow control kept the naive implementation of *prefix scan* tightly coupled, the cluster-optimized version of *prefix scan* exhibits a different challenge: internal load imbalance. Figure 5.15 depicts the communication pattern used by an 8-process optimized prefix-scan operation and the corresponding data dependencies. As the figure shows, processes 0–3 receive no messages after the final barrier and can therefore exit as soon as their participation in the barrier is complete. Processes 4–7, however, must each wait for an additional message to arrive. They then have to perform an additional local computation step (the final **for** loop in Algorithm 5.4) before they can exit. Hence, even in the absence of unresponsiveness, processes 4–7 have more

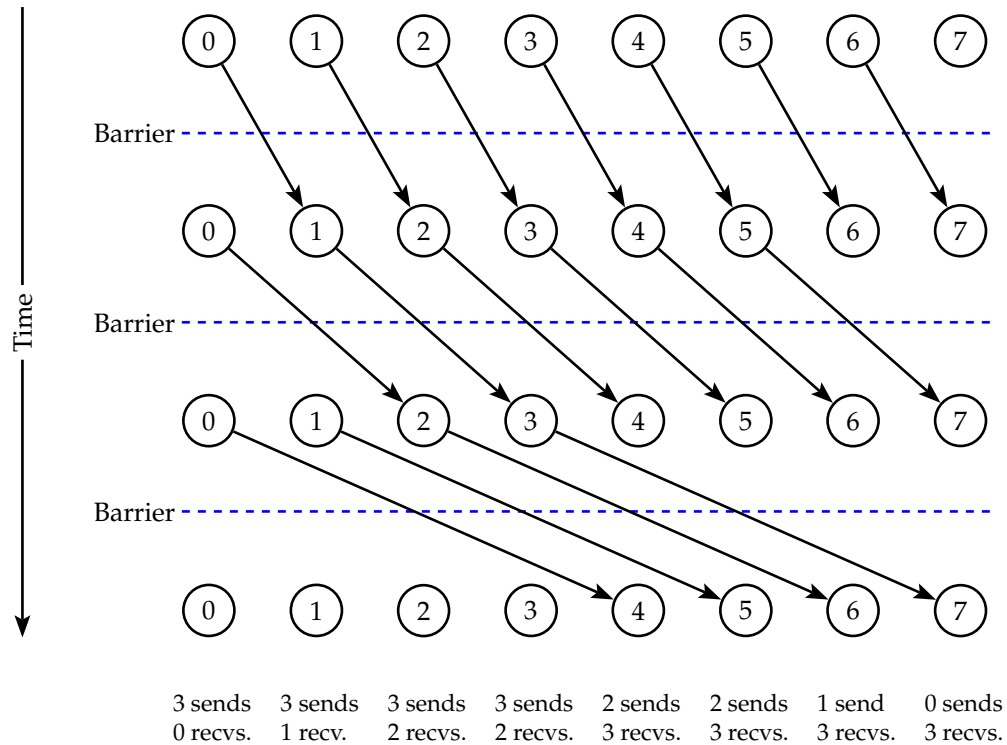


Figure 5.15: Communication pattern for a prefix-scan operation

work to do than processes 0–3. Of course, if one of the first four processes in the computation is unresponsive, its peer will not be able to start its local computation until the unresponsive process awakens.

Figure 5.16 shows the performance of the cluster-optimized version of *prefix scan* with each of traditional and nonblocking barriers. The figure includes the time measured individually on each of the eight processes partaking in the computation in addition to the median time. Figure 5.16(a) shows the performance when polling is used for message notification, and Figure 5.16(b) shows the performance when blocking is used. Performance measurements are shown for both traditional barriers and nonblocking barriers.

Some key observations one should make from the measurements in Figure 5.16 are the following:

- The performance when using traditional barriers is bimodal; half the processes finish faster than the other half.
- Processes 4–7, the slow processes, complete in approximately the same time whether traditional or nonblocking barriers are used.

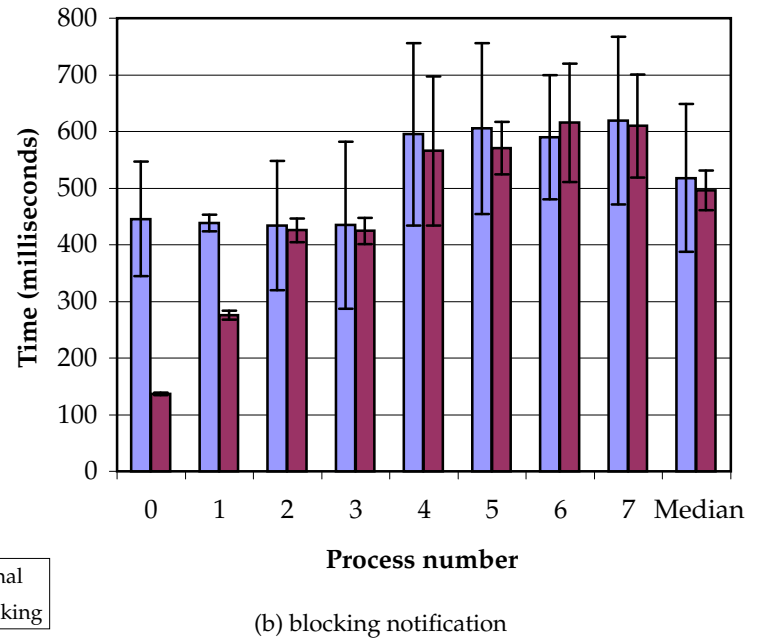
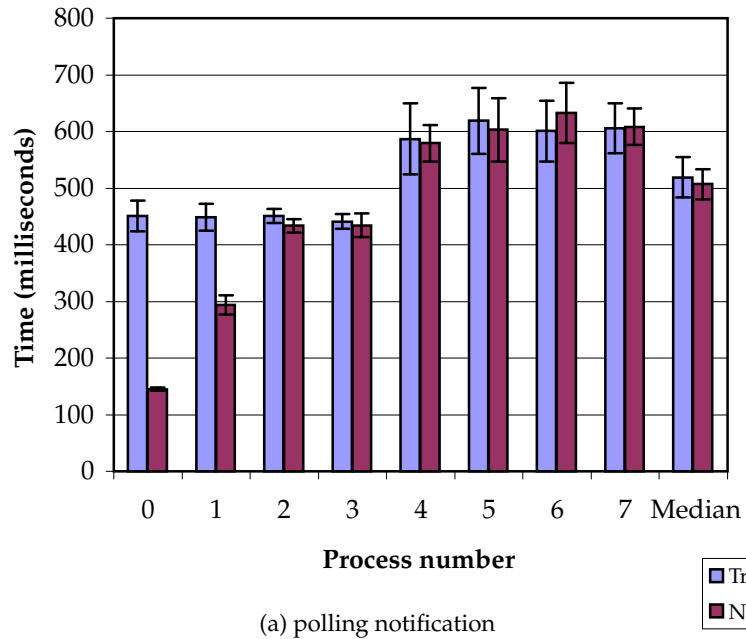


Figure 5.16: Performance of the cluster-optimized *prefix scan*

- Processes 0–3, the fast processes, finish much faster when nonblocking barriers are used.

As would be expected, in the no-competition case, nonblocking barriers do little to improve the performance of processes 4–7, because the same data dependencies and load imbalance apply as in the traditional barrier case. However, processes 0–3 can finish sooner because they have less work and fewer data dependencies. As Figure 5.15 indicates, process 0 receives no messages. Nonblocking barriers therefore enable it to perform all its local computation at once, participate in the three barriers, and then terminate. Process 1 has only one data dependency, and after receiving its single message, performing the dependent local computation, and participating in the barriers, it can terminate. Processes 2 and 3 each have to receive two messages and perform some local computation before they can terminate. The height of each nonblocking-barrier bar in Figure 5.16 is therefore proportional to the number of data dependencies (message receives) the corresponding process has. To be more precise, the execution time for the cluster-optimized *prefix scan* can be approximated by the following equations:

$$T_{\text{trad}}[p] \approx \begin{cases} \lceil \lg P \rceil T_{\text{local}} & \text{if } p \geq P/2 \\ (1 + \lceil \lg P \rceil) T_{\text{local}} & \text{if } p < P/2 \end{cases} \quad (5.2)$$

$$T_{\text{nblock}}[p] \approx (1 + \lceil \lg(p + 1) \rceil) T_{\text{local}} \quad (5.3)$$

Equation 5.2 approximates the time when using traditional barriers ( $T_{\text{trad}}$ ), and Equation 5.3 approximates the time when using nonblocking barriers ( $T_{\text{nblock}}$ ).  $T_{\text{local}}$  is the time spent in a single block of local computation and is measured by timing the first stanza of Algorithm 5.4.  $T_{\text{local}}$  corresponds to the height of the process 0 nonblocking bar in Figure 5.16. (It is also the difference in height between the “slow” and “fast” traditional bars in that figure.) The variables  $P$  and  $p$  have the same meanings as in Table 5.4 on page 63. As Figure 5.17 shows, Equations 5.2 and 5.3 accurately<sup>8</sup> predict the performance of *prefix scan* when using either traditional or nonblocking barriers. (The reason error exists in the analytic plots is that there is measurement error in  $T_{\text{local}}$ , which propagates through Equations 5.2 and 5.3.) The difference between  $T_{\text{trad}}$  and  $T_{\text{nblock}}$  therefore provides a simple, but accurate, analytical model of the minimal amount of additional performance that can be seen by each process

<sup>8</sup>The  $R^2$  value when predicting the performance of traditional barriers is 0.987866, and the  $R^2$  value when predicting the performance of nonblocking barriers is 0.988998.

on this particular problem when traditional barriers are replaced with nonblocking ones.

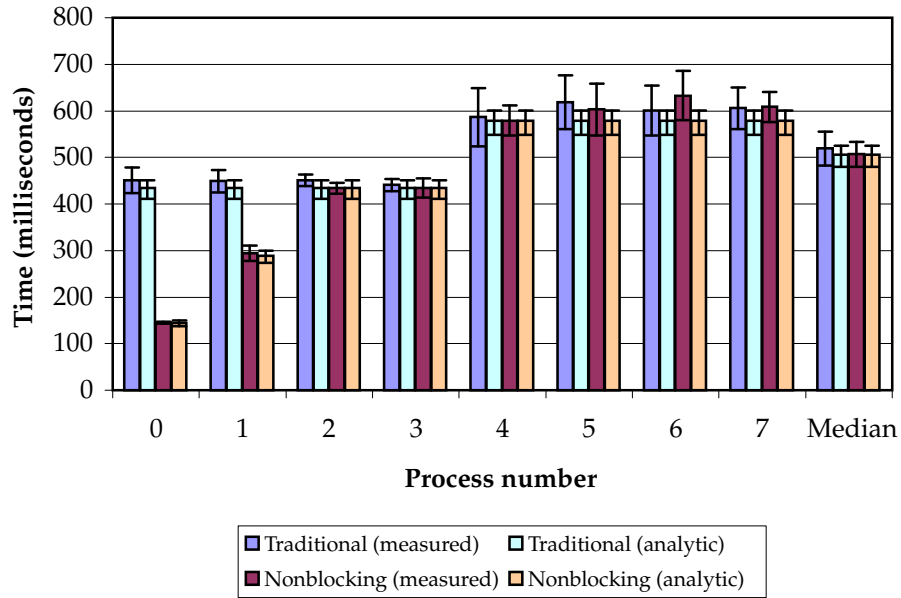


Figure 5.17: Comparison of measured vs. analytic *prefix scan* time (polling notification)

When unresponsiveness is explicitly introduced into the system in the form of a competitor for the CPU (external contention), nonblocking barriers improve performance even further than in the no-load case. Figure 5.18 shows the performance of traditional versus nonblocking barriers on the cluster-optimized prefix scan microbenchmark, but unlike Figure 5.16, there is a competitor for the CPU (a simple spin loop) running on each node. While the measured times in Figure 5.18 are greater than in Figure 5.16—note the larger values on the *y* axes—the observations made on page 88 still apply, although the bimodality of the measurements in the traditional barrier case is less apparent, due to the greater variation in execution times due to process scheduling involving an additional process.

Finally, Figure 5.19 summarizes the performance gained by tolerating unresponsiveness in *prefix scan*. Each bar represents the ratio of the time spent when nonblocking barriers are used to the time spent when traditional barriers are used. A ratio of 1.0 means there was no performance improvement. Ratios less than 1.0 mean unresponsiveness was tolerated and performance was improved. Ratios greater than 1.0 mean the overhead from nonblocking barriers dominated the performance gain from tolerating unresponsiveness. As Figure 5.19 shows, nonblocking barriers improve the performance observed by each process by an average factor of 0.81

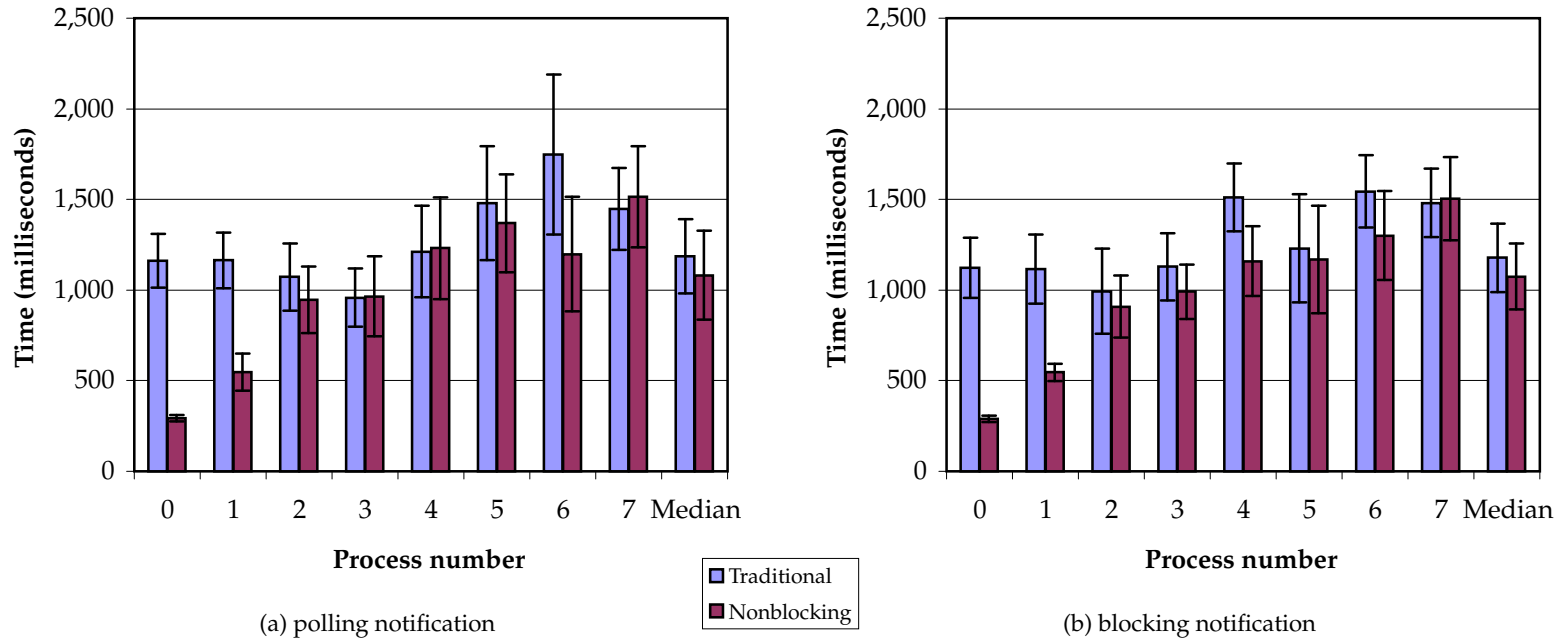


Figure 5.18: Performance of the cluster-optimized *prefix scan* with one competitor per node



when there is no competition for the CPU and by an average factor of 0.71 when there is competition for the CPU.

