The Performance Evaluation of Workstation Clusters

A thesis submitted for the degree of
PhD in Computer Science

by
Panagiotis Melas

Department of Electronics and Computer Science
University of Southampton

May 2000

University of Southampton

ABSTRACT

Faculty of Engineering

Electronics and Computer Science

Doctor of Philosophy

The Performance Evaluation of Workstation Clusters

by Panagiotis Melas

The requirements for High Performance Computing (HPC) have increased dramatically over the years. Parallelism is the only key technology today which can deliver the required computing performance for very large scale scientific and commercial applications, although its implementation in practice has proved to be a far more difficult task than originally envisaged.

Advances in microprocessor and networking technologies in conjunction with the development and standardisation of the message-passing model and widespread availability of distributed software have enabled workstation clusters to have the potential for HPC at an attractive price-performance ratio. This combination of technologies provides several advantages for clusters but at the same time their evolution and performance is determined and limited by technologies designed for other systems. As a result clusters often fail to deliver at the application level their underlying potential performance.

This thesis investigates the key components of commodity workstation clusters and evaluates the performance of these systems as an integrated HPC platform. It demonstrates the need for a new performance evaluation tool, and proposes the Specific Cluster Operation and Performance Evaluation (SCOPE) benchmark set which has been especially designed to evaluate the performance behaviour of cluster characteristics and promote the workstation cluster concept by assisting commodity workstation cluster designers to understand and analyse the performance behaviour of these systems.

An initial implementation of the SCOPE benchmark suite has been developed and run on a wide variety of workstation clusters and MPP platforms. Results from the SCOPE tests have demonstrated the potential to identify and classify the performance evaluation of workstation clusters. Moreover the SCOPE evaluation tool methodology can be extended to provide support for the development of parallel applications and algorithms tailored to a specific parallel platform.

Contents

1 Introduction 1

1.1 The Requirement for High Performance Computing 1

1.2 High Performance Computing and Parallelism 2

1.3 Workstation Clusters as an Alternative Platform for HPC 4

1.4 Performance Evaluation of Workstation Clusters 6

1.5 Summary 7

2 Low-level Internode Communication 9

2.1 Introduction 9

2.2 Interconnection Issues 9

2.3 Communication Software Layers 10

2.3.1 The TCP/IP Stack 11

2.4 Analysing Communication Overhead 12

2.4.1 Optimising the Communication Processing Overhead 15

2.4.2 High-speed Interconnection Networks 16

2.5 Case Study: Internetworking with Ethernet 21

2.6 User-space Protocols 23

2.6.1 "Careful" Protocols 27

2.6.2 Light Weight Protocols 27

2.6.3 Semantics of User-Space Network Protocols 29

2.7 The BIP Zero-Copy Protocol Approach 31

2.7.1 Future Trends of Network Subsystems 33

2.8 Summary 34

3 Clusters of Workstations 36

3.1 Introduction 36

3.2 Basic Distributed Computing Primitives and Concepts 37

3.2.1 High Level Communication Primitives and Concepts 38

3.2.2 Client-Server Model 39

3.2.3 Remote Procedure Call 39

3.2.4 Coordination, Synchronisation, Concurrency control 41

3.2.5 Distributed File System Concepts 42

3.3 Multicomputers . . . 42
3.4 Clusters as Parallel Computing Platforms . . . 43
3.4.1 Cluster Hardware Aspects and Structures . . . 44
3.4.2 Communication Requirements . . . 47
3.5 The ASCI Project . . . 48
3.6 The Beowulf Class Cluster Computers . . . 48
3.7 The NOW project in Berkeley . . . 49
3.8 The Clusters of the University Campus . . . 50
3.9 Summary . . . 52

4 Message-Passing and MPI **54**
4.1 Introduction . . . 54
4.2 The Parallel Virtual Machine . . . 55
4.3 Bulk Synchronous Parallelism . . . 55
4.4 The Message Passing Interface, Concepts and Semantics . . . 57
4.4.1 MPI Procedures and Semantics . . . 58
4.5 MPICH . . . 59
4.5.1 The Architecture of MPICH . . . 60
4.6 Case Study: Matrix Multiplication . . . 61
4.7 MPI-2 and Parallel I/O . . . 61
4.7.1 Dynamic Processes . . . 62
4.7.2 Single-Sided Communications . . . 63
4.7.3 MPI-2 and Parallel I/O . . . 63
4.8 Summary . . . 64

5 Benchmarks **65**
5.1 Introduction . . . 65
5.2 The Requirement for Benchmarks . . . 65
5.3 Benchmark Objectives . . . 66
5.4 Typical Benchmarking Metrics . . . 68
5.5 Existing Benchmarks . . . 70
5.5.1 The Livermore Loops . . . 70
5.5.2 The LINPACK Benchmark. . . . 71
5.5.3 The PARKBENCH Benchmark . . . 71
5.5.4 The NAS Benchmark . . . 72
5.5.5 The EuroBen Benchmark . . . 73
5.5.6 The Perfect Club Benchmark . . . 74
5.5.7 The SPEC Benchmark . . . 74
5.5.8 The LogP Model . . . 75
5.5.9 Other Benchmarks . . . 76
5.6 Comparison and Assessments . . . 76
5.7 Shortcomings of Existing Benchmarks . . . 77

5.8 Summary 78

6 SCOPE: a Tailored Benchmark Suite 80

6.1 The Requirement for a Tailored Release 80

6.1.1 SCOPE Requirements and Objectives 82

6.2 The Structure of the SCOPE Benchmark 83

6.3 The SCOPE Benchmark Methodology 84

6.3.1 Benchmark Specification 84

6.3.2 Performance Metrics 85

6.3.3 Software Requirements of SCOPE 86

6.3.4 Implementation Rules and Optimisation 86

6.3.5 Time Measurement and Considerations 87

6.4 SCOPE Single Node Tests 89

6.5 SCOPE Low-level Tests 90

6.5.1 The Underlying Network-level Performance Tests 91

6.5.2 Low-level Communication Library Tests 92

6.5.3 Peer-to-Peer Tests 92

6.5.4 Collective Calls Test 94

6.6 SCOPE Kernel-level Tests 98

6.6.1 Kernel-level Message Passing Operation Tests 99

6.6.2 Kernel-level Broadcast Test 100

6.6.3 Kernel-level Scatter/Gather Operations 100

6.6.4 Kernel-level Shift Operation Test 101

6.7 SCOPE Kernel-level Algorithmic Tests 101

6.7.1 Matrix-matrix Benchmarks 102

6.7.2 Row/Column Striped Algorithm Test 102

6.7.3 Cannon's Algorithm Test 103

6.7.4 Sorting Routine Test 103

6.7.5 Multi-grid Relaxation Routine Test 104

6.8 Summary 105

7 SCOPE: Experimental Results and Analysis 106

7.1 Tests on MPPs 107

7.2 Low-level Communication Tests Results 108

7.2.1 TCP/IP and Berkeley Sockets Interface Tests 108

7.2.2 The Linux Cluster 109

7.2.3 The NT cluster 111

7.2.4 The SPARC cluster 111

7.2.5 The SGI Cluster 113

7.2.6 The BIP Cluster 115

7.2.7 Analysis of Peer-to-Peer Test Results 117

7.3 Low-level Collective Call Tests Results 118

7.3.1 Collective Call Tests on the SPARC Cluster . . . 119
7.3.2 Collective Call Tests on the SGI Cluster . . . 121
7.3.3 Collective Call Tests on the BIP Cluster . . . 123
7.3.4 Analysis of Collective Call Test Results . . . 127
7.4 Kernel-level Tests . . . 128
7.5 Kernel-level Operation Tests Results . . . 129
7.5.1 The Broadcast Operation Test . . . 129
7.5.2 The Scatter/Gather Operation Tests . . . 131
7.5.3 The Shift Operation Test . . . 133
7.5.4 Analysis of Kernel-level Operation Tests . . . 134
7.6 Kernel-level Algorithmic Tests Results . . . 135
7.6.1 Matrix-matrix Benchmark Results . . . 135
7.6.2 Sorting Algorithm Results . . . 137
7.6.3 Multi-grid Relaxation Test Results . . . 139
7.6.4 Analysis of Kernel-level Algorithmic Tests . . . 139
7.7 SCOPE Overview . . . 142
7.8 Summary . . . 143

8 Future Work **144**
8.1 Standard Module and Baseline Tests . . . 144
8.2 Results: Analysis and Presentation . . . 145
8.3 Advanced and Experimental Module Tests . . . 146
8.4 Benchmark Expansion and Future Tests . . . 146
8.5 SCOPE and Other Benchmarks . . . 147
8.6 Other Issues . . . 148

9 Conclusions **149**
9.1 The Requirement for HPC and Workstation Clusters . . . 149
9.2 Evaluation of Workstation Clusters with SCOPE . . . 150

A The 802.3 MAC Sublayer **171**

B Processor Timing Mechanisms **174**
B.1 Pentium Time Stamp Counter . . . 174
B.2 DEC Alpha timer . . . 174
B.3 Ultra-SPARC TICK Register . . . 175
B.4 RS6000 Tick Register . . . 175
B.5 CPU Speed . . . 175

C Case Study: Matrix Multiplication **177**
C.1 The Implementation . . . 178
C.2 The Environment . . . 179
C.3 Results, Analysis and Comparisons . . . 179