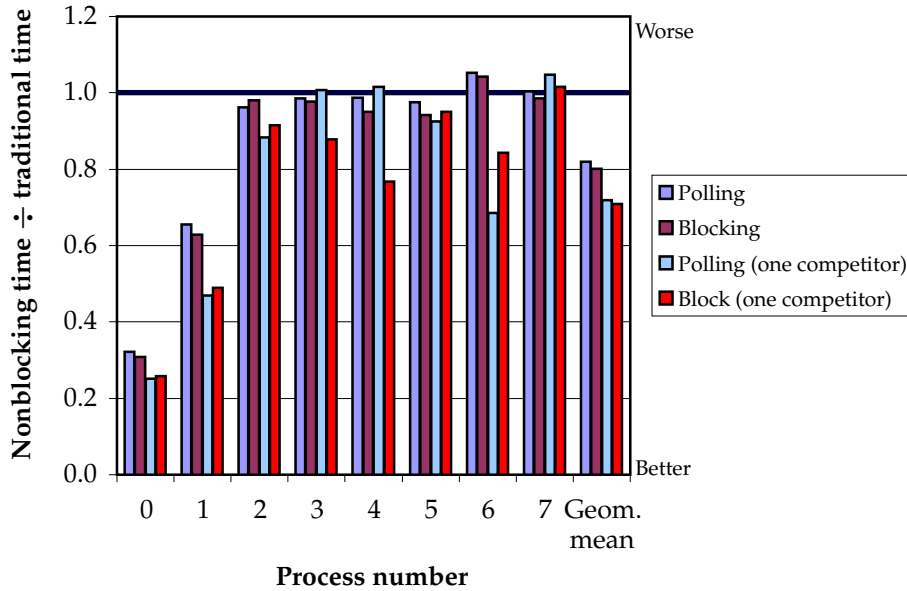


Figure 5.19: Performance gain from tolerating unresponsiveness in the cluster-optimized *prefix scan*

Unlike *prefix scan*'s processes, all of *radix sort*'s processes perform an equal amount of communication. Figure 5.20 shows the performance improvement observed when running *radix sort* on an otherwise idle cluster.<sup>9</sup> When polling notification is used, nonblocking barriers decrease the penalty due to unresponsiveness from 9.9% down to 6.3%, and when blocking notification is used, nonblocking barriers decrease the penalty due to unresponsiveness from 6.3% down to 4.8%. Although this improvement is small, we will see in the following section that nonblocking barriers improve performance significantly when there is internal or external contention for the CPU.

### 5.3.3 Sources of performance gain

Figure 5.21 provides an expanded view of the performance gain contributed by nonblocking barriers. The goal is to characterize the effect that nonblocking barriers have on waiting times. Our hypothesis is that nonblocking barriers reduce the amount of time that a process spends waiting for point-to-point messages to arrive. Because nonblocking barriers enable non-barrier work to overlap the time

<sup>9</sup>Figure 5.20 portrays a superset of the data that appeared in the top part of Figure 3.1.

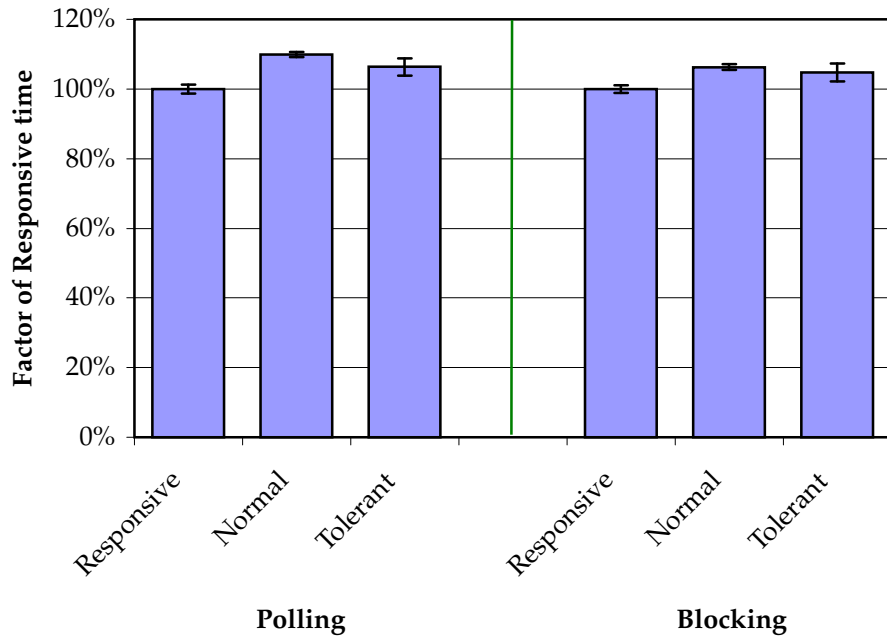
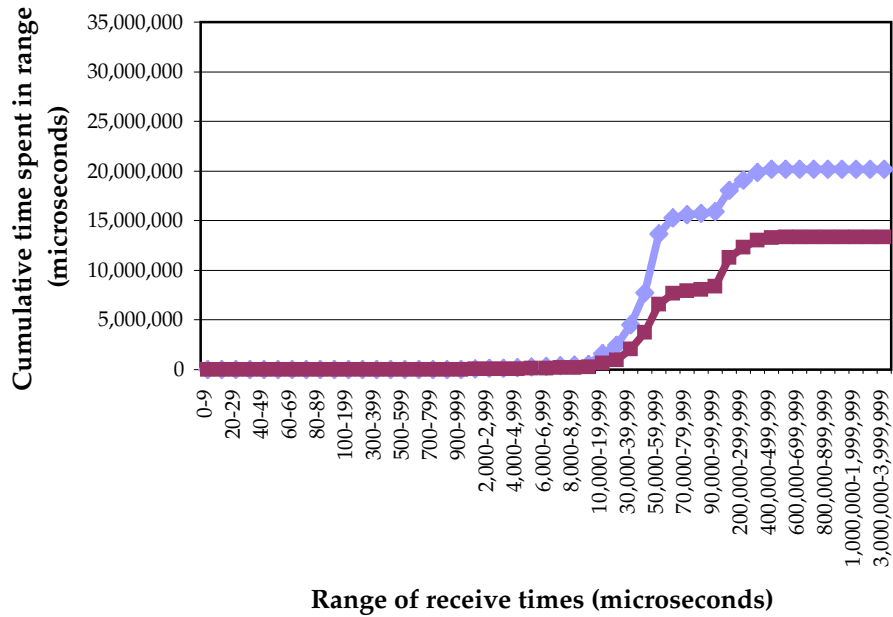


Figure 5.20: Performance gain from tolerating unresponsiveness in *radix sort*

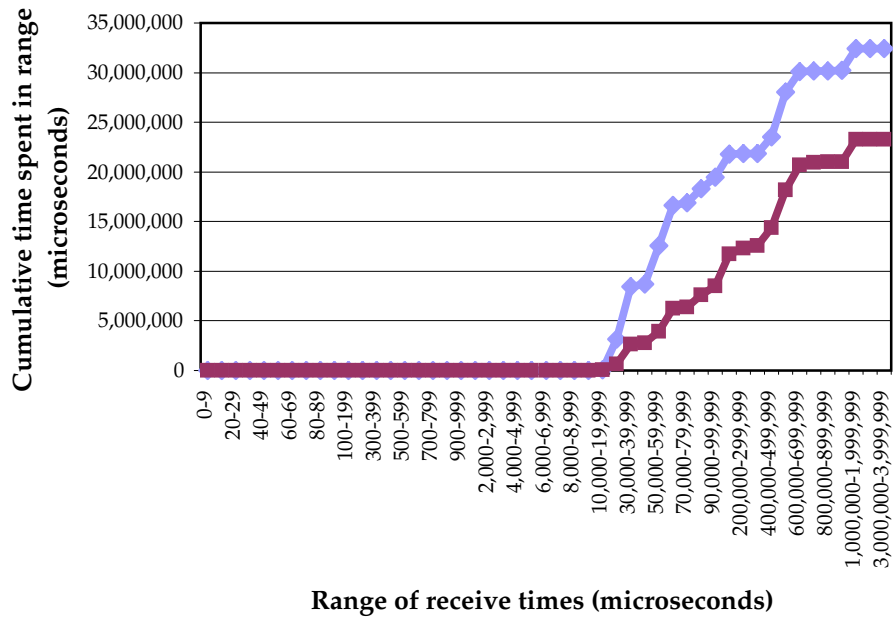
spent within a barrier, we expect to see to total waiting time drop by the amount of time spent in overlapped work.

The experimental setup consisted of an 8-process run of *radix sort* with a competitor process sharing each node in the cluster. The messaging layer was instrumented to log the time spent in each invocation of the point-to-point receive function. Figure 5.21(a) characterizes the performance when polling notification is used for point-to-point messages, and Figure 5.21(b) characterizes the performance when blocking notification is used. In each subfigure, the  $x$  axis shows ranges of receive times, and the  $y$  axis shows the contribution of each range and all preceding ranges to the total waiting time. Data for both traditional and nonblocking barriers are shown.

Two facts are evident from Figure 5.21. First, nonblocking barriers do decrease the time spent waiting on incoming point-to-point messages. The vertical difference between the rightmost points of the two curves (i.e., the difference in total waiting time) is 6.8 seconds in Figure 5.21(a) and 9.2 seconds in Figure 5.21(b). This means that nonblocking barriers overlapped 6.8 (respectively, 9.2) seconds' worth of work with the execution of the barrier. That is, nonblocking barriers successfully hid 33.8% (respectively, 28.2%) of the original, total waiting time behind useful work. The second point that Figure 5.21 elucidates is that the real divergence in cumulative time occurs in the OS time-quantum ranges for compute-bound processes—those between 10,000 and 50,000  $\mu$ secs. Beyond those ranges, the curves roughly parallel



(a) Polling notification



(b) Blocking notification



Figure 5.21: Cumulative time blocked on receives in *radix sort*

each other. This implies that nonblocking barriers are able to overlap useful work with time that would otherwise be spent waiting for an unscheduled peer to be rescheduled and regain responsiveness.

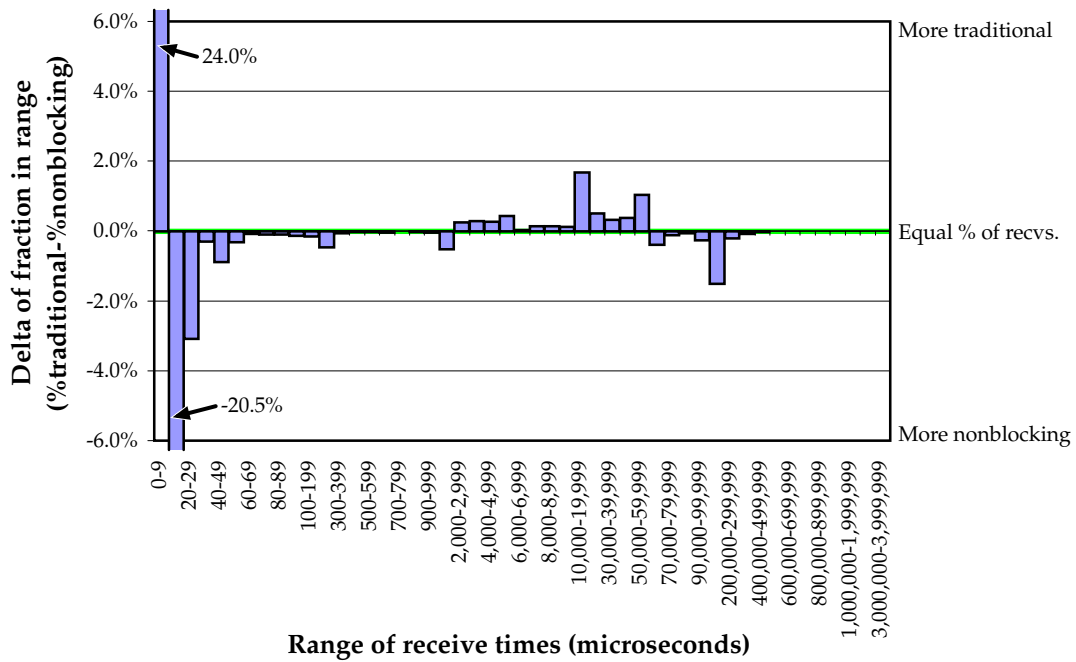
Figure 5.22 verifies the latter point. Like Figure 5.21, it shows data for both both polling and blocking notification (subfigures 5.22(a) and 5.22(b)) and shows ranges of receive times on the  $x$  axes. The  $y$  axes, however, present tallies (out of 100%) instead of times. More precisely, the  $y$  axes show the *difference* between the traditional and nonblocking tallies. This arrangement makes it easy to contrast the receive times observed when using traditional barriers and when using nonblocking barriers. A positive bar indicates that the corresponding receive times are more common when traditional barriers are used; a negative bar indicates that the corresponding receive times are more common when nonblocking barriers are used; and a zero bar indicates that receive times in the corresponding range are equally common regardless of barrier implementation. The  $y$  axes in Figure 5.22 are truncated to  $\pm 6\%$  to make it easier to see the bulk of the graphs.

Figure 5.22 supports our hypothesis that nonblocking barriers are able to overlap useful work with time that would otherwise be spent waiting for an unscheduled peer to be rescheduled and regain responsiveness. The bars in the 10,000–50,000  $\mu\text{sec}$  ranges are predominantly positive, and the bars in the lesser-delay ranges are predominantly negative. The conclusion that one can draw from this is that nonblocking barriers can frequently reduce quantum-sized delays to much smaller levels by pushing forward with the computation instead of waiting idly for an unscheduled peer.

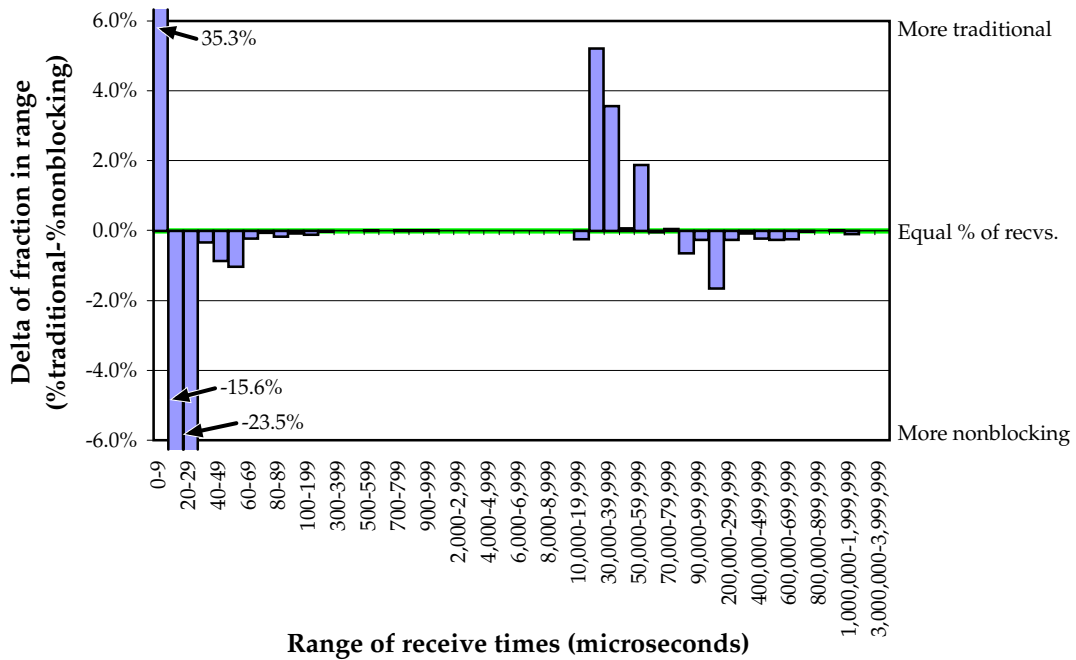
There are two additional observations that can be made from Figure 5.22. First, the overhead induced by nonblocking barriers' bookkeeping makes 0–9  $\mu\text{sec}$  receives much less common than in the traditional-barrier case. (10–29  $\mu\text{sec}$  receives, however, are more common when nonblocking barriers are used.) Second, the 60,000  $\mu\text{sec}$ -and-up ranges appear predominantly when nonblocking barriers are used. This is likely the effect of a process overlapping a significant amount of work with a barrier, running far ahead of another process, and finally reaching a point at which it can make no more progress until the slower process catches up. Fortunately, these long receive delays are few in number and, as shown by Figure 5.21, do not noticeably decrease the gap between the Traditional and Nonblocking curves.

## 5.4 Performance robustness

This section investigates the robustness of nonblocking barriers. Section 5.4.1 shows that nonblocking barriers continue to improve performance when used alongside other unresponsiveness-tolerating techniques. That is, the performance gain is ad-



(a) Polling notification



(b) Blocking notification

Figure 5.22: Time blocked on receives in *radix sort*, expressed as the difference in the percentage tally of receive times

ditive. Section 5.4.2 shows that nonblocking barriers increasingly improve performance as cluster size grows.

### 5.4.1 Compatibility with other unresponsiveness-tolerating techniques

It is instructive to evaluate how well nonblocking barriers perform when used in conjunction with other unresponsiveness-tolerating techniques. The unresponsiveness-tolerating techniques we use for this evaluation are *flat trees* and *smart exchanges*. With flat trees, which were illustrated earlier in Figure 5.3, the root of a multicast operation sends directly to each of the remaining processes. In a reduction operation, the flow direction is reversed; all processes send directly to the root. While flat trees take linear time, as opposed to the logarithmic time required by a binary tree, the intuition is that performance will nevertheless be improved. The reasons are as follows:

1. An unresponsive participant in a binary tree will delay communication to all processes downstream of it. An unresponsive participant in a flat tree—excluding the root process—does not impact other processes.
2. Because communication overhead is low and the performance penalty from unresponsiveness can be high, the performance lost to using an  $O(n)$  algorithm instead of an  $O(\lg n)$  algorithm is expected to be regained by tolerating unresponsiveness.

Smart exchanges alter the scheduling of the point-to-point messages that comprise an all-to-all exchange. Ordinarily, with “dumb” exchanges, each process sends a message to its first neighbor for whom it has a message to send. In contrast, with smart exchanges, each process communicates with a peer that is likely to be responsive at the time. This approach is similar to that taken by adaptive thread-scheduling algorithms such as dynamic coscheduling [95, 96] and implicit coscheduling [5]. Smart exchanges are implemented by having each process send a message to whichever process last sent to it. If a process has no messages for that particular peer, it temporarily reverts to ordinary message scheduling, sending to its first neighbor for whom it has a message to send. The intuition behind smart exchanges, just like that behind dynamic coscheduling and implicit coscheduling, is that receiving a message implies that the sender is likely to still be running and therefore be responsive to the network.

For clarity of exposition, the figures that follow in this section utilize only four combinations of values from the set {“binary” vs. “flat”, “traditional” vs. “nonblocking”, “dumb” vs. “smart”}. These combinations are enumerated and summarized in Table 5.8. The four combinations are labeled Base, +flat, +nonblocking, and +smart

exch. Base corresponds to the case in which no unresponsiveness-tolerant mechanisms are used. With +flat, flat communication trees are the only mechanism in place. +nonblocking adds nonblocking barriers to +flat. Finally, +smart exch. adds adaptive message scheduling to +nonblocking.

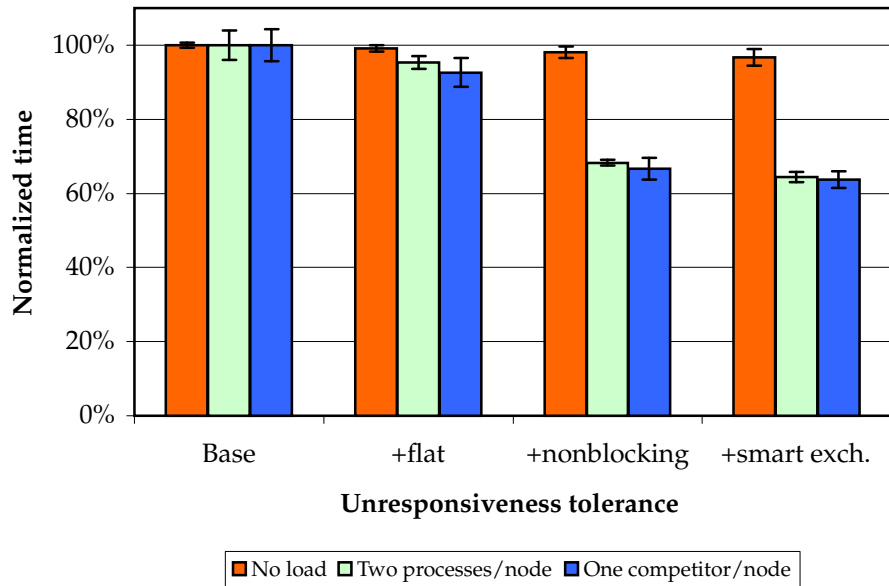
Table 5.8: Sets of unresponsiveness-tolerating techniques used in radix-sort figures

| Name         | Variable values |             |           |
|--------------|-----------------|-------------|-----------|
|              | Tree            | Barriers    | Exchanges |
| Base         | binary          | traditional | dumb      |
| +flat        | flat            | traditional | dumb      |
| +nonblocking | flat            | nonblocking | dumb      |
| +smart exch. | flat            | nonblocking | smart     |

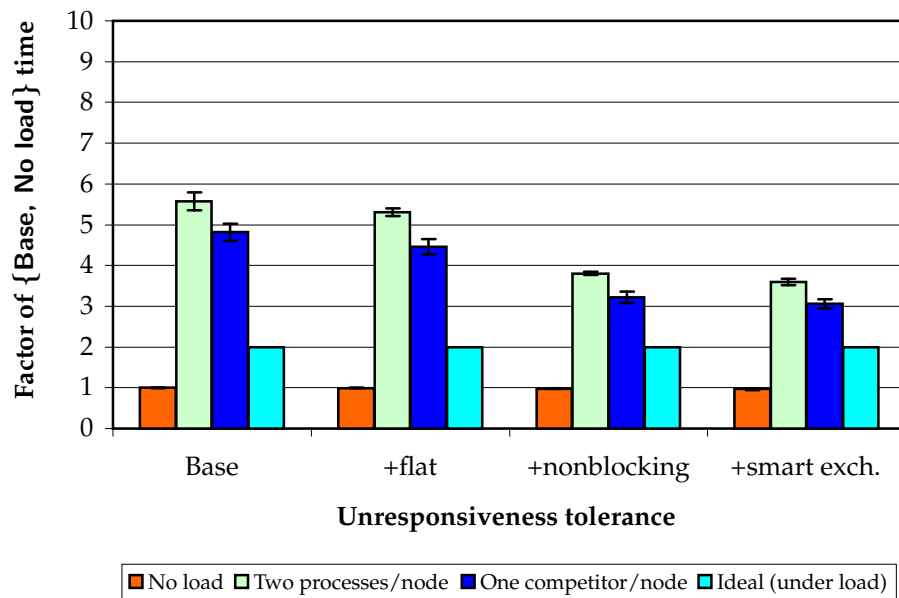
Figure 5.23 shows the performance improvement due to each of +flat, +nonblocking, and +smart exch. over Base. There are three data series in the graph, one for each condition under which the experiments were carried out: No load, Two processes/node, and One competitor/node. No load means one process was running on each node. Two processes/node means that processes were doubled up on each node and, hence, *radix sort's* processes compete with each other for the CPU. And finally, One competitor/node means that each node contains one radix sort process and one spin loop process, which runs forever.

In Figure 5.23(a), performance is shown as normalized time, with Base as 100% and each of +flat, +nonblocking, and smart exch. being normalized to the corresponding Base bar. (Smaller bars are better.) Figure 5.23(b) shows the same data as a factor slower than the Base bar in the No load series. An additional Ideal (under load) series indicates the ideal factor slower under load. Because each radix-sort process is competing for the CPU with either another radix-sort process or an external competitor, it receives half as much CPU time as in the No load case and should therefore, ideally, take twice as long to run. Ideal (under load) is therefore set to 2.0.

Figure 5.23 shows that each form of unresponsiveness-tolerance improves performance. The No load case sees the least performance improvement, as would be expected. The improvement it does see corresponds to tolerating what can be termed “short-term unresponsiveness:” OS services, TLB misses, hardware interrupts, and miscellaneous other occurrences that periodically use a small amount of CPU time. Figure 5.23 indicates that the performance gained by tolerating these is small—at most a few percent. The Two processes/node and One competitor/node cases both gain a little bit of performance when the binary tree communication is replaced with a degenerate tree. The big performance boost comes from my non-



(a) Expressed as normalized time (Base=100%)



(b) Expressed as performance degradation

Figure 5.23: Radix sort performance (polling notification)



blocking barriers: an additional 26–27%. Introducing +smart exh. further improves performance only slightly. The total performance improvement due to unresponsiveness tolerance is 36% for each of the Two processes/node and One competitor/node cases.

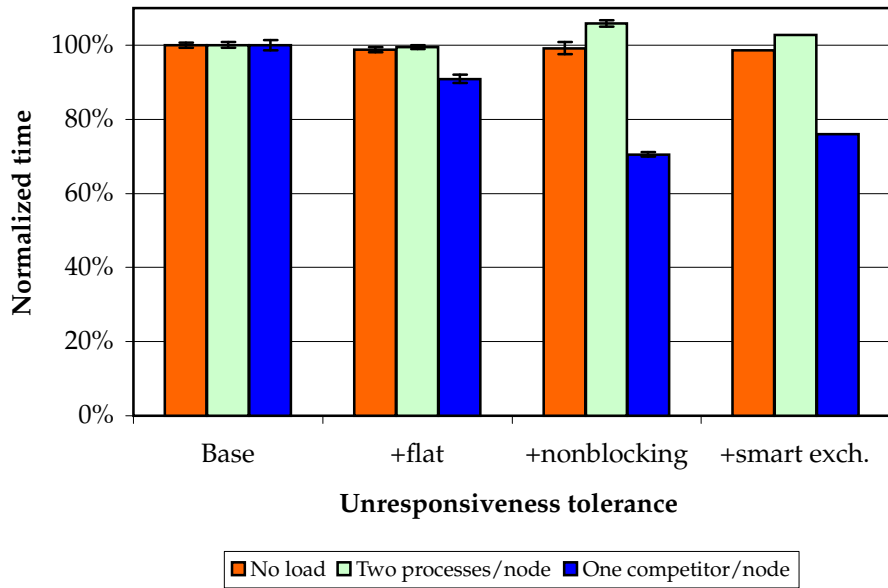
The performance results given in Figure 5.23 strongly support *my thesis*, in that they show that application performance can be noticeably improved if unresponsiveness is tolerated at the endpoints. Nonblocking barriers—this thesis’ contribution in terms of new unresponsiveness-tolerating techniques—generate the largest performance gain. Nevertheless, simpler forms of unresponsiveness tolerance, namely flat communication trees and adaptive message scheduling, do improve performance further, albeit only slightly.

The results are less optimistic when the underlying point-to-point communication blocks, instead of polls, for messages. Figure 5.24 is the blocking-communication analogue of Figure 5.23. Figure 5.24 shows that my nonblocking barriers actually *hurt* performance in the Two processes/node case. The reason for this is unclear, but it must somehow involve the “wrong” process being scheduled on a node and precluding the other process from making progress. The performance loss cannot be due to the extra bookkeeping overhead or to the application’s being overzealous in relinquishing the CPU; if it were, +nonblocking would not improve performance—by a noteworthy 22%—in the One competitor/node case.

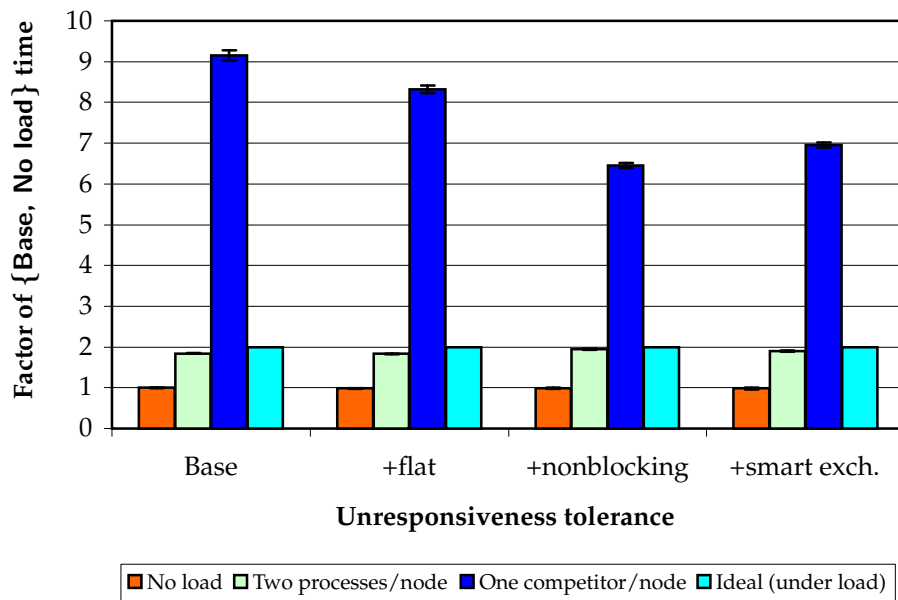
Another pessimistic observation is that +smart exh. hurts performance in the One competitor/node case. However, that is to be expected. +smart exh. assumes that if process *B* receives a message from process *A*, then process *A* must be scheduled. But this is *never* the case when blocking notification is used, because process *A* yields the CPU immediately after sending. Hence, every message arrival triggers a context switch. Note that the increase in time between +flat and +nonblocking in the Two processes/node case is approximately the same as the time needed for these additional context switches. This may be a clue to the poor performance of the {Two processes/node, +nonblocking} case.

Conclusions to draw from Figures 5.23 and 5.24 include the following:

1. Nonblocking barriers do not preclude other unresponsiveness-tolerating techniques; performance gain is additive.
2. Nonblocking barriers improve performance substantially in light of either internal or external CPU contention (with the exception of the Two processes/node case when blocking notification is used).



(a) Expressed as normalized time (Base=100%)



(b) Expressed as performance degradation

Figure 5.24: Radix sort performance (blocking notification)

### 5.4.2 Robustness to cluster scale

We now examine the impact of unresponsiveness and unresponsiveness tolerance as the number of processes in the computation varies. Figure 5.25 shows the result of this experiment. The  $x$  axis is the form of unresponsiveness tolerance that was used. It is either “Intolerant,” which corresponds to Base in the previous graphs, or “Tolerant,” which corresponds to the best of +flat, +nonblocking, and +smart exch.. There are sets of Intolerant plus Tolerant bars for each of the forms of explicit unresponsiveness that we have been using in this section—No load, Two processes/node, and One competitor/node—and one extra bar for the Responsive case. The  $y$  axis is the number of processes that partook in the radix sort: 4, 8, or 16. And the  $z$  axis is the normalized time each experiment took, with Responsive being set to 1.0. Polling point-to-point communication is used for the runs plotted in Figure 5.25(a), and blocking point-to-point communication is used for the runs plotted in Figure 5.25(b).

The important result shown in Figure 5.25 is that unresponsiveness rapidly takes its toll on performance as the number of processes increases. Ideally, the Two processes/node and One competitor/node bars should all be at the 2.0 mark—twice the time a completely unresponsiveness-tolerant *radix sort* should take. In practice, the 4-process run is 5.9 times as slow as Responsive in the {Blocking, One competitor/node } case, when the system is unresponsiveness-intolerant. The 8-process run is roughly twice as slow as that (9.5 times Responsive), and the 16-process run is roughly twice as slow as that (21.4 times Responsive). Unresponsiveness tolerance improves performance substantially in this case. For instance, in the 16-process case, unresponsiveness tolerance reduces the slowdown from 21.4X down to 9.1X.

When polling is used for point-to-point communication (Figure 5.25(a)), both the Two processes/node and One competitor/node cases are observably bad and grow worse as the number of processes increases. Unresponsiveness tolerance significantly improves the situation. With 16 processes, unresponsiveness tolerance decreases *radix sort*'s slowdown from 15.1X down to 6.4X when there is internal contention for the CPU (Two processes/node) and from 12.2X down to 5.2X when there is external contention (One competitor/node).

While Figure 5.25 uses normalized data to show that larger runs imply greater performance loss, the unnormalized data yields an additional important result: When there is external contention for the CPU, *radix sort* runs faster with 8 processes and unresponsiveness tolerance enabled than with 16 processes and no unresponsiveness tolerance (38.4 vs. 64.7 seconds with blocking notification, 18.5 vs. 34.4 seconds with polling notification). This result is noteworthy because it shows that adding processors to a computation is not always the best way to make an application run faster; using the existing resources wiser can yield better performance.