C.4 Solutions and Suggestions . . . 182

D MPI-2 and Parallel I/O **186**

D.1 MPI I/O Concepts and Semantics . . . 186
D.2 MPI-I/O Data Partitioning . . . 188
D.3 MPI-I/O Data Access Functions . . . 189
D.4 Positioning . . . 189
D.5 Synchronisation . . . 190
D.5.1 Coordination . . . 191
D.6 Collective Operations . . . 191
D.6.1 Consistency Semantics . . . 191

E Parallel I/O Tests **192**

E.1 Test Conditions . . . 192
E.2 Writing to the file test . . . 192
E.3 Reading from the file test . . . 193
E.4 Rewriting a file . . . 193
E.5 Experimental results, tests . . . 194
E.5.1 Running the benchmark for one node . . . 194
E.5.2 Running parallel I/O benchmark for two nodes . . . 194
E.5.3 Running parallel I/O benchmark for four nodes . . . 196
E.6 Summary . . . 196

F Kernel-level Algorithmic Tests **199**

F.1 Row/Column Striped Algorithm . . . 199
F.2 Cannon's Algorithm . . . 200
F.3 Sorting Routine . . . 201
F.4 Multigrid Relaxation Routine . . . 202

G Modified Algorithmic-level Test Results **203**

List of Figures

1.1 Moore's Law [149] . . . 2
1.2 The evolution of computers towards clusters (networks of workstations) . . . 5

2.1 The TCP/IP protocol suite . . . 12
2.2 Relative performance difference between processors and DRAM . . . 13
2.3 Conventional TCP/IP implementation . . . 14
2.4 A typical Myrinet packet structure . . . 18
2.5 Myrinet NI block diagram. . . . 19
2.6 2-D Concurrent Network Architecture source [111] . . . 20
2.7 Communication bandwidth and latency measurements over Ethernet . . . 22
2.8 Communication bandwidth and latency measurements over FastEthernet . . . 24
2.9 MPI on top of the TCP/IP protocol stack . . . 25
2.10 GAMMA throughput taken from [40] . . . 28
2.11 The BIP protocol stack compared with TCP/IP . . . 31
2.12 Network Interface attachments on a workstation system . . . 34

3.1 Client Server model . . . 39
3.2 The RPC mechanism . . . 40
3.3 Network communication with RPC calls [217] . . . 40
3.4 *Components of a distributed file system* . . . 42
3.5 Categories of Cluster hardware . . . 46
3.6 The Farm . . . 51
3.7 Classification of clusters of workstations in the campus . . . 52

4.1 The message passing model . . . 57
4.2 Distributed IPC functions . . . 58
4.3 A communicator identifies the process group and context . . . 59
4.4 MPI calls format: the data part plus the envelope part . . . 59
4.5 MPI communication types . . . 60
4.6 Upper and lower layers of MPICH . . . 61
4.7 One sided communication access (put) . . . 63

5.1 Different benchmark levels could have different importance evaluation interest . . 67
5.2 The LogP abstract model parameters; source [53]. . . . 75

5.3 Benchmarking classification areas uniprocessor vs. multiprocessor 77
5.4 The importance of efficiency from low-level to higher-level tests 78

6.1 A simplified communication cost model $t = t_s + t_w n$ 93
6.2 Latency of blocking and non-blocking MPI communication modes 94
6.3 The underlying *MPI_SentRecv* calls of an *MPI_Barrier* routine 96
6.4 MPICH Broadcast call communication pattern 97
6.5 MPICH Reduce call communication pattern on an 8 node communicator 98
6.6 MPICH All-to-all call communication pattern on 5 node communicator 98
6.7 The data parallelism model with a domain decomposition phase 99
6.8 Gather/Scatter operations 101
6.9 A shift right operation within all processors of a communicator 101

7.1 Latency and bandwidth of SP2 and CS2 Southampton and SP2 at Argonne. . . . 108
7.2 Network latency and bandwidth performance on the Linux cluster 110
7.3 Latency and bandwidth of the Linux cluster 110
7.4 Latency and bandwidth of the NT cluster 111
7.5 Latency and bandwidth on SPARC workstation clusters 112
7.6 Latency and bandwidth of the Solaris cluster 113
7.7 Latency and bandwidth on O2 cluster 114
7.8 Communication level latency and bandwidth on O2 cluster 114
7.9 Latency and bandwidth on the BIP cluster for bare network protocols 115
7.10 Latency and bandwidth on a Myrinet cluster with page alignment 116
7.11 Comparing latency and bandwidth between a Myrinet cluster and MPPs 117
7.12 Comparing Latency and bandwidth of our clusters 118
7.13 Barrier Synchronisation test for the SPARC cluster 120
7.14 Broadcast test on a SPARC cluster 120
7.15 Reduce test on a SPARC cluster 121
7.16 Barrier Synchronisation test for the SGI cluster 122
7.17 Broadcast test on a SGI cluster 122
7.18 Reduce test on an SGI cluster 123
7.19 All-to-all test on an SGI cluster 123
7.20 Barrier Synchronisation tests for the BIP cluster and MPPs 124
7.21 BIP cluster broadcast tests 125
7.22 BIP cluster reduce tests 126
7.23 Speed-up curves, Amdahl's law and communication overhead 129
7.24 Kernel-level broadcast operation tests 130
7.25 Kernel-level scatter operation tests on the SPARC cluster and the BIP cluster . . 132
7.26 Kernel-level gather operation tests on the SPARC cluster 133
7.27 Kernel-level shift operation tests on the BIP cluster 134
7.28 Matrix Row/Column Striped and Canon algorithm on the SGI cluster 135
7.29 Communication vs computation part between matrix algorithms 136

7.30 Matrix Row/Column Striped and Cannon's algorithm on the SPARC cluster. . . 137
7.31 Communication vs computation part between matrix algorithms 138
7.32 Relative speedup results of the multiplication algorithms 138
7.33 Sorting algorithm results 140
7.34 SGI cluster multi-grid relaxation test results 141

8.1 Baseline and standard module tests 145

A.1 The 802.3 frame format 171
A.2 Theoretical performance of an Ethernet packet 173
A.3 The structure of an Ethernet packet with TCP/IP header overhead 173

C.1 Data flow among processes 179
C.2 The heterogeneous network architecture used 180
C.3 Elapsed time between sequential and parallel code 181
C.4 The two MPI sessions 181
C.5 Matrix multiplication: a total view upshot of the program 183
C.6 Sequential algorithm vs. parallel Strassen's algorithm 184
C.7 MPI sessions I and II of the SGI O2 switched Fast Ethernet cluster 184

D.1 An etype, a filetype, and a view of file for a process 187
D.2 Tiling a file with filetypes of three processes 188

E.1 The interconnection of the cluster 193
E.2 Parallel I/O for one node, writing to a file 194
E.3 Parallel I/O node reading a file for one 195
E.4 Parallel I/O for one node, rewriting a file (throughput) 195
E.5 Writing a file test on two nodes 196
E.6 Reading a file test on two nodes 196
E.7 Rewriting a file on two nodes 197
E.8 Writing to a file over 4 nodes 197
E.9 Reading from a file over 4 nodes 197
E.10 Rewriting to a file over 4 nodes 198

G.1 Modified striped algorithm (first part) 204
G.2 Modified striped algorithm (second part) 204
G.3 Modified Cannon algorithm 205

List of Tables

1.1 The 1998 Semiconductor Industry Association Roadmap 3
1.2 Forthcoming CPUs and OS are built on 64 bit architecture. 5

2.1 Protocol stack comparison [155] 11
2.2 Latency and bandwidth characteristics for different networks of workstations . . 23
2.3 User space protocol characteristics 29
2.4 Latency and bandwidth performance on a Myrinet cluster using BIP 32
2.5 Ping-pong test results on various communication libraries 33

3.1 Parallel Systems, Clusters and Distributed Systems comparison 45
3.2 ASCI machines summary 48
3.3 Important software development areas for Beowulf class clusters 50
3.4 Different cluster configurations 52

5.1 Units and symbols used in PARKBENCH follow the extension of the SI system . 69
5.2 PARKBENCH tests 72

6.1 Differences between MPPs and cluster 81
6.2 Timing Registers of modern processors, for more information see Appendix B . . 88

7.1 Cluster configurations used for testing with SCOPE benchmarks 106
7.2 Individual processor SPECint95 SPECfp95 figures [204] 107
7.3 Ping-pong test results on various communication libraries 117
7.4 Latency and Bandwidth results 119
7.5 Broadcast operation test measurements (time in μs) 126
7.6 Reduce operation test measurements (time in μs) 127
7.7 Comparisons with other benchmarks 142

8.1 Minimum hardware system requirements for the SCOPE baseline tests 145
8.3 Workstation (w/s) architectures and OS SCOPE distribution will support 148

A.1 Ethernet frame field sizes 172
A.2 $n_{1/2}$ size for each protocol layer 172

C.1 The heterogeneity of the cluster 180

C.2 Sub-matrices exchange among processes before optimisation 182
C.3 Optimised sub-matrices exchange 183

G.1 Performance improvement for matrix-to-matrix algorithms 206

Acknowledgements

First and foremost I would like to thank my supervisor Ed Zaluska for his inspiration, guidance and support throughout the period of this research.

Invaluable support and feedback was received from Parallel and Distributed Computing (former Concurrent Computation Group) group members, in particular I would like to thank Jeff Reeve, David Lancaster and Vladimir Getov together with my fellow postgraduate students. In particular I am grateful to Duncan Simpson, Antonio Barragan, Emilio Hernandez, and Mario Dantas who have been an invaluable source of encouragement.

I am grateful for all the support I received from computing services staff in PDC, SUCS, Lyon and Argonne who have assisted with my research. I would like to thank Andrew Perkins, Stephan Mechnig, Antje Neuhoff, Costas Koufopoulos and Mirjana Andric for their valuable friendship and support.