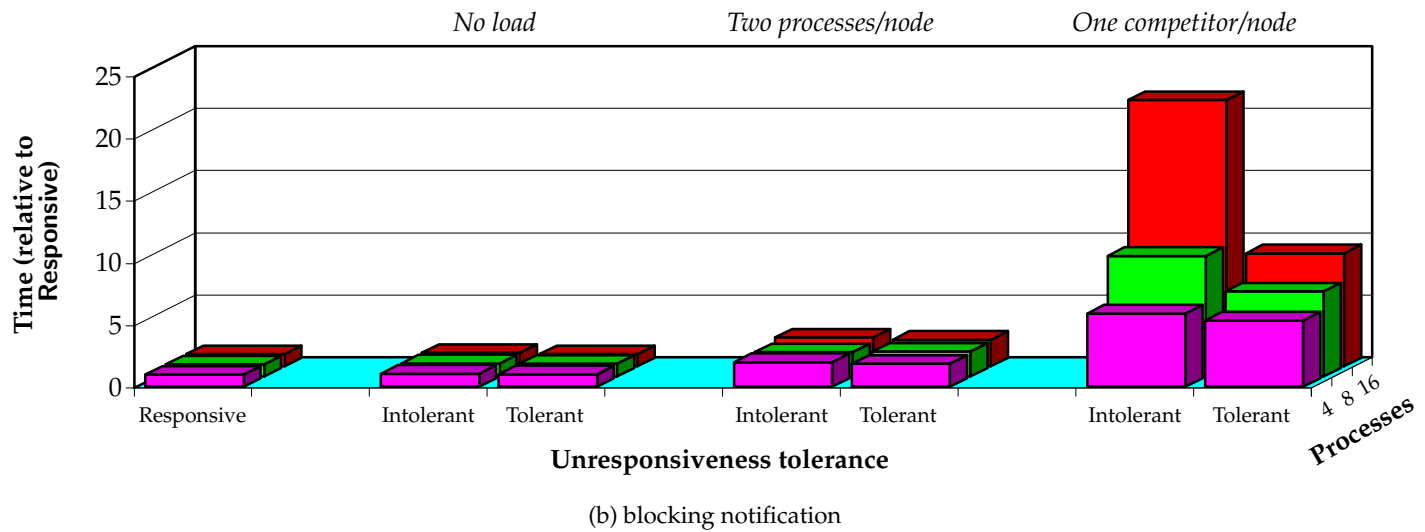
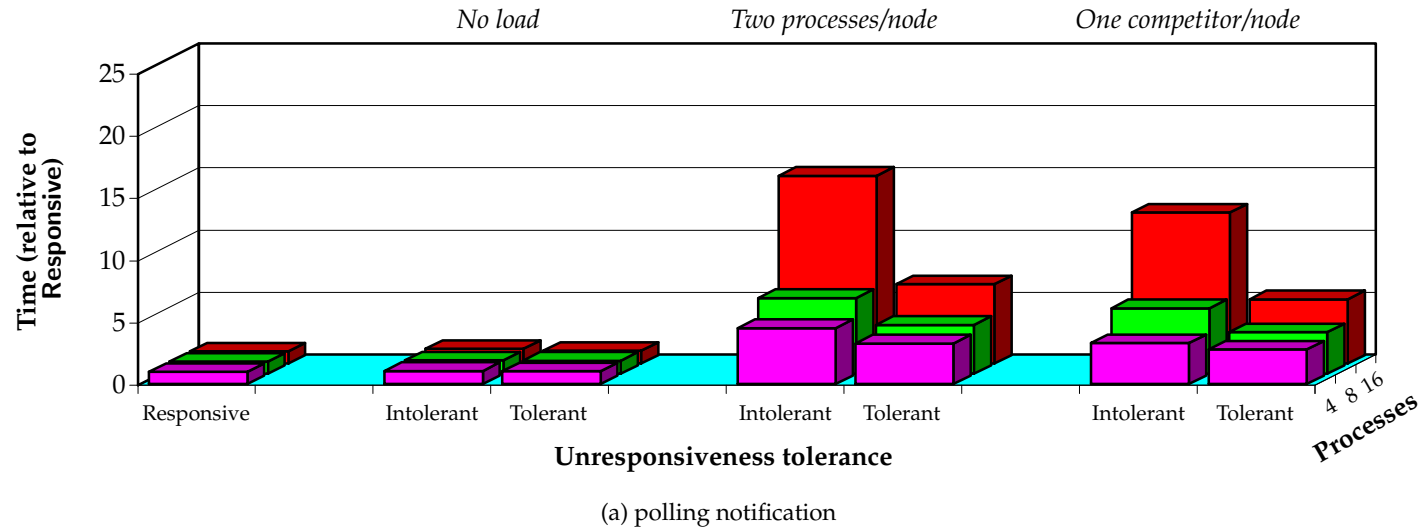


Figure 5.25: Performance improvement in *radix sort* as a function of the number of processes

## 5.5 Comparative performance

Section 4.6.3 discussed coordinated thread scheduling as an alternative or complement to nonblocking barriers. We will now quantify and compare the performance of these two techniques. Specifically, we will compare implicit coscheduling [5] to nonblocking barriers.

The key mechanism behind implicit coscheduling is *spin-block notification*. Instead of simply polling or blocking for message arrival, spin-block dictates that a process should spin (i.e., poll) for some period of time, and block only if no message arrives within that time. Arpaci-Dusseau’s dissertation [5] details how to determine the optimal spin time for various message types (requests, responses, one-way messages, and barrier messages), but the basic idea is to spin for a length of time that is a function of the blocking overhead. For the **machines used in this dissertation**, we determined empirically that 67 spins corresponds to this length of time. Barriers in implicit coscheduling are implemented not as a butterfly network (Figure 4.1 on page 27), but as a reduction followed by a multicast. Flat trees (Figure 5.3 on page 70) are used for both the reduction and the multicast. This enables the root to yield the CPU if any of the leaves is unresponsive and the leaves to yield the CPU if the root is unresponsive. By yielding the CPU when one’s peers are descheduled (as implied by their unresponsiveness) and retaining the CPU when one’s peers are scheduled (and responsive), the hope is that all processes will eventually become coscheduled, thereby eliminating unresponsiveness.

Figure 5.26 shows the performance of implicit coscheduling versus that of nonblocking barriers on an 8-process run of *radix sort*. The **same cluster** was used for all experiments. The  $y$  axis is the absolute running time, in seconds, of *radix sort*. The  $x$  axis is the load on the system: No load, One competitor/node, or Two processes/node. There are eight bars in each set. The two dotted bars represent the base running times (blocking and polling); the two checkered bars represent nonblocking barriers (blocking and polling); and the four solid bars represent four variations of implicit coscheduling. The first spins only once before blocking; it is therefore similar to an always-block scheme. The second spins the empirically determined 67 times. The third spins 10,000 times before blocking; this number was found to yield performance halfway between the always-poll and always-block bars in the One competitor/node case. The fourth variation of implicit coscheduling spins 100,000,000 times before blocking; it is therefore similar to an always-poll scheme. The rationale behind implementing implicit coscheduling using a variety of spin times is to contrast pure polling and pure blocking with a spectrum of intermediate values.

In the No load case, implicit coscheduling marginally outperforms nonblocking barriers, which themselves outperform the base case. However, there is only

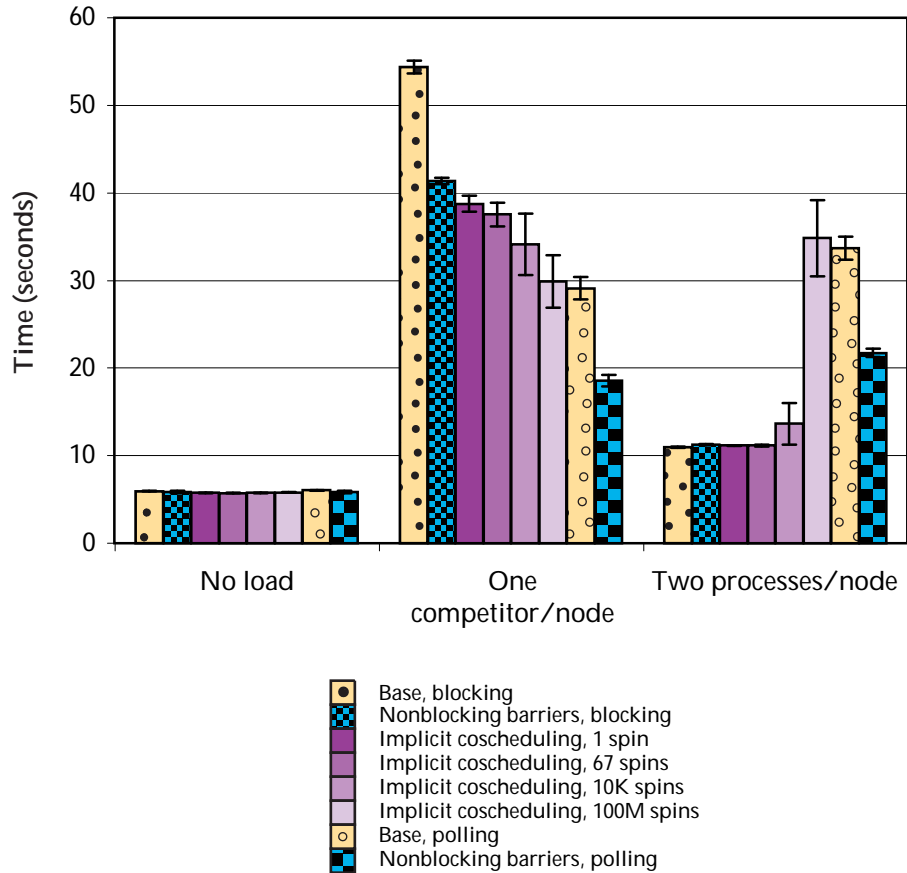


Figure 5.26: Comparison of nonblocking barriers to implicit coscheduling

a 5.6% difference in performance from the fastest configuration (Implicit coscheduling, 67 spins) to the slowest (Base, polling), so this performance gain is less significant than in the cases in which there is contention for the CPU. The One competitor/node bars portray monotonically increasing performance as notification tends from blocking to polling. Nonblocking barriers, polling is the best performer overall, completing *radix sort* in 37.9% less time than the closest implicit-coscheduling alternative, implicit coscheduling, 100M spins. The implicit-coscheduling runs do, however, outperform both always-block runs, Base, blocking and Nonblocking barriers, blocking.

While polling does better than blocking in the One competitor/node case, the reverse is true in the Two processes/node case. The best performer overall is Base, blocking, with Nonblocking barriers, blocking and most of the implicit-coscheduling runs close behind. Nonblocking barriers improve performance substantially over Base, polling and Implicit coscheduling, 100M spins, but still yield less performance than the more polling-oriented runs.

In short, neither nonblocking barriers nor implicit coscheduling is superior to the other in all cases. Both techniques are sensitive to the point-to-point notification mechanism (blocking or polling—or, more generally, the maximum number of spins to perform before blocking) and to the source of unresponsiveness (internal or external CPU contention). In fact, to alleviate implicit coscheduling’s sensitivity to the number of spins, Dusseau et al. propose an adaptive algorithm [36] that dynamically alters the number of spins according to past history. In Figure 5.26, we see that the better of Nonblocking barriers, blocking and Nonblocking barriers, polling never performs significantly worse than the best implicit-coscheduling version, and in the One competitor/node case, nonblocking barriers perform substantially better. We can therefore hypothesize that by augmenting nonblocking barriers with spin-block notification (possibly an adaptive version) as we did with the simple unresponsiveness-tolerating mechanisms in Section 5.4.1, we could further increase application performance.

## 5.6 Discussion

The first important discovery made using the experiments in Chapter 5 is that unresponsiveness degrades performance even on an “idle” cluster. This is due to the various tasks that a COTS operating system performs periodically or sporadically: interrupt handling, paging, and myriad system services that require the CPU from time to time. The preliminary experiments in Section 5.2 showed that barrier time increases with the amount of interbarrier computation and that this increase is strongly correlated with OS context switching. Further experimentation determined that the same process unresponsiveness that causes an increase in barrier time leads to a performance hit of up to 9.9% in *radix sort*. Again, this is on a small, idle cluster. As cluster size increases, the performance hit increases, as well. If one were to build a cluster with as many processors as the ASCI Q supercomputer (12,288) [25], *radix sort* would run up to 122 times slower than would be possible in a more responsive environment. This is a tremendous performance loss and underscores the necessity of unresponsiveness-tolerant collective communication.

If internal or external contention for the CPU is introduced, the performance becomes significantly worse. A radix sort routine running on an ASCI Q-sized cluster with two processes per node (either two *radix sort* processes or a *radix sort* and a competitor) would be expected to run over 10,000 times slower than it could if there were no unresponsiveness in the system.

The experiments presented in this chapter also indicate that neither blocking notification nor polling notification universally outperforms the other. Table 5.9 lists the 16-process Intolerant data from Figure 5.25, showing the factor slowdown when

there is no load on the system, one competitor per node, and two *radix sort* processes per node. Bold text is used to emphasize the worse-performing notification mechanism in each case. As Table 5.9 indicates, polling performs worse than blocking when there is either no load or internal CPU contention, while blocking performs worse than polling when there is a competing process on every node. This is somewhat intuitive, because yielding the CPU in the Two processes/node case enables the application to make further progress, as it is granting one of its runnable processes (if any) access to the CPU; polling does better in the One competitor/node case, because it enables an application to make progress as soon as a message arrives, rather than having to wait for a competitor to finish its time quantum. The conclusion is that a successful unresponsiveness-tolerating technique should improve performance with either notification mechanism.

Table 5.9: Factor of responsive *radix sort* time (16 processes)

| Load                | Blocking    | Polling     | Ideal |
|---------------------|-------------|-------------|-------|
| No load             | 1.1         | <b>1.2</b>  | 1.0   |
| One competitor/node | <b>21.4</b> | 12.2        | 2.0   |
| Two processes/node  | 2.3         | <b>15.1</b> | 2.0   |

Sections 5.3 and 5.4 demonstrated that nonblocking barriers do, in fact, improve performance with either notification mechanism. Section 5.5 took this evaluation one step further and compared nonblocking barriers with implicit coscheduling [5], which uses a spin-block mechanism that juxtaposes polling and blocking notification in an effort to induce coordinated process scheduling across nodes. While both nonblocking barriers and implicit coscheduling improve performance over the unresponsiveness-intolerant baseline, nonblocking barriers outperform implicit coscheduling when there is external CPU contention and perform similarly when there is internal or no CPU contention. With the “wrong” notification mechanism (blocking in the One competitor/node case and polling in the Two processes/node case), nonblocking barriers perform worse than a well-tuned implicit coscheduling implementation (although still better than the baseline). While unproven, we believe that nonblocking barriers could be augmented with implicit coscheduling’s spin-block mechanism to avoid suboptimal notification-mechanism selection without impeding their ability to improve performance by tolerating unresponsiveness.

On an 8-process *radix sort*, nonblocking barriers improve performance by up to 27%, which is a substantial amount, especially given that no source-code modifications were required to achieve that. As the number of processes increases, the performance gain from nonblocking barriers increases, too. Sections 5.3 and 5.4



showed that nonblocking barriers have two more beneficial properties: they can tolerate internal load imbalance, as is present in *prefix scan*, by enabling less-loaded processes to run ahead of more-loaded processes; and the performance gain they yield is additive with regard to other unresponsiveness-tolerating techniques. By using flat trees and smart message exchanges, *radix sort*'s total performance gain increases to 36% on 8 processes.

Finally, the experimental data show that nonblocking barriers, when used alongside flat trees and smart exchanges, can give a greater performance improvement than would doubling the number of nodes but not tolerating unresponsiveness. This is an important result, because it demonstrates that more performance can be achieved by using nonblocking barriers than merely by throwing additional hardware at the problem.

If it can't be expressed in figures, it  
is not science; it is opinion.

*Robert A. Heinlein*

TIME ENOUGH FOR LOVE, 1973

The plural of anecdote is data.

*Ben J. Wattenberg*

VALUES MATTER MOST, 1995

# 6 Related Work

Nonblocking barriers are a novel way to tolerate unresponsiveness on PC clusters. However, there are a number of projects related to it on any of a number of axes. Section 6.1 describes alternatives to my *ad hoc* VIA++ [84] library that also support collective communication and discusses them in the context of unresponsiveness tolerance. Section 6.2 covers collective-communication algorithms and analysis techniques. As Section 4.5.2 argued, nonblocking barriers can be implemented not just in software, but also in hardware or firmware. Hence, Section 6.3 investigates other systems that support collective communication in either hardware or firmware. Section 6.4 differentiates the focus of my performance-centric view of collective communication from the reliability-centric view often taken by distributed-computing protocols. A key feature of nonblocking barriers is that they require no application modifications to be effective. By way of contrast, Section 6.5 discusses a set of projects that require that applications be modified or even completely rewritten to exploit improved collective-communication performance. Finally, Section 6.6 reports the results of several papers that have evaluated collective-communication performance and drawn conclusions about the effects of unresponsiveness on application performance.

## 6.1 Collective-communication libraries

There are a number of messaging libraries that support parallel computations. Many of these support collective-communication operations. MPI [73] is by far the most widely used and supports all of the collective operations listed on page 2 with a vast variety of options and combinations (such as “reduce to all”). In fact, the Cholesky factorization application used in Chapter 5 was written to the MPI interface, as was the original version of the *mg* benchmark.

SLICC [59] specifies an elegant interface for describing participants in a collective operation. While the model relies on a shared address space and library is implemented on a system that supports one-sided communication operations (the Cray T3D [27]), it is unclear whether SLICC can tolerate any unresponsiveness by exploiting one-sided communication. (Tolerating unresponsiveness was certainly not a design goal.)