I would also like to thank the Greek State Scholarships Foundation for the support I received during the first period of this research.

Finally I am indebted to my family for their understanding and their invaluable support throughout my life without which I could not have completed this thesis.

Chapter 1

Introduction

This thesis investigates the performance evaluation of workstation clusters, in particular those configured to provide a cost-effective High Performance Computing (HPC) environment. The thesis reviews the recent developments in commodity hardware and software which have made such clusters possible, introduces a methodology for a comprehensive examination of workstation cluster performance and proposes a tailored benchmark evaluation tool for clusters called Specific Cluster Operation and Performance Evaluation (SCOPE).

This introductory chapter will first review the present-day requirement for HPC before discussing briefly the fundamental difficulties in delivering this requirement. It will then discuss the role of workstation clusters in providing HPC and introduce the importance of being able to evaluate the intrinsic performance of commodity workstation clusters.

1.1 The Requirement for High Performance Computing

The information revolution has been described as the third evolution in our civilisation (the first two being agricultural and industrial) [177] and one of the most influential innovations of the last century. Over the past few decades computers have become an essential part of the modern world. In science and engineering the classical methodology of design and experimentation has been converted into analysis-simulation-optimisation-implementation and now each of these steps would be impracticable without the aid of computers. Classical business and commerce have adapted to exploit distributed computing, while Internet e-business and e-commerce have become an integrated part of everyday life. Entertainment and multimedia over the past few years have also increased demands for HPC dramatically. Moreover new application opportunities that require HPC emerge e.g. artificial intelligence (AI), genetic programming, dynamic interactive simulation [207].

The impact of computing in all of these domains has become a driving force which encourages the even greater development of computing [3]. Technological improvements in VLSI integrated circuits have been able to follow Moore's law [85, 238] over the last twenty years and this trend is expected to continue in the near future [76]. The implication of Moore's law is twofold: a linear increase in transistor switching clock rate together with a quadratic increase in

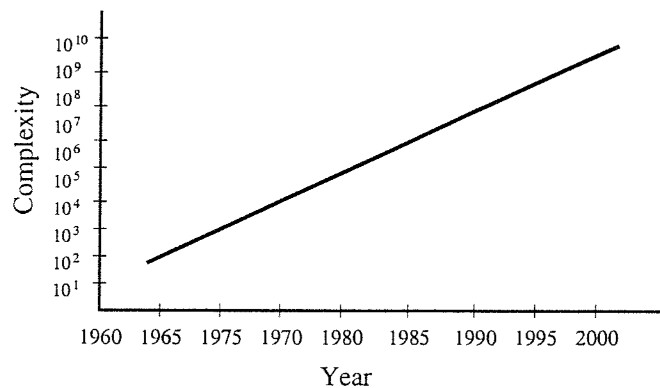

Figure 1.1: Moore's Law [149]

the number of transistors. This additional functionality together with architectural innovation and software development (e.g. advances in compiler technology to increase instruction-level parallelism) has provided improved computer performance, by a factor of perhaps two every year, [107] compared to Grosch's law [98], which postulated that computation cost is related to the square root of computational power [68, 121].

Despite those impressive achievements there are problems of great importance in science and engineering which remain intractable, because their solution inevitably requires an enormous amount of computational power or computing resources. Such problems are known as the "Grand Challenge" [115, 239] applications; examples of such applications include Fluid Dynamic modeling, Molecular Dynamics, Quantum Chromadynamics, simulations of various physical phenomena, weather forecasting, etc. In business the challenging problems include database and transaction processing, data mining and warehousing, telecommunication processing, network applications and high-performance real-time systems [136]. The availability of cost-effective computational power also encourages new classes of applications with no direct non-computing equivalent [104, 176]. New applications and demands for computing have in addition emerged from the computer industry itself known as "market-enabled" and "user-driven" such as image processing, computer graphics, animation and virtual reality (with applications to the entertainment industry i.e. film-making, film restoration, games), medicine, education (multimedia) [12]. Taking all these potential applications into account, the demand for HPC is expected to increase very considerably over the next few years.

1.2 High Performance Computing and Parallelism

A common requirement of most high-performance computing applications is to accomplish their tasks as fast as possible. There are three fundamental ways to accomplish a task faster: increase the hardware speed (e.g. clock rate), use more intelligent and efficient algorithms, or make use of parallel processing [178].