CCL [7] and ICC [75] are two collective-communication libraries designed to split a large collective operation (in terms of data size, not the number of participants) into multiple smaller collective operations. The goal is to increase message pipelining. While this approach is orthogonal to unresponsiveness-tolerance, it shares with it a common attitude towards collective communication. Specifically, both unresponsiveness-tolerance and message grain-size tuning eschew naive models of collective-communication performance. My research investigates what happens when one breaks the assumption that nodes are responsive before and during a collective-communication operation. CCL and ICC investigate what happens when one breaks the assumption that multiple outgoing messages from a single node can proceed in parallel.<sup>1</sup>

## 6.2 Collective-communication algorithms

As mentioned in Section 6.1, it is a somewhat unrealistic model to assume that multiple messages can be transmitted from a single node in parallel. While CCL and ICC force the collective operation to fit the model (by sending small enough messages so that multiple ones can effectively be transmitted in parallel),  $\lambda$ -trees [8] and  $\alpha$ -trees [10] work under a more realistic model of communication and optimize their communication patterns within that model. Both communication structures assume that collective communication is composed of phases, in each of which, up to one send and up to one receive can occur. The main idea is to try to have all processes finish at approximately the same time by having nodes that start the collective operation sooner do more work. This may help processes avoid becoming unresponsive. One area of future investigation for my research would be to replace the flat trees utilized in Section 5.4.1 with  $\lambda$ -trees or  $\alpha$ -trees. The idea is that  $\lambda$ -trees and  $\alpha$ -trees have better scaling properties than flat trees and may also have better unresponsiveness-tolerance properties than naive binary trees; the “smarter” structures do not introduce load imbalance in the same way that naive trees do, because they try to make all of the processes finish their participation in the collective operation at approximately the same time.

## 6.3 Collective communication in clusters

There are few works that ① address collective communication in a PC cluster environment, ② focus on parallel applications, and ③ permit modifications only to the components described in Section 3.3. The most relevant projects to mine are

---

<sup>1</sup>Note that all of the multicast and barrier figures in this dissertation are drawn as if multiple outgoing messages from a single node can, in fact, proceed in parallel.

LFC [11] (and its predecessor, FM/MC [102]) and MAGPIE [58], both from *Vrije Universiteit*. LFC runs atop Myrinet, and what makes it unique is that it implements multicast—including hooks for totally-ordered multicast—in the Myrinet firmware. While LFC has no unresponsiveness tolerance and supports only multicast, it does serve as a point of reference for my research because it implements a collective operation at a fairly low level of the system. MAGPIE builds upon the LFC work by extending collective communication to the wide area. It uses one set of communication algorithms within a cluster, to maximize parallelism, and a different set between clusters, to minimize diameter. This is similar to my work if one draws an analogy between “local cluster” and “responsive” and between “remote cluster” and “unresponsive,” although in my work, the distinction is dynamic, while in MAGPIE, it is static.

The remaining works that we now describe all place collective communication in the switching fabric (which my thesis does not do). The Memory Channel interconnect [37] supports hardware multicast. Like LFC, Memory Channel has no unresponsiveness tolerance and supports only multicast (which, like LFC, is totally ordered). Memory Channel’s primary drawback from the perspective of my thesis is that it does not scale beyond 16 nodes.<sup>2</sup> Because one of the strengths of a PC cluster is its ease of scaling (in terms of hardware cost) and the virtually unlimited number of nodes that can be connected, I ensured that the approach taken in my thesis does not crimp cluster scalability.

A number of ATM switches, such as *FORE Systems*’ ASX 4000 switch [40], support multicast in hardware. Like VIA, ATM is connection-oriented. However, multicasts are not flow-controlled;<sup>3</sup> data can be dropped in the switch as well as at the endpoints. However, parallel applications generally expect reliable communication, and therefore need some form of flow control to ensure reliability.

In addition to ATM, other industry-standard networks that support collective communication (specifically, multicast) are Ethernet [74] and Fibre Channel [97]. Ethernet was originally based on a shared-bus architecture, which is an inherent broadcast medium. However, this also implies that its performance is non-scalable—aggregate bandwidth stays constant as nodes are added to the network—so it is therefore inapplicable to high-performance PC cluster research. Newer, switched Ethernet behaves more like ATM: more scalable than a bus, but still (semantically) unreliable. Fibre Channel, more beneficially to parallel applications, utilizes link-level flow control in the switching fabric and can even do flow-controlled

---

<sup>2</sup>Compaq’s assumption is that each node is a moderately-large ( $\approx 16$ -processor) SMP, so it is still possible to cluster together a few hundred processors.

<sup>3</sup>Recent ATM switches [40] and certain research ones [14] do support flow control, but the ATM *interface* specifies that communication is unreliable.

multicasts. However, Fibre Channel is purely hardware and doesn't extend all the way up to software messaging layers as does my thesis work. Because nonblocking barriers can span the software/hardware boundary, there is more potential for optimizing performance using nonblocking barriers than is using a switch-level scheme.

While all of the aforementioned works limit their definition of collective communication to multicasts, one of the few networks for PC clusters that supports barriers is the PAPERS [32] family. In fact, early implementations supported *only* barriers—data transfer required a separate network. Because PAPERS's barriers are implemented in hardware, they require host responsiveness only to enter the barrier, just like the hardware implementation of my nonblocking barriers presented in Section 4.5.2. However, the PAPERS project focused primarily on reducing barrier latency, while my research centers around tolerating unresponsiveness.

A number of past and present parallel computers support collective operations—especially barriers and multicasts—in hardware [48, 51, 68, 93]. (One can also argue that cache-coherent shared-memory machines such as the Origin 2000 [64] also effectively support multicasts in the sense that one processor can write data and multiple other processors can read it.) However, these systems all utilize custom NICs integrated further up in the memory hierarchy than the I/O bus. As stated in Section 3.3, for my work to have impact in the space of PC clusters, I cannot necessitate modifications to the workstation architecture, which would essentially be required to integrate the network interface elsewhere in the system.

## 6.4 Wide-area collective communication

My thesis research was performed in the context of parallel computing. However, there has also been much attention paid to efficient collective communication in distributed applications, especially in wide-area systems (as opposed to clusters). The distributed-system view towards collective communication takes a fundamentally different attitude towards the problem than the parallel-computer view. In distributed systems, collective communication is generally limited to multicasts, but with added focus on unregulated operations. That is, while I have focused on situations in which a set of nodes all agree to perform a collective operation, in the distributed-systems literature, the notion is that multiple nodes decide independently to multicast to multiple other nodes. The emphasis in this situation is on portability, fault tolerance, and advanced ordering semantics. For example, global, total ordering of multicast messages from multiple sources is a common goal, while low-latency, low-overhead communication is not. Because the network fabric is the communication bottleneck, there is little need to optimize the endpoints. Hence, the schemes mentioned in the context of distributed systems are generally based on

heavyweight, OS-level messaging. My research assumes that all nodes cooperate and agree on the ordering of accesses to shared state; distributed-system communication layers, such as SRM [38], RMTMP [69], RMP [105], the communication layers described below, and Chang and Maxemchuk’s seminal work on reliable multicast [20] do not.

Wide-area messaging layers such as RAMP [60], LTRC [76], and MTP [4] tolerate unresponsiveness, but do in a very different way from that presented in this dissertation. Rather than tolerate comparatively **short-term unresponsiveness**, they try to adapt to longer-term unresponsiveness, specifically, that due to network congestion as determined by number of lost packets. That is, they detect semi-permanently slow nodes and gradually begin sending data to them at a lessened rate. MTP even ejects particularly slow nodes from the group—an action that would be unacceptable in a parallel computation.

Also note that many distributed-system multicasts rely on hardware multicast (or a hierarchy of multicasts based on IP multicast [31]) for performance. For example, RMP and RAMP achieve good performance only when run over bus-based Ethernet. In contrast, my thesis assumes only point-to-point communication in hardware.

## 6.5 Application/runtime-system techniques

While my research involves modifying system middleware, an alternative I have chosen not to pursue is to modify individual applications to make them load balance their work. Generally, this involves utilizing large numbers of threads and context switching whenever a thread is blocked (e.g., when it is waiting for an unresponsive peer). Naturally, the more unresponsiveness is present in the system, the more threads will be required to tolerate it. An aggressive multithreading + load balancing approach in this category is Cilk [15]. The idea is that the user rewrites—and typically, rethinks—his application in a multithreaded extension of C and links it against the Cilk-NOW runtime system (the version of the Cilk runtime system designed for clusters). When a process has completed all of the tasks in its task queue, Cilk-NOW steals work from another process’ task queue, thereby balancing load and increasing efficiency.

Charm++/Converse [55, 56] is another example of a system with load balancing integrated into the runtime system. The programmer writes fine-grained object-oriented code in Charm++, a distributed, C++-like language in which objects communicate with explicit message passing. The runtime system then exploits global system state, observations of objects’ communication patterns, and application-provided triggers to periodically load balance the application across

processors. Brunner and Kalé [19] apply their load balancing techniques to a barrier microbenchmark, similar to that used in Chapter 5, and improve its performance. In a sense, Charm++/Converse tackles unresponsiveness by *preventing* it, while my thesis is designed to *tolerate* it. In addition, because the programmer explicitly writes message-driven code, more information is made available to the task scheduler, so Charm++/Converse can also handle collective-communication operations other than barriers, e.g., reductions [54]. When using nonblocking barriers, in contrast, a programmer does not need to write message-driven code; nonblocking barriers are implicitly message-driven. Being implicitly message-driven gives nonblocking barriers the advantage that they can deliver improved performance to unmodified programs. The disadvantage is that they can't detect as much concurrency as a programmer might specify explicitly with a system like Charm++/Converse.

While my nonblocking barriers preserve traditional barrier semantics, an alternative would have been to alter these semantics in order to tolerate unresponsiveness. Noncommittal barriers [79] are an example of such an alternative. Although designed to work as a message-fencing operation, noncommittal barriers allow processes to exit the barrier before it is known that all pre-barrier messages have been delivered. If a pre-barrier message does arrive after a process exits the barrier, the application is responsible for rolling itself back to a pre-barrier state, processing all the late messages (which may entail further messaging), and retrying the barrier. The primary differences between noncommittal and nonblocking barriers are the following:

1. Noncommittal barriers are intended to be used as a fencing operation, while nonblocking barriers are intended to be used for ordering messages across application phases.
2. Noncommittal barriers require applications to be modified to save and restore global program state (a generally nontrivial task), while nonblocking barriers require no application modifications whatsoever.

A second approach that alters barrier semantics is the fuzzy barrier [45]. Fuzzy barriers are a split-phase form of barrier. They separate barrier entry from barrier exit and allow programmer-specified work to proceed between the two. (Gupta's implementation of fuzzy barriers [45] flags instructions as being within either "barrier regions" or "non-barrier regions," which is essentially the same as having separate ENTER BARRIER and EXIT BARRIER operations.) There are a few key differences between fuzzy barriers and nonblocking barriers. Fuzzy barriers have different semantics from traditional barriers, while nonblocking barriers preserve traditional barrier semantics. With fuzzy barriers, programmers must rearrange their

programs to place code between the barrier entry and exit points and must consciously consider what code is safe to place there. Nonblocking barriers, in contrast, automate the process. With their message-driven semantics, nonblocking barriers execute only known-safe code until the barrier completes. This is a more conservative approach than fuzzy barriers, but it requires no programmer intervention or code modifications. Finally, code that is placed within a fuzzy barrier's barrier region is determined statically, while nonblocking barriers dynamically execute any code that will not violate barrier semantics.

## 6.6 Evaluating unresponsiveness

There have been a few studies of unresponsiveness that relate to my work. First, in a study that Brewer and Kuszmaul [18] performed on the Thinking Machines CM-5 [100], they found that adding unnecessary barriers to parallel programs sometimes *increases* performance. They attribute this effect to fan-in. While VIA drops packets and (in the case of reliable delivery/reception) introduces channel resets when a receiver is unresponsive, the CM-5 backs up the network, causing additional delays. In other words, on the CM-5, the nodes are responsive, as they do not run a heavyweight operating system, but the network is unresponsiveness, because it has extremely limited buffering available and is therefore easy to clog. Barriers ensure that all of the processes are servicing the network in unison, thereby largely preventing it from backing up. On a PC cluster, in contrast, the nodes are unresponsive but the network has a relatively large amount of buffering (in VIA, the size of host memory). Adding (traditional) barriers therefore does little to help the network, but makes the nodes—and the cluster as a whole—less tolerant of unresponsiveness.

Based on results on both the CM-5 and T3D, Karamcheti and Chien [57] also conclude that fan-in is an important problem in parallel computing. Their solution is to employ “pull messaging,” in which receivers *pull* messages from each sender in turn (using remote reads), as opposed to the more traditional “push messaging.” This ensures that messages are transmitted no faster than the receiver can process them. Pull messaging is relevant to unresponsiveness-tolerant collective communication, because it helps alleviate the problem of fan-in and because it migrates buffer management to the senders, who are responsive at the time of the send operation.

I find that a great part of the information I have was acquired by looking up something and finding something else on the way.

*Franklin P. Adams*



# 7 Conclusions

Having presented and evaluated nonblocking barriers, we now draw some overall conclusions and place my thesis work in its larger context. Section 7.1 summarizes the prior chapters of this dissertation. While the dissertation has so far focused primarily on the strengths of nonblocking barriers, Section 7.2 states some of their weaknesses and on the limitations of my approach. Avenues for future research are presented in Section 7.3. Section 7.4 discusses the results of my research from a broader vantage point. And finally, Section 7.5 enumerates the contributions of my thesis work.

## 7.1 Summary

Over the past 15 years, the high-performance computing community has migrated from vector supercomputers to massively parallel processors to distributed shared-memory machines. Commodity-based clusters of personal computers are the community's latest darling because of their low cost per MIPS and competitive node performance. However, PC clusters are a qualitatively different—and more complex—platform than their predecessors. The key distinction is that traditional “big iron” machines run the user's application in dedicated mode; virtually nothing else occupies a node's memory space or CPU time. In contrast, applications running on a PC cluster must share their resources with a heavyweight, commodity operating system, a menagerie of OS daemons, and, possibly, other users' applications. Furthermore, the PC's deep, complex memory hierarchies make memory access times largely unpredictable and varying over many orders of magnitude. The consequence of the PC cluster's more complex environment is that an application's processes frequently become unresponsive for a length of time—of unpredictable duration.

If all applications were loosely coupled, with processes rarely depending upon data from other processes, this unpredictability would largely be a nonissue. However, many problem domains and individual applications, such as those ported from previous generations of supercomputers, are tightly coupled and use collective-communication operations to coordinate processes and maintain a consistent data state. Chapter 5 demonstrated that endpoint unresponsiveness (i.e., unresponsiveness within the PCs, as opposed to within the network) occurs frequently in PC

clusters, even on an unloaded system. As clusters increase in size, the impact of end-point unresponsiveness on application performance is expected to increase, according to the data presented in this dissertation. Unless something is done about unresponsiveness, applications will make inefficient use of cluster resources, thereby wasting time (with unnecessarily long execution times) or money (with a need for more computers to achieve a given level of performance).

The solution I propose in this dissertation is to tolerate unresponsiveness at the endpoints. My research focuses on collective-communication-intensive applications, because collective communication is widely considered an important problem, and because collective-communication’s higher-level semantics provide more opportunity for optimization than point-to-point communication. I specifically targeted barriers for performance optimization, because barriers are a common synchronization mechanism for parallel applications and because they are highly sensitive to unresponsiveness—a single unresponsive participant delays the entire operation.

As part of my thesis research, I developed a new synchronization primitive: the *nonblocking barrier* (Chapter 4). Nonblocking barriers work just like ordinary barriers, in that they are intended to synchronize a set of processes. However, while a traditional barrier ensures that no process exits the barrier until all processes have entered the barrier, a nonblocking barrier allows processes to exit the barrier before the barrier actually completes. The only restrictions are that ① no message sent after exiting a barrier can be received until all processes have entered the barrier, and that ② there are no hidden channels with which information can bypass the ordinary send/receive mechanisms and thereby cause “anomalous behavior” [61]. The key insight is that in a system with no global state, synchrony is a function of the messages delivered, not a function of absolute time. By delaying only message delivery until the barrier has truly completed, nonblocking barriers enable processes to compute, send messages, synchronize, and deliver messages sent before the barrier—all without altering the traditional barrier semantics presented to the application. Section 5.4.1 shows that an unmodified application kernel can gain a 26–27% performance improvement on eight processes, just by replacing traditional barriers with nonblocking barriers in the communication library. If a few additional unresponsiveness-tolerating mechanisms (flat communication trees and adaptive message scheduling) are implemented, as well, the total performance improvement can grow to 36%. This is a significant improvement, especially considering that it requires no changes to a user’s application. To put a 36% performance improvement in context, it takes about 11½ months for the best numbers reported on the SPEC CFP95 benchmark to increase by 36% (Figure 7.1).<sup>1</sup>

---

<sup>1</sup>The data for Figure 7.1 comes from <http://www.spec.org/cgi-bin/osgresults?conf=cfp95>.