For the first alternative, it will be difficult for current CMOS technology to continue to improve at the current rate in the future, because fundamental limits will soon be approached and

Table 1.1: The 1998 Semiconductor Industry Association Roadmap update for high-end processors [10, 76]

Specification/Year	1997	1999	2001	2003	2006	2009	2012
Feature size (micron)	0.25	0.18	0.15	0.13	0.1	0.07	0.05
Supply voltage (V)	1.8-2.5	1.5-1.8	1.2-1.5	1.2-1.5	0.9-1.2	0.6-0.9	0.5-0.6
Transistors/chip	11M	21M	40M	76M	200M	520M	1.4B
DRAM bits/chip	1 67M	1.07G	1.7G	4.29G	17.2G	68.7G	275G
Die size (mm^2)	300	340	385	430	520	620	750
Local clk freq. (MHz)	750	1250	1500	2100	3500	6000	10000
Global clk freq. (MHz)	750	1200	1400	1600	2000	2500	3000
Max power/chip (W)	70	90	110	130	160	170	175

there is no obvious follow-on technology. According to the Semiconductor Industry Association (SIA) projections, the number of transistors per chip and local clock frequencies will continue to grow exponentially in the near future as Table 1.1 shows. However, increased complexity (resulting in additional constrains) at that level will limit its usefulness [76].

The alternative of algorithmic improvements is not always possible and can be restricted (not well mapped) by the underlying hardware architecture. Long before reaching these limits research projects had begun to consider the alternative of parallel processing which theoretically can provide a viable solution to limitations in processing power. The term High Performance Computing (HPC) has now replaced the older descriptions of "parallel computing" or "supercomputing" as almost all present-day HPC involves the use of parallel computing.

From the second half of the 1980's there has been a trend towards parallel and distributed computing as computers have become more available and accessible with a wider range of applications at an improved price-performance ratio. Parallel processing, at that time, was widely believed to be the key enabling technology which must inevitably be adopted to achieve the necessary high performance for grand challenge and other future applications [64]. The accepted view was that switching to parallelism was only a matter of time before compilers were developed capable of optimising programs automatically to run efficiently on multiprocessor systems. Since then much research has been completed, parallelism difficulties are more fully appreciated and (with hindsight) the earlier optimism appears naive.

As Wilkes [238] quotes "parallelism is not a panacea" and an underlying hardware technology on its own is not enough to exploit parallelism. Moreover the nature of parallelism inherent in applications has to be first understood and then implemented in the programming model. The algorithm implementation should also map onto the underlying hardware architecture naturally. This dependence between parallel algorithms and architectures introduces an inherent disadvantage, the design space of such systems becomes very wide (hence a difficult task) and provides neither a flexible unified programming model nor portability.

Although in the past parallel computing was regarded as a theoretical possibility with limited interest restricted to academia due to its difficulties, parallelism today is now accepted as

the only viable solution for HPC. The development and establishment of parallel programming models and standards together with performance evaluation studies have accelerated the deployment and use of both parallel systems and parallel applications. In other words, commercial hardware and software vendors have gradually started to build parallelism into their products. Improvements in availability and cost/performance ratio for these systems are a fundamental requirement for the effective use of parallelism in scientific and commercial applications. The first part of this thesis investigates and reviews the design space of workstation clusters as a HPC system.

1.3 Workstation Clusters as an Alternative Platform for HPC

Originally the success of Parallel Computers was limited to Grand Challenge applications or high-end large-scale commercial applications because the cost of a parallel platform was prohibitively high for other applications [38]. Parallel computers have never achieved the volume production necessary to generate "significant economics" of scale in the same way as personal computers or workstations [12]. Low-volume manufacturing is clearly a crucial disadvantage of large parallel systems, in contrast to the cost-effective manufacturing possible with mass-produced workstations. At the same time the steady developments in "sequential" processor technology has eliminated the historical performance gap between traditional mainframe-processors and commodity workstation processors. This trend has encouraged vendors of Massively Parallel Processors (MPP) to use "killer micro" components (such as microprocessors and low-cost memory) as the building blocks for their new high-performance parallel computing systems [63, 6, 43]. One side-effect of this strategy is of course the engineering lag time (estimated to one or two years) between delivered workstations and delivered MPPs based on the same microprocessor components [6, 242].

MPP systems of this type incorporate commodity workstation parts with dedicated tightly-coupled high-speed networks. A full version of the Operating System (OS) usually runs on each node and coordination among nodes is achieved by the explicit exchange of messages. Future technology trends are moving towards microprocessor-based symmetric multiprocessor (SMP) clustered schemes [43]. Examples of this class include the IBM SP series, the older Intel Paragon, Thinking Machines CM5, Cray T3E, and the machines of the ASCI project [42]. This type of computation is usually a Multiple Instruction Multiple Data (MIMD) or Single Program Multiple Data (SPMD) type [75], both of which can provide scalability and high performance for many application domains. A loosely-coupled version of such a parallel platform can be implemented using a network of workstations, in which each node has its own stand-alone OS connected by moderate-to-high latency LANs. Typical examples of this class of clusters, examined in this thesis, are the NOW project [51] and the Beowulf project [208] both of which are briefly examined in Chapter 3. Another form of clustering is "cycle harvesting" on idle workstations which in some circumstances can provide an extremely cost-efficient parallel computing environment [54].