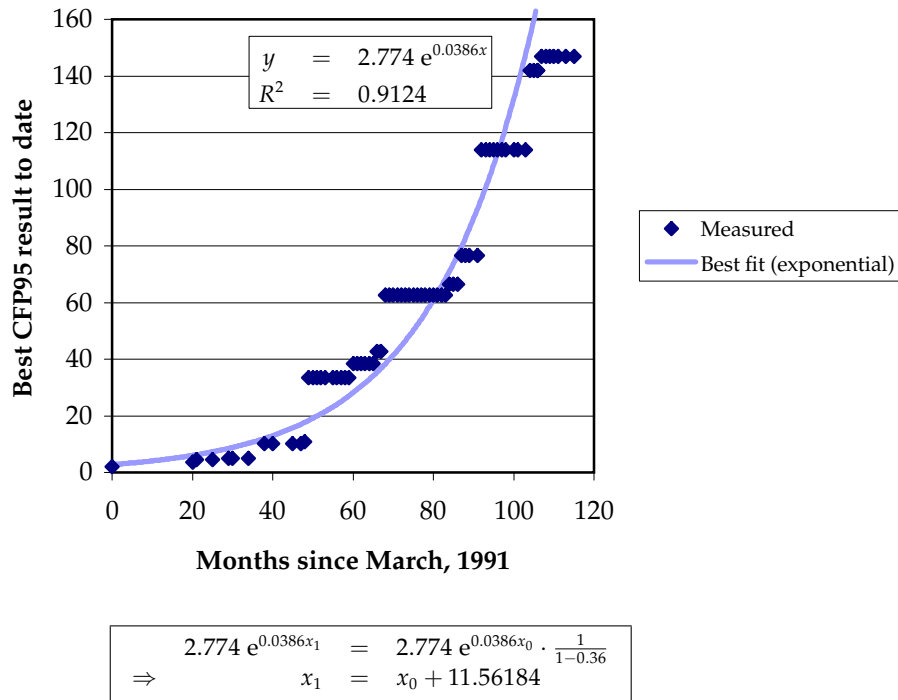


Figure 7.1: Best reported performance on the SPEC CFP95 benchmark over time

## 7.2 Experience gained

While a 36% performance improvement from tolerating unresponsiveness in radix sort with no application modifications is a significant achievement, there were a number of stumbling blocks along the way to reaching it. The important impediments are described below, so future researchers can benefit from the experience I gained in the course of my thesis work.

**Sources of unresponsiveness** I had initially hypothesized that performance lost to small sources of unresponsiveness—OS services, TLB misses, hardware interrupts, etc.—would add up to a significant amount. This would have been an exciting result. While Figure 3.1 on page 18 and the various experiments in Chapter 5 do indicate that there is a non-negligible amount of “no load” unresponsiveness, they also show that orders of magnitude more performance is lost when applications are mischeduled, load-imbalanced, or need to vie for computational resources.

**Intra-operation unresponsiveness tolerance** My initial approach to tolerating unresponsiveness—tolerating unresponsiveness *within* collective-communication operations (Section 4.6.1)—turned out to be a dead end. The problem is that collective-communication operations take orders of magnitude less time than the

major sources of unresponsiveness (microseconds versus tens of milliseconds). Because there is not enough communication work within an individual operation to overlap the idle time, rescheduling communication within an operation is inefficient.

**Applicability** While nonblocking barriers can substantially improve performance, they do so only for a specialized set of applications, as described in Section 7.4.3. It turned out to be more difficult than anticipated to find suitable applications for demonstrating the merit of my approach. Many classes of applications *do* exhibit the previously enumerated “bad” characteristics. For instance, MPI applications often shun collective-communication operations altogether because of the cumbersome MPI collective-communication semantics (e.g., having to create a unique “communicator” for every set of processes that wants to perform a collective-communication operation [73]). In contrast, SIMD and data-parallel applications are rich in their use of collective communication. Nevertheless, those applications are so fine-grained that flow control becomes a serious issue when porting them to a message-passing cluster environment. (Flow control is not needed in SIMD machines, because the processors execute in lockstep and have *a priori* knowledge of what every other processor is doing in every timestep.) Finally, BSP programs usually have a *send...barrier...receive...compute* structure and therefore do not provide a sufficient amount of work that can overlap the barrier’s idle time.

### 7.3 Future work

All of the experiments in Chapter 5 were carried out on a fairly typical PC cluster, composed of commodity PCs and stock high-speed networks (Table 5.1 on page 60). It would be reasonable, however, to try to repeat the same set of experiments on a different platform, in order to demonstrate that the unresponsiveness-tolerating techniques described in this dissertation are robust both to node configuration and to network type and speed. Once that is demonstrated, the next study would logically be to investigate the effect of node/network heterogeneity on performance, both with and without unresponsiveness tolerance. Nonblocking barriers are designed to tolerate short-term unresponsiveness, such as a process being temporarily descheduled; it would be instructive to examine their effect when there is long-term unresponsiveness, such as a few nodes being slower than the rest.

Nonblocking barriers—as well as flat trees and adaptive message scheduling—are compatible with coordinated thread scheduling, load balancing, the application/runtime-system techniques described in Section 6.5, and other forms of unresponsiveness tolerance. However, it is an open question which of those are

complemented by nonblocking barriers, which are hurt, and which are unaffected. An area for future study would be to determine the combination of techniques to use to achieve the greatest increase in performance.

As stated on the previous page, there are a number of reasons why legacy applications may not be able to fully exploit nonblocking barriers. However, future work can involve restructuring legacy applications, writing new applications from scratch, or porting applications from other communication models, such as a release-consistent shared memory model (Section 4.4). Once a set of applications has been assembled, the first additional study would be to repeat the experiments from Sections 5.3 and 5.4 with the new applications. This would quantify the range of performance improvement that can be achieved by tolerating unresponsiveness. The next additional study would be to examine the performance of workloads consisting of various sets of applications. The goal of such a study would be to examine throughput and fairness both with and without unresponsiveness tolerance and in comparison to other unresponsiveness-tolerating techniques, such as coordinated thread scheduling.

Section 5.3.2 showed how nonblocking barriers enable the less-loaded processes in a load-imbalanced kernel (*prefix scan*) to leave the barrier early, so that they can proceed with useful work. However, enabling the less-loaded processes to leave early may exacerbate the load imbalance. Future work could include examining the impact that this has on the rest of a larger computation, to verify that the performance gained from nonblocking barriers does not lead to a greater performance loss elsewhere in the application.

In Section 4.5.2, this dissertation described how one could implement nonblocking barriers in hardware. The natural next step would be to build a simulator for dedicated nonblocking-barrier hardware and to evaluate the performance improvement over the current, software-only implementation. As an intermediate step, one could implement nonblocking barriers in firmware and examine how a firmware implementation compares to the software and hardware implementations in terms of cost, performance, and flexibility. This analysis would lead to an understanding of the costs and tradeoffs involved in tolerating unresponsiveness in various parts of the system. It would also show how much of the overhead due to unresponsiveness-tolerance can be decoupled from a process' execution.

Finally, this dissertation proposed one new mechanism for tolerating unresponsiveness (two, if adaptive message scheduling is included). Future research would be to investigate alternatives, such as the explicit unresponsiveness detection proposed in Section 4.6.2, or additional techniques. A possible approach would be to examine each collective-communication operation in turn and find a way to tolerate unresponsiveness involving that operation. Ideally, these individual techniques

could then be generalized into a single new mechanism for tolerating unresponsiveness.

## 7.4 Perspective

After examining the behavior of parallel programs running on workstation clusters and measuring their performance with and without unresponsiveness-tolerating mechanisms installed, it appears that the basic premise of this dissertation holds true:

- Unresponsiveness is present in workstation clusters.
- Unresponsiveness noticeably hurts performance.
- Unresponsiveness can be tolerated at the endpoints (nodes).
- Tolerating unresponsiveness can significantly improve performance.

### 7.4.1 Problem importance

Although it is convenient to think of a PC cluster as being essentially identical to a parallel supercomputer, this dissertation proposes that there is a key distinction: endpoint unresponsiveness. The same commodity components that simplify the cluster constructor's task also introduce unresponsiveness into the system, detracting from application performance. Empirically, when running an 16-process *radix sort* on an otherwise-idle cluster, unresponsiveness induced by the operating system and node hardware adds 8.4% to the execution time when blocking notification is used and 18.9% when polling notification is used (Section 5.2.2). These are both fairly substantial percentages and indicate that unresponsiveness is indeed a problem for PC clusters.

Of even more importance, the magnitude of the problem increases with cluster size. As Section 5.2.1 argued analytically, an increase in cluster size and/or computation granularity leads to an increase in the likelihood that a barrier will be delayed by an unresponsive peer. The model predicting the fraction of slow barriers (Equation 5.1, page 72) shows that even in a PC cluster running only a single program, unresponsiveness is present and affects the performance of collective-communication operations. The model indicates that the National Center for Supercomputing Applications' 1024-processor cluster could be expected to see at least 99.9% slow barriers, where "slow" means more than one standard deviation slower than in the responsive case. Sections 5.2.1 and 5.4.2 provide measurement data which confirms that the more processes are involved in a cluster application, the greater the penalty

caused by unresponsiveness. On a 16-process run, so much performance is lost to unresponsiveness that, in some circumstances, it is better to use only 8 processes, but tolerate unresponsiveness, than to use 16 processes and not tolerate unresponsiveness (Section 5.4.2). This is an important point, because it indicates that the performance lost to unresponsiveness cannot be regained simply by throwing money and hardware at the problem. Rather, the existing resources need to be used more intelligently—using nonblocking barriers.

## 7.4.2 Advantages of nonblocking barriers

There are a number of advantages to using nonblocking barriers to tolerate unresponsiveness:

- They honor all the constraints presented in Section 3.3. Most notably, they require no modifications to users' applications.
- They can improve performance significantly (Sections 5.3 and 5.4).
- Their resource requirements are such that they can be implemented in either software (Section 4.2) or hardware (Section 4.5.2).
- They are not limited to a particular form of unresponsiveness, such as load imbalance or uncoordinated scheduling. Rather, they can tolerate any of the forms of unresponsiveness listed on page 5.
- They can be used alongside other unresponsiveness-tolerating mechanisms, such as coordinated scheduling, load balancing, or the simple mechanisms described in Section 5.4.1, flat trees and adaptive message scheduling.

While nonblocking barriers improve performance more than either flat communication trees or adaptive message scheduling, each unresponsiveness-tolerating technique does improve performance. Flat trees are the “low-hanging fruit;” they are easy to implement, yet they can give a moderate performance boost, at least on a cluster small enough that the point-to-point communication overhead is dominated by the time lost to unresponsiveness; on a larger cluster, flat trees could be expected to hurt performance. When *radix sort* is run with one competitor on each node, flat trees improve performance over the baseline by 7.4% (polling) to 9.1% (blocking). The other simple mechanism, adaptive message scheduling, exploits the ordering flexibility in an all-to-all communication pattern to tolerate unresponsiveness. It shares the same insight as dynamic coscheduling [95, 96], namely, that message reception is a good indicator that the sender is responsive. When *radix sort* is run with one competitor on each node, adaptive message scheduling improves performance over the baseline by an additional 3.0% (polling notification).

### 7.4.3 Applicability

While Chapter 5 showed that much performance can be regained by tolerating unresponsiveness, the techniques described in this dissertation will not improve performance for every collective-communication-intensive application, but only for a particular subset. Applications that can expect to see the greatest performance gains are those in which multiple collective-communication operations proceed concurrently and those that rely more on fan-out operations, such as multicasts, than fan-in operations, such as reductions (Section 4.4.3). The intuition is that unresponsiveness tolerance is similar in applicability to multithreading; if there is other work to overlap with a blocked thread (respectively, other work to overlap with a blocked collective-communication operation), then multithreading (respectively, unresponsiveness tolerance) should be able to improve performance by increasing throughput. Nonblocking barriers are unlikely to improve performance for the following types of applications:

1. Applications that are too fine-grained and therefore contain implicit message dependencies due to flow control, and
2. Applications that block on message reception immediately after a barrier or other collective-communication operation (e.g., applications with a *send...barrier...receive...compute* structure).

Because of the former restriction, naive ports of SIMD applications are unlikely to see much performance improvement (Section 5.3.1). And because of the latter restriction, applications that use barriers as a fencing operation will be unlikely to see much performance improvement (Section 4.4). Of course, unresponsiveness tolerance will also be ineffective on applications and workloads in which there is little unresponsiveness in the system to begin with, or in which collective communication is not on the critical path.

### 7.4.4 Notification mechanism

One result encountered repeatedly in the course of this dissertation concerns the different behavior of blocking versus polling notification for point-to-point messages. Because VIA supports both mechanisms [24], it is instructive to comment on the tradeoffs in the context of unresponsiveness tolerance. In a system supporting user-level communication, polling takes little time per invocation—usually only the time needed for a few memory reads—but wastes CPU time on each poll. Blocking, in contrast, defers to the operating system, which is a comparatively expensive operation, but uses no CPU time while the process awaits a message. In short,



polling gives the process better response time, while blocking gives the system better throughput.

Traditional wisdom dictates that polling be used when communication is frequent and blocking when communication is infrequent.<sup>2</sup> The problem that occurs when this traditional wisdom is applied to a PC cluster is that it assumes that each user has the good of all users at heart. However, in a PC cluster running multiple processes on each node and relying on collective communication, it would be beneficial for a (selfish) process to block only if it knew *a priori* that another process in the same application would receive the CPU. Otherwise, a competitor will get to run on that node. While that may increase overall system throughput, it will almost certainly decrease the blocking application's response time, because it stalls any collective-communication operations that the blocking application attempts to perform. Hence, the unresponsiveness-tolerating techniques presented in this dissertation, nonblocking barriers plus flat trees and adaptive message scheduling, are ideal for PC clusters and collective communication, because these techniques enable processes to hoard the CPU as much as they are able, by using polling notification, yet tolerate the case in which a peer is not coscheduled, which is normally one of the benefits of blocking notification.

## 7.5 Contributions

The following are the three main contributions of my thesis research:

1. A demonstration that unresponsiveness is a problem in PC clusters,
2. The introduction of nonblocking barriers, a new mechanism for tolerating unresponsiveness, and
3. A characterization and evaluation of nonblocking barriers' performance.

While it is widely known that adding load to a system will decrease performance, my research additionally shows that a cluster running a single application and nothing else still does not perform at peak efficiency. The same features that make PC clusters an attractive computing platform—low cost, availability of commodity components, and interoperability of hardware and software from different vendors—also introduce unresponsiveness and degrade performance from what the cluster would otherwise be capable of observing. When applications rely on collective communication, their performance becomes hypersensitive to unresponsiveness. My research shows that a single competitor per node can introduce enough

---

<sup>2</sup>A number of research projects advocate polling for an initial period of time and then blocking if the polling was unsuccessful [30, 36, 72, 95, e.g.].

unresponsiveness to make a 16 process application run 20 times as slow as it would on a responsive system, versus the optimum for the single-competitor case of only 2 times as slow. As cluster size increases, the performance lost to unresponsiveness increases, as well. With large clusters starting to become commonplace, the first contribution of my thesis work is to bring unresponsiveness in PC clusters to light as an increasingly important problem and one that needs to be dealt with if clusters are to achieve their full potential as a high-performance computing platform.

To reduce the performance lost to unresponsiveness, I created a new mechanism for tolerating unresponsiveness: nonblocking barriers. The key idea behind nonblocking barriers is that they distinguish barrier termination from barrier completion. That is, a process is permitted to exit a barrier before all the participating processes have synchronized. However, to ensure correctness, the process is not allowed to deliver messages sent after the barrier in logical time. The insight is that that restriction maintains the correct sequencing of communication operations in the absence of hidden channels. The benefits of nonblocking barriers are the following:

- They require no modifications to the users' applications.
- They can be implemented in either hardware or software.
- They are not tied to a particular point-to-point notification mechanism (either blocking or polling).
- They can be used in conjunction with other unresponsiveness-tolerating techniques to provide even greater unresponsiveness tolerance.

While nonblocking barriers comprise the second contribution of my thesis work, their evaluation marks the third. By running a variety of experiments, I showed that nonblocking barriers can tolerate unresponsiveness caused by internal contention for the CPU, external contention for the CPU, load imbalance, and even the short-term unresponsiveness caused when operating system services awake and perform brief tasks. I quantified nonblocking-barrier performance when used with each of blocking and polling notification, defining the space in which each outperforms the other and verifying that nonblocking barriers improve performance in both cases. I demonstrated that nonblocking barriers can be used in conjunction with other unresponsiveness-tolerating techniques and complement them to further improve application performance. And I presented experimental data that shows how the performance gain from nonblocking barriers increases with cluster size.

The overall message of my thesis work is that unresponsiveness is the dark side of cluster computing, causing performance loss and reducing system efficiency. To date, unresponsiveness has largely been ignored in the context of clusters, as only recently have organizations begun installing large clusters intended for

high-performance computing. As cluster size increases, so does the performance lost to unresponsiveness. My thesis identifies unresponsiveness as an important, new problem to solve and demonstrates that nonblocking barriers are a viable solution.

Throughout this article we've made imprecise statements and statements that ought to have had all sorts of qualifications and reservations attached to them; and some of our statements may be flatly false. Lack of sufficient information and the need for brevity made it impossible for us to formulate our assertions more precisely or add all the necessary qualifications. And of course in a discussion of this kind one must rely heavily on intuitive judgment, and that can sometimes be wrong. So we don't claim that this article expresses more than a crude approximation to the truth.

*Theodore Kaczynski*

THE UNABOMBER MANIFESTO, 1995

# References

- [1] Aberdeen Group, Inc., Boston, Massachusetts. *Giganet: Building a Scalable Internet Infrastructure with Windows NT and Linux*, 1999. Available from [http://www.giganet.com/technology/whitepapers\\_lookup.asp?id=5](http://www.giganet.com/technology/whitepapers_lookup.asp?id=5).
- [2] Tilak Agerwala, Joanne L. Martin, Jamshed H. Mirza, David C. Sadler, Daniel M. Dias, and Mark Snir. SP2 system architecture. *IBM Systems Journal*, 34(2):152–184, 1995. Available from <http://www.research.ibm.com/journal/sj/agerw/agerw.html>.
- [3] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992. Available from <http://www.cs.berkeley.edu/~brewer/cs262/Scheduler.pdf>.
- [4] S. Armstrong, A. Freier, and K. Marzullo. Multicast transport protocol. RFC 1301, Internet Engineering Task Force, February 1992. Available from <http://www.rfc-editor.org/rfc/rfc1301.txt>.
- [5] Andrea C. Arpaci-Dusseau, David E. Culler, and Alan M. Mainwaring. Scheduling with implicit information in distributed systems. In *Proceedings of the SIGMETRICS '98/PERFORMANCE '98 Joint Conference on the Measurement and Modeling of Computer Systems*, Madison, Wisconsin, June 1998. Available from <http://now.cs.berkeley.edu/Implicit/sig98.ps>.
- [6] David Bailey, Tim Harris, William Saphir, Rob van der Wijngaart, Alex Woo, and Maurice Yarrow. The NAS Parallel Benchmarks 2.0. Technical Report NAS-95-020, Numerical Aerodynamic Simulation Facility, NASA Ames Research Center, December 1995. Available from [http://www.nas.nasa.gov/NAS/NPB/Specs/npb2\\_report.ps](http://www.nas.nasa.gov/NAS/NPB/Specs/npb2_report.ps).
- [7] Vasanth Bala, Jehoshua Bruck, Robert Cypher, Pablo Elustondo, Alex Ho, Ching-Tien Ho, Shlomo Kipnis, and Marc Snir. CCL: A portable and tunable collective communication library for scalable parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 6(1):154–164, February 1995. Available from <http://www.cs.jhu.edu/~cypher/pubs/ccl.ps>.

- [8] Amotz Bar-Noy and Shlomo Kipnis. Designing broadcasting algorithms in the postal model for message-passing systems. *Mathematical Systems Theory*, 27(5):431–452, 1994.
- [9] Ray Barriuso and Allan Knies. *SHMEM User's Guide*. Cray Research, Inc., May 18, 1994.
- [10] Massimo Bernaschi and Giulio Iannello. Collective communication operations: Experimental results vs. theory. *Concurrency: Practice and Experience*, 10(5):359–386, April 1998. Available from <ftp://www.grid.unina.it/pub/Papers/iannello/ps-files/ours-art/cc-exp.ps>.
- [11] Raoul A. F. Bhoedjang, Tim Rühl, and Henri E. Bal. Efficient multicast on myrinet using link-level flow control. In *Proceedings of the 1998 International Conference on Parallel Processing*, Minneapolis, Minnesota, August 1998. Available from [ftp://ftp.cs.vu.nl/pub/amoeba/orca\\_papers/multicast98.ps.gz](ftp://ftp.cs.vu.nl/pub/amoeba/orca_papers/multicast98.ps.gz).
- [12] Kenneth Birman, André Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.
- [13] Kenneth P. Birman, Robert Cooper, and Barry Gleeson. Design alternatives for process group membership and multicast. Technical Report TR91-1257, Cornell University, Computer Science, December 1991.
- [14] T. Blackwell, K. Chan, K. Chang, T. Charuhas, B. Karp, H. T. Kung, D. Lin, R. Morris, M. Seltzer, M. Smith, C. Young, O. Bahgat, M. Chaar, A. Chapman, G. Depelteau, K. Grimble, S. Huang, P. Hung, M. Kemp, I. Mahna, J. McLaughlin, T. Ng, J. Vincent, and J. Watchorn. An experimental flow-controlled multicast ATM switch. In *Proceedings of the First Annual Conference on Telecommunications R&D in Massachusetts*, October 1995. Available from <http://www.eecs.harvard.edu/~rtm/mtc-sw.ps>.
- [15] Robert D. Blumofe and Philip A. Lisiacki. Adaptive and reliable parallel computing on networks of workstations. In *Proceedings of the USENIX 1997 Annual Technical Symposium*, Anaheim, California, January 6–10, 1997. Available from <http://www.cs.utexas.edu/users/rdb/papers/USENIX97/cnow.html> or <ftp://theory.lcs.mit.edu/pub/cilk/USENIX97.ps.gz>.
- [16] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, 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>.
- [17] Lawrence S. Brakmo, Sean W. O’Malley, and Larry L. Peterson. TCP Vegas: New techniques for congestion detection and avoidance. In *Proceedings of the SIGCOMM ’94 Symposium*, pages 24–35, August 1994. Available from <ftp://ftp.cs.arizona.edu/xkernel/Papers/vegas.ps>.
- [18] Eric A. Brewer and Bradley C. Kuszmaul. How to get good performance from the CM-5 data network. In *Proceedings of the International Parallel Processing Symposium*, pages 858–867, Cancun, Mexico, April 1994. Available from <ftp://ftp.lcs.mit.edu/pub/supertech/papers/IPPS94-bandwidth.ps.Z>.
- [19] Robert K. Brunner and Laxmikant V. Kalé. Handling application-induced load imbalance using parallel objects. Technical Report 99-03, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, May 1999. Available from <http://charm.cs.uiuc.edu/papers/AppBalancerSC99.ps>.
- [20] Jo-Mei Chang and N. F. Maxemchuk. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, August 1984.
- [21] Andrew Chien, Scott Pakin, Mario Lauria, Matt Buchanan, Kay Hane, Louis Giannini, and Jane Prusakova. High performance virtual machines (HPVM): Clusters with supercomputing APIs and performance. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, Minneapolis, Minnesota, March 1997. Available from <http://www-csag.ucsd.edu/papers/hpvm-siam97.ps>.
- [22] Andrew A. Chien, Mario Lauria, Rob Pennington, Mike Showerman, Giulio Iannello, Matt Buchanan, Kay Connelly, Louis Giannini, Greg Koenig, Sudha Krishnamurthy, Qian Liu, Scott Pakin, and Geetanjali Sampemane. Design and evaluation of an HPVM-based Windows NT supercomputer. *International Journal of High Performance Computing Applications*, 13(3):201–219, Fall 1999. Special issue on clusters and computational grids for scientific computing. Available from <http://www-csag.ucsd.edu/papers/csag/external/bbfarm.ps>.
- [23] David D. Clark, Van Jacobson, John Romkey, and Howard Salwen. An analysis of TCP processing overhead. *IEEE Communications Magazine*, 27(6):23–29, June 1989.

- [24] Compaq Computer Corp., Intel Corp., and Microsoft Corp. *Virtual Interface Architecture Specification*, December 16, 1997. Available from [http://www.viarch.org/html/Spec/vi\\_specification\\_version\\_10.htm](http://www.viarch.org/html/Spec/vi_specification_version_10.htm).
- [25] Compaq High Performance Technical Computing Group. U.S. DOE selects Compaq to build ASCI Q. *High Performance Technical Computing News*, 17, September/October 2000. Available from [http://www6.compaq.com/hpc/tsn/iss017/hptc\\_iss017\\_fa.html](http://www6.compaq.com/hpc/tsn/iss017/hptc_iss017_fa.html).
- [26] Margaret Corbit. Windows NT-based cluster in full production at CTC. Press release, Cornell Theory Center, Ithaca, New York, March 6, 2000. Available from <http://www.tc.cornell.edu/news/releases/2000/production.asp>.
- [27] Cray Research, Inc., Eagan, MN. *Cray T3D System Architecture Overview*, March 1993.
- [28] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauer, 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 (PPoPP)*, pages 1–12, San Diego, California, May 19–22, 1993. Available from <http://www.cs.berkeley.edu/~culler/papers/logp.ps>.
- [29] Helen Custer. *Inside Windows NT*. Microsoft Press, Redmond, Washington, 1993.
- [30] Stefanos N. Damianakis, Yuqun Chen, and Edward W. Felten. Reducing waiting costs in user-level communication. In *Proceedings of the 11th International Parallel Processing Symposium (IPPS '97)*, pages 381–387, Geneva, Switzerland, April 1–5, 1997. Available from <http://www.cs.princeton.edu/shrimp/Papers/ipps97ULC.ps>.
- [31] Steve Deering. Host extensions for IP multicasting. RFC 1112, Internet Engineering Task Force, August 1989. Available from <ftp://ds.internic.net/rfc/rfc1112.txt>.
- [32] H. G. Dietz, T. M. Chung, and T. I. Mattox. A parallel processing support library based on synchronized aggregate communication. In *Languages and Compilers for Parallel Machines, 8th Annual Workshop (LCPC '95)*, Columbus, Ohio, August 10–12, 1995. Available from <http://garage.ecn.purdue.edu/~papers/LCPC95/paper.html> or <http://garage.ecn.purdue.edu/~papers/LCPC95/standard.ps.Z>. Published in *Lecture Notes in Computer Science*, volume 1033, pages 254–268, 1996.

- [33] Peter Druschel and Larry L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles (SOSP)*, pages 189–202, Asheville, North Carolina, December 1993. ACM SIGOPS, ACM Press. Available from <ftp://ftp.cs.arizona.edu/xkernel/Papers/fbuf.ps>.
- [34] Cezary Dubnicki, Angelos Bilas, Yuqun Chen, Stefanos Damianakis, and Kai Li. VMMC-2: efficient support for reliable, connection-oriented communication. In *Proceedings of Hot Interconnects V*. IEEE, August 1997. Available from <http://www.cs.princeton.edu/shrimp/Papers/hotIC97VMMC2.ps>.
- [35] Dave Dunning, Greg Regnier, Gary McAlpine, Don Cameron, Bill Shubert, Frank Berry, Anne Marie Merritt, Ed Gronke, and Chris Dodd. The Virtual Interface Architecture: A protected, zero copy, user-level interface to networks. *IEEE Micro*, 18(2), March/April 1998.
- [36] Andrea C. Dusseau, Remzi H. Arpaci, and David E. Culler. Effective distributed scheduling of parallel workloads. In *ACM SIGMETRICS '96 Conference on the Measurement and Modeling of Computer Systems*, pages 25–36, 1996. Available from <http://www.cs.berkeley.edu/~dusseau/Papers/sigmetrics96.ps>.
- [37] Marco Fillo and Richard B. Gillett. Architecture and implementation of memory channel 2. *Digital Technical Journal*, 9(1), 1997. Available from <http://www.digital.com/DTJP03/DTJP03HM.HTM> or <http://www.digital.com/DTJP03/DTJP03PF.PDF>.
- [38] Sally Floyd, Van Jacobson, Ching-Gung Liu, Steven McCanne, and Lixia Zhang. A reliable multicast framework for light-weight sessions and application level framing. In *Proceedings of ACM SIGCOMM '95*, Boston, MA, September 1995. Available from <ftp://ftp.ee.lbl.gov/papers/srml.ps.Z>.
- [39] M. J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966.
- [40] FORE Systems, Inc. *ForeRunner ASX-4000*, 7/98 edition, July 1998. Available from <http://www.fore.com/products/swtch/asx4000.html>.
- [41] G. A. Geist and V. S. Sunderam. Network based concurrent computing on the PVM system. *Journal of Concurrency: Practice and Experience*, 4(4):293–311, June 1992. Available from <http://www.netlib.org/ncwn/nbccpvm.ps>.
- [42] Kouros Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering



- in scalable shared-memory multiprocessors. In *Proceedings of the 17th International Symposium on Computer Architecture (ISCA '90)*, pages 15–26, Seattle, Washington, May 1990. Available from <ftp://www-flash.stanford.edu/pub/flash/ISCA90.ps.Z>.
- [43] Karen Green and Lisa Lanspery. IBM and NCSA create world's fastest Linux supercomputers in academia. *Access Magazine*, January 16, 2001. Available from <http://access.ncsa.uiuc.edu/Headlines/01Headlines/010116.IBM.html>.
- [44] William Gropp, Ewing Lusk, and Anthony Skjelum. *Using MPI*. MIT Press, Cambridge, Massachusetts and London, England, third edition, 1994. ISBN 0-262-57104-8.
- [45] Rajiv Gupta. The fuzzy barrier: A mechanism for high speed synchronization of processors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, pages 54–63, Boston, Massachusetts, April 3–6, 1989. ACM Press.
- [46] Danny Hillis. *The Connection Machine*. MIT Press, Cambridge, Massachusetts, 1985.
- [47] W. Daniel Hillis and Guy L. Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, December 1986. Available from <http://www.acm.org/pubs/articles/journals/cacm/1986-29-12/p1170-hillis/p1170-hillis.pdf>.
- [48] Marl Homewood and Moray McLaren. Meiko CS-2 interconnect Elan – Elite design. In *Proceedings of the IEEE Hot Interconnects Symposium*, August 1993.
- [49] Y. Huang, C. C. Huang, and P. K. McKinley. Multicast virtual topologies for collective communication in MPCs and ATM clusters. In *Proceedings of Supercomputing '95*, San Diego, California, December 1995. Available from [http://www.supercomp.org/sc95/proceedings/435\\_HUAN/SC95.PS](http://www.supercomp.org/sc95/proceedings/435_HUAN/SC95.PS) or [http://www.supercomp.org/sc95/proceedings/435\\_HUAN/SC95.PDF](http://www.supercomp.org/sc95/proceedings/435_HUAN/SC95.PDF).
- [50] International Business Machines Corp., Poughkeepsie, New York. *IBM Parallel Environment for AIX, MPL Programming and Subroutine Reference, Version 2, Release 1*, first edition, August 1995. Document Number GC23-3893-00. Available from [http://sp.unige.ch/doc/IBM/PE/pe\\_mpl\\_subref.ps.Z](http://sp.unige.ch/doc/IBM/PE/pe_mpl_subref.ps.Z).
- [51] Hiroaki Ishihata, Takeshi Horie, Satoshi Inano, Toshiyuki Shimizu, and Sadayuki Kato. An architecture of highly parallel computer AP1000. In

- Proceedings of the IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing*, pages 13–16, May 9–10, 1991. Available from [ftp://fcapwide.fujitsu.co.jp/ap1000/english/rim/rim\\_91.ps.Z](ftp://fcapwide.fujitsu.co.jp/ap1000/english/rim/rim_91.ps.Z).
- [52] ISO/IEC. *Information Technology–Programming Languages–Fortran–Part 1: Base Language*, December 1997. ISO/IEC 1539-1:1997.
- [53] Harry F. Jordan. A special purpose architecture for finite element analysis. In G. Jack Lipovski, editor, *Proceedings of the 1978 International Conference on Parallel Processing (ICPP '78)*, pages 263–266. IEEE Computer Society, August 22–25, 1978.
- [54] L. V. Kalé and A. Gursoy. Performance benefits of message driven execution. In *Proceedings of the Intel Supercomputer Users' Group 1993 Annual North America Users' Conference*, St. Louis, Missouri, October 3–6 1993. Available from <http://charm.cs.uiuc.edu/papers/MessageDrivenISUG93.ps>.
- [55] L. V. Kalé and Sanjeev Krishnan. Charm++: Parallel programming with message-driven objects. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.
- [56] Laxmikant V. Kalé, Milind Bhandarkar, Narain Jagathesan, Sanjeev Krishnan, and Joshua M. Yelon. Converse: an interoperable framework for parallel programming. In *Proceedings of the International Parallel Processing Symposium*, pages 212–217, 1996. Available from <http://charm.cs.uiuc.edu/papers/converse-ipp96.ps>.
- [57] Vijay Karamcheti and Andrew A. Chien. A comparison of architectural support for messaging on the TMC CM-5 and the Cray T3D. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA '95)*, pages 298–307, Santa Margherita Ligure, Italy, June 1995. Available from <http://www-csag.ucsd.edu/papers/cm5-t3d-messaging.ps>.
- [58] Thilo Kielmann, Rutger F. H. Hofman, Henri E. Bal, Aske Plaat, and Raoul A. F. Bhoedjang. MAGPIE: MPI's collective communication operations for clustered wide area systems. In *Proceedings of the 1999 Symposium on Principles and Practice of Parallel Programming (PPoPP '99)*, Atlanta, Georgia, May 4–6, 1999. Available from [http://www.cs.vu.nl/~kielmann/magpie\\_ppopp.ps.gz](http://www.cs.vu.nl/~kielmann/magpie_ppopp.ps.gz).
- [59] Allan D. Knies, F. Ray Barriuso, William J. Harrod, and George B. Adams III. SLICC: A low latency interface for collective communication. In *Proceedings of Supercomputing '94*, pages 89–96, Washington, DC, November 1994. Available from <http://www.computer.org/conferen/sc94/knies.ps>.