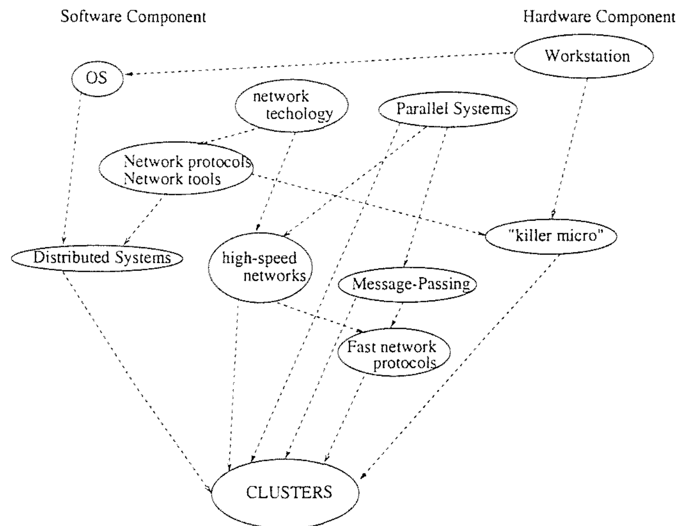

Figure 1.2: The evolution of computers towards clusters (networks of workstations)

Table 1.2: Forthcoming CPUs and OS are built on 64 bit architecture.

Vendor	CPU	System Arch.	OS	Available
Intel	IA-64	64 bit	(HP/Windows/Linux)	No
Sun	UltraSparcII/III	64 bit	Solaris	Yes
DEC	Alpha	64 bit	Digital UNIX	Yes
MIPS	R10000	64 bit	IRIS	Yes
IBM	RS6000/ Power3	64 bit	AIX	No
Motorola	PowerPC	64 bit	Mac OS	Yes

Advances in microprocessors, intercommunication networks and distributed software tool development are converging in favour of clusters of workstations as Figure 1.2 illustrates. Moreover, forthcoming 64-bit-generation systems (Table 1.2) will provide enhanced features for distributed clustering system support as well as extended scalability, availability and interoperability features. As a result clusters of commodity workstations are now accepted as a viable platform major HPC applications [6]. The initial motivation for this thesis is based on this technological convergence that enables low-cost workstation clusters to exploit parallelism. The key infrastructure components required for workstation clusters infrastructure will be examined in the first chapters of this thesis.

Despite improvements in accepted standards and interoperability over the past twenty years, clusters have inherent potential difficulties resulting from a relatively-high software communication latency and lack of a "single system image" in terms of the software programming environment [80], OS support, job allocation, load balancing and run time support [171]. Recent research in this area [38, 182, 151, 236, 39] has nevertheless demonstrated that clusters of inexpensive high-end PCs interconnected with off-the-shelf hardware and a suitable OS can deliver

acceptable performance over a wide range of parallel applications [59, 57]. The next chapter of this thesis investigates in detail the impact of off-the-shelf interconnection components on various workstation cluster schemes.

In terms of the underlying programming model, networks of workstations implement a form of a multicomputer parallel machine based on the concept of message-passing. In this model each processor has its own local memory and processes interact by sending and receiving messages over the interconnecting network. In practice this has been recognised as the most efficient programming paradigm on clusters of workstations [80, 185] and it is identical to the computational model currently used by most large parallel computers. This feature is very important as it automatically implies that workstation clusters can have the same approach to computation and an identical problem description model in addition to the same execution model as MPPs [201]. Hence networks of workstation can inherit most of the existing software techniques and methodologies from parallel systems (MPPs) which can be applied or adapted to clusters relatively easily [104]. Single Program Multiple Data (SPMD) applications can be implemented directly using the message-passing mechanism while a limited number of Multiple Program Multiple Data (MPMD) programs can also be implemented [3].

Many libraries supporting the message-passing model have been developed (e.g. PVM [83], Linda P4 [146], Express [174]). The Message Passing Interface (MPI) [77] has now became the accepted standard for this model. Chapter 4 of this thesis investigates MPI implementation issues on workstation clusters.

1.4 Performance Evaluation of Workstation Clusters

Traditionally the primary target of parallel systems is the delivery of high-performance computing at the application level. Efficient use of resources or a high algorithm efficiency was not always a concern of primary importance in parallelism which sometimes made relatively inefficient use of the resources available. Performance evaluation and benchmarking, which historically was developed for assessment and comparisons between different computers, has increasingly become important for parallel computer systems where the nature of a particular class of applications might map preferably into one particular parallel architecture. In addition, benchmarking results provide very useful feedback to system designers and application developers to assist in the understanding HPC systems behaviour.