- [60] Alex Koifman and Stephen Zabele. RAMP: A reliable adaptive multicast protocol. In *Proceedings of the Fifteenth Annual Joint Conference of the IEEE Computer and Communications Societies*, San Francisco, California, March 1996. Available from <http://www.tasc.com/simweb/papers/RAMP/ramp.htm>.
- [61] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
- [62] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.
- [63] Butler W. Lampson. Reliable messages and connection establishment. In Sape Mullender, editor, *Distributed Systems*, chapter 10, pages 251–281. ACM Press, New York, 2nd edition, 1993. ISBN 0-201-62427-3. Available from <http://www.research.microsoft.com/~lampson/47-ReliableMessages/Acrobat.pdf>.
- [64] Jim Laudon and Dan Lenoski. The SGI Origin: A ccNUMA highly scalable server. In *Proceedings of the 24th International Symposium on Computer Architecture (ISCA '97)*, Denver, Colorado, June 2–4, 1997. Available from <http://www.sgi.com/Technology/Comcon/isca.pdf>.
- [65] Antoine Le Hyaric. Converting the NAS benchmarks from MPI to BSP. In *High Performance Computing and Networking (HPCN '96) [BSP Birds of a Feather session]*, Brussels, Belgium, April 17, 1996. Available from <ftp://ftp.comlab.ox.ac.uk/pub/Packages/BSP/NASfromMPIToBSP.tar>.
- [66] Samuel J. Leffler, Robert S. Fabry, and William N. Joy. A 4.2bsd interprocess communication primer. Technical Report CSD-83-145, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, Berkeley, California, July 27, 1983. Available from <http://sunsite.berkeley.edu/Dienst/UI/2.0/Describe/ncstr1.ucb/CSD-83-145>.
- [67] Ulana Legedza and William E. Weihl. Reducing synchronization overhead in parallel simulation. In *Proceedings of the Tenth ACM/IEEE/SCS Workshop on Parallel and Distributed Simulation (PADS '96)*, pages 86–95, Philadelphia, Pennsylvania, May 22–24, 1996. Available from <http://www.acm.org/pubs/articles/proceedings/simulation/238788/p86-leggedza/p86-leggedza.pdf>.
- [68] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, W. Daniel Hillis, Bradley C.

- Kuszmaul, Margaret A. St. Pierre, David S. Wells, Monica C. Wong, Shaw-Wen Yang, and Robert Zak. The network architecture of the Connection Machine CM-5. *Journal of Parallel and Distributed Computing*, 33(2):145–158, March 15, 1996. Available from <ftp://theory.lcs.mit.edu/pub/people/bradley/jpdc96.ps.Z>.
- [69] John C. Lin and Sanjoy Paul. RMTP: A reliable multicast transport protocol. In *Proceedings of IEEE INFOCOM '96*, pages 1414–1424, 1996. Available from <http://www.bell-labs.com/user/sanjoy/rmtp.ps>.
- [70] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, July 1973.
- [71] Alan M. Mainwaring and David E. Culler. Active Messages: Organization and applications programming interface. Technical report, Computer Science Division, University of California at Berkeley, 1995. Available from <http://now.cs.berkeley.edu/Papers/Papers/am-spec.ps>.
- [72] Olivier Maquelin, Guang R. Gao, Herbert H. J. Hum, Kevin B. Theobald, and Xin-Min Tian. Polling Watchdog: Combining polling and interrupts for efficient message handling. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture (ISCA '96)*, pages 179–188, Philadelphia, Pennsylvania, May 22–24, 1996. Available from <ftp://ftp-acaps.cs.mcgill.ca/pub/doc/papers/ISCA23.ps.gz>.
- [73] Message Passing Interface Forum. *MPI: A Message Passing Interface Standard*, June 12 1995. Version 1.1. Available from <http://www.mpi-forum.org/docs/mpi-11.ps.Z>.
- [74] Robert M. Metcalfe and David R. Boggs. Ethernet: Distributed packet switching for local computer networks. *Communications of the Association for Computing Machinery*, 19(7):395–404, July 1976. Available from <http://www.acm.org/classics/apr96/>.
- [75] Prasenjit Mitra, David G. Payne, Lance Shuler, Robert van de Geijn, and Jerrell Watts. Fast collective communication libraries, please. In *Proceedings of the Intel Supercomputer User's Group Meeting*, 1995. Available from [http://www.cs.utexas.edu/users/rvdg/papers/icc-vs\\_other.ps](http://www.cs.utexas.edu/users/rvdg/papers/icc-vs_other.ps).
- [76] Todd Montgomery. A loss tolerant rate controller for reliable multicast. Technical Report NASA-IVV-97-011, NASA Ames IV&V facility (Fair-

- mont, WV), August 1997. Available from <http://www.cs.wvu.edu/~tmont/ltrc-doc.ps.gz>.
- [77] David Mosberger. Memory consistency models. Technical Report TR 93/11, Department of Computer Science, University of Arizona, 1993. Available from <ftp://ftp.cs.arizona.edu/reports/1993/TR93-11.ps>.
- [78] NCSA Communications Group Staff. High-performance supercomputing at mail-order prices. *Access*, April 22, 1998. Available from <http://access.ncsa.uiuc.edu/CoverStories/SuperCluster/super.html>.
- [79] David M. Nicol. Noncommittal barrier synchronization. *Parallel Computing*, 21:529–549, 1995. Available from <http://www.cs.dartmouth.edu/~nicol/barrier/barrier.ps>.
- [80] Jaroslaw Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global Arrays: A portable “shared-memory” programming model for distributed memory computers. In *Supercomputing '94*, 1994. Available from <http://www.computer.org/conferen/sc94/nieploch.html>.
- [81] Yu. Ofman. On the algorithmic complexity of discrete functions. *Soviet Physics Doklady*, 7(7):589–591, January 1963. Translated from *Doklady Akademii Nauk SSSR*, 145(1):48–51, July 1962.
- [82] John K. Ousterhout. Scheduling techniques for concurrent systems. In *Proceedings of the 3rd International Conference on Distributed Computing Systems*, pages 22–30, October 1982.
- [83] Özalp Babaoglu and Keith Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In Sape Mullender, editor, *Distributed Systems*, chapter 4, pages 55–96. ACM Press, New York, 2nd edition, 1993. ISBN 0-201-62427-3.
- [84] Scott Pakin. VIA++. Local user’s manual for the VIA++ class library., December 1999.
- [85] Scott Pakin, Vijay Karamcheti, and Andrew A. Chien. Fast Messages: Efficient, portable communication for workstation clusters and MPPs. *IEEE Concurrency*, 5(2):60–73, April-June 1997. Available from <http://www-csag.ucsd.edu/papers/fm-pdt.ps>.
- [86] Steven G. Parker and Christopher R. Johnson. SCIRun: A scientific programming environment for computational steering. In *Proceedings of the 1995*

- ACM/IEEE Supercomputing Conference*, San Diego, California, December 3–8, 1995. Available from [http://www.supercomp.org/sc95/proceedings/499\\_SPAR/SC95.HTM](http://www.supercomp.org/sc95/proceedings/499_SPAR/SC95.HTM).
- [87] Craig Partridge. A faster UDP. *IEEE/ACM Transactions on Networking*, 1(4):429–440, August 1993.
- [88] Jon Postel. User datagram protocol. RFC 768, Internet Engineering Task Force, August 28, 1980. Available from <http://www.rfc-editor.org/rfc/rfc768.txt>.
- [89] Loic Prylli and Bernard Tourancheau. BIP: a new protocol designed for high performance networking on Myrinet. In *Proceedings of the PC-NOW '98 International Workshop on Personal Computer based Networks Of Workstations*, Orlando, Florida, April 1998. Available from <http://www-bip.univ-lyon1.fr/PUBLICATIONS/pub/PT97pcnow.ps.gz>.
- [90] Daniel A. Reed, Christopher L. Elford, Tara M. Madhyastha, Evgenia Smirni, and Stephen E. Lamm. The next frontier: Interactive and closed loop performance steering. In *Proceedings of the 1996 ICPP Workshop on Challenges for Parallel Processing*, pages 20–31, Bloomington, Illinois, August 12, 1996. Available from <http://www-pablo.cs.uiuc.edu/Publications/Papers/ICPP96.ps.gz>.
- [91] David A. Rusling. The Linux kernel. Version 0.8-2. Available from <http://sunsite.unc.edu/LDP/LDP/tlk/tlk.html> or <ftp://sunsite.unc.edu/pub/Linux/docs/linux-doc-project/linux-kernel/>, April 1997.
- [92] Michael Schneider. Phase one of the Pittsburgh Supercomputing Center terascale system is operational. News release, Pittsburgh Supercomputing Center, Pittsburgh, Pennsylvania, January 29, 2001. Available from <http://www.psc.edu/publicinfo/news/2001/tcs-01-29-01.html>.
- [93] Steven L. Scott. Synchronization and communication in the T3E multiprocessor. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 26–36, Cambridge, Massachusetts, October 2–4 1996. Available from [http://reality.sgi.com/sls\\_craypark/Papers/asplos96.html](http://reality.sgi.com/sls_craypark/Papers/asplos96.html).
- [94] Tom Shanley and Don Anderson. *PCI System Architecture*. Addison-Wesley, 3rd edition, 1995. ISBN 0-201-40993-3.
- [95] Patrick Sobalvarro, Scott Pakin, Andrew Chien, and William Weihl. Dynamic coscheduling on workstation clusters. In *12th Annual International Parallel Processing Symposium & 9th Symposium on Parallel*

- and Distributed Processing (IPPS/SPDP), 4th Workshop on Job Scheduling Strategies for Parallel Processing*, Orlando, Florida, March 1998. Available from <http://www.research.digital.com/SRC/scheduling/papers/pgs/nfmdcs.ps>. Published in *Lecture Notes in Computer Science*, vol. 1459, pp. 231–256. Springer-Verlag. ISBN 3-540-64825-9. Available from <http://link.springer.de/link/service/series/0558/papers/1459/14590231.pdf>.
- [96] Patrick Gregory Sobalvarro. *Demand-based Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessors*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, January 1997. Available from <http://www.psg.lcs.mit.edu/~pgs/papers/thesis.ps>. Also appears as MIT Laboratory for Computer Science technical report MIT-LCS-TR-710, available from [http://www.lcs.mit.edu/doc\\_repository/MIT-LCS-TR-710.ps](http://www.lcs.mit.edu/doc_repository/MIT-LCS-TR-710.ps).
- [97] Gary R. Stephens and Jan V. Dedek. *Fibre Channel: The Basics*. AN-COT Corporation, Menlo Park, California, second edition, February 1997. ISBN 0-9637439-3-Z.
- [98] Sun Microsystems, Inc., Palo Alto, California. *Better by Design—The Solaris™ Operating Environment*, 1998. Available from <http://www.sun.com/software/white-papers/wp-solaris7>.
- [99] Hiroshi Tezuka, Atsushi Hori, Yutaka Ishikawa, and Mitsuhsa Sato. PM: An operating system coordinated high performance communication library. In Peter Sloot and Bob Hertzberger, editors, *High Performance Computing and Networking*, volume 1225 of *Lecture Notes in Computer Science*, pages 708–717. Springer-Verlag, April 1997. Available from <http://www.rwcp.or.jp/lab/pdslab/papers/tezuka-hpcn97.ps.gz>.
- [100] Thinking Machines Corporation, Cambridge, MA. *The Connection Machine CM-5 Technical Summary*, October 1991.
- [101] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990. Available from <http://www.acm.org/pubs/articles/journals/cacm/1990-33-8/p103-valiant/p103-valiant.pdf>.
- [102] Kees Verstoep, Koen Langendoen, and Henri Bal. Efficient reliable multicast on Myrinet. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, volume III, pages 156–165, Bloomingdale, IL, August 1996. Available from [ftp://ftp.cs.vu.nl/pub/amoeba/orca\\_papers/icpp96.ps.Z](ftp://ftp.cs.vu.nl/pub/amoeba/orca_papers/icpp96.ps.Z).

- [103] Werner Vogels, Dan Dumitriu, Ashutosh Agrawal, Teck Chia, and Katherine Guo. Scalability of the Microsoft Cluster Service. In *Proceedings of the Second Usenix Windows NT Symposium*, Seattle, Washington, August 1998. Available from <http://www.cs.cornell.edu/rdc/mscs/nt98>.
- [104] Matt Welsh, Anindya Basu, and Thorsten von Eicken. Incorporating memory management into user-level network interfaces. In *Hot Interconnects V*, Stanford, California, August 1997. Available from <http://www.cs.cornell.edu/U-Net/papers/hoti97.ps>.
- [105] Brian Whetten, Todd Montgomery, and Simon Kaplan. A high performance totally ordered multicast protocol. In *Lecture Notes in Computer Science*, volume 938. Springer-Verlag, March 1995. Available from [http://research.ivv.nasa.gov/RMP/Docs/RMP\\_dagstuhl.ps](http://research.ivv.nasa.gov/RMP/Docs/RMP_dagstuhl.ps).

If you steal from one author, it's  
plagiarism. If you steal from many,  
it's research.

*Wilson Mizner*



# Colophon

This dissertation was typeset with  $\text{\LaTeX}2_{\epsilon}$  using the book class and the following packages:

|             |           |             |           |            |
|-------------|-----------|-------------|-----------|------------|
| algorithm   | bytefield | hanging     | multicol  | thumbpdf   |
| algorithmic | calc      | hyperref    | multirow  | topcapt    |
| alltt       | color     | hyphenat    | nameref   | trig       |
| amsbsy      | comment   | ifthen      | pifont    | uiucthesis |
| amsgen      | epigraph  | keyval      | rotating  | url        |
| amsmath     | everyshi  | latexsym    | setspace  | varioref   |
| amsopn      | fancyvrb  | listliketab | snapshot  | xspace     |
| amstext     | float     | longtable   | subfigure |            |
| array       | graphics  | ltxtable    | tabularx  |            |
| booktabs    | graphicx  | mathpazo    | textcomp  |            |

The body typeface is Palatino (actually, URW Palladio L Roman). The primary fixed-width typeface is Computer Modern Teletype 10pt. (CMTT10). Both are typeset at 11pt.

A printed work which cannot be read becomes a product without a purpose.

*Emil Ruder*

# Vita

Scott Pakin was born in Lake Forest, Illinois on May 17, 1970. He received a B.S. degree in Mathematics/Computer Science from Carnegie Mellon University in 1992, an M.S. degree in Computer Science from the University of Illinois at Urbana-Champaign in 1995, and a Ph.D. in Computer Science from the University of Illinois at Urbana-Champaign in 2001.

Since 1993, Scott has worked in the Concurrent Systems Architecture Group at the University of Illinois at Urbana-Champaign under the supervision of Prof. Andrew A. Chien. Scott also had an internship at Intel during the summer of 1995. He was the recipient of the W. J. Poppelbaum Award in 1997 for being the Department of Computer Science's most outstanding graduate student in the areas of hardware and architecture; he was sponsored by an Intel Foundation Graduate Fellowship for the 1998–1999 school year; and he was inducted into the Honor Society of Phi Kappa Phi in 1999.

Scott's research interests include system architecture, high-speed communication, and parallel and distributed computing.

Plaudite, amici, comedia finita est.  
(Applaud, friends, the comedy is  
over.)

*Ludwig von Beethoven*  
FINAL WORDS, 1827