The main motivation for this thesis set out in the previous section is the technology evolution of commodity workstation components that now allows workstation clusters to be used as a flexible cost-effective HPC platform available for a wide range of applications and usable by programmers without specialist skills. However, in practice workstation clusters have often failed to exploit the potential advantages. Further evolution and development of these systems requires detailed performance evaluation measurements and results analysis. Performance on these systems is greatly dependent on the efficient implementation and integration of technologies first developed for other systems such as communication libraries, underlying network protocols and network architectures [167]. Any generalisation of performance results from networks of workstations becomes hard to qualify because of the large numbers of system

variables.

Existing HPC benchmark suites for message-passing systems are designed primarily for Distributed Memory or Shared Memory MPP systems for several scientific application classes such as Computational Fluid Dynamics problems, numerical analysis, etc.

Most of these benchmarks, in principle, will also run on clusters of workstations but simply because clusters support the identical programming model as MPPs. Although theoretically the above condition is sufficient for an MPP benchmark to run on a workstation cluster to provide quantitative performance evaluation ("how" much), it does not necessarily provide qualitative performance evaluation ("why") about specific performance characteristics of clusters of workstations. Most of these benchmark suites, when they run on clusters of workstations, are not suitable to provide relevant information either because their workload does not take into account the individual characteristics and configuration issues of workstation clusters (such as limitations of the messaging system and its impact on different applications) or they measure performance at a higher level which is not sufficient for performance evaluation.

The original contribution of this research presented in this thesis is a performance evaluation tool known as the Specific Cluster Operation and Performance Evaluation (SCOPE) benchmark set which will assist further the establishment of workstation clusters concept. SCOPE is a benchmark suite that provides a comprehensive and optimised set of tests for workstation clusters. This benchmark is designed to achieve the following aims and objectives:

- Evaluate the potential characteristics of workstation clusters by the provision of a comprehensive benchmark set suitable for the design space of these systems.
- Provide commodity system managers/developers with a tool that will assist them to understand, analyse and optimise in the best possible way the performance behaviour of their clusters.
- The SCOPE benchmark methodology will in addition can be expanded to provide application developers with a useful tool to understand and program clusters in the most effective fashion e.g. provide application cost and scalability prediction.

1.5 Summary

This thesis examines the fundamental performance factors inherent in workstation clusters and demonstrates that additional analysis is necessary to optimise overall performance. Chapter 2 discusses workstation cluster intercommunication issues such as networking and communication protocols. The next Chapter 3, examines the workstation cluster concept from the prospective of distributing systems and parallel systems. In Chapter 4 the message passing paradigm as the main computational model of networks of workstations is reviewed. The next chapters of the thesis focus on the key benchmarking issues. Chapter 5 outlines benchmarking issues for HPC systems, Chapter 6 describes in detail the proposed SCOPE benchmark suite tailored for clusters of workstation and Chapter 7 provides experimental results of benchmark tests on

various workstation clusters. Finally the conclusions and proposed future work are discussed in Chapter 8 and 9.

Chapter 2

Low-level Internode Communication

2.1 Introduction

The communication subsystem is a key fundamental component of distributed systems in general and networks of workstations in particular. The performance of the internode communication subsystem in clusters of workstations is fundamental, because any imbalance in the design of this subsystem can cause communication bottlenecks which will have a significant impact on both the behaviour and the overall performance of the entire system [144]. The task of a *communication sub-system* is to transfer data from one application to another application (which resides on another node) transparently. The term communication subsystem includes software components such as interfaces, protocols and communications handlers as well as the communication hardware. Some of these software components could be incorporated into the Operating System or even form part of the applications software [49]. This chapter provides a survey of the fundamental characteristics and semantics of communication subsystems and protocols adopted over the last few years for clusters of workstations.

2.2 Interconnection Issues

This section presents the main interconnection categories used in computers and clusters of workstations. There are three main groups into which computer networks could be divided [126]:

- wide-area networks,
- local-area networks (LANs, SANs[1]),
- massively parallel processor (MPP) networks (direct and indirect networks)

The last category includes very fast (traditionally proprietary) networks able to interconnect thousands of nodes in a very small physical distance. They are constructed mainly from

[1] A System Area Network is a communication network which provides low latency, high bandwidth, and very low error rate links between nodes [48].

switches and links (switches route the messages and links carry the messages). The communication subsystem of MPPs is strongly influenced and bound by the underlying network connection topology e.g. tree, mesh, switch, etc [142]. Workstation clusters do not often use such networks usually because of the relatively-high cost and the lack of standardisation of these networks.

WANs and LANs differ from direct and indirect proprietary networks because they are based on standards which are widely approved. WANs often include thousands of computers distributed throughout a region, with an error rate generally significantly higher than LANs and they usually provide connection-oriented services. LANs on the other hand connect hundreds of computers located closely together (e.g. in one or more buildings) with low error rates and they usually provide connectionless services. A special category of LANs has emerged over the last few years with the advance of network technology known as "System Area Networks" (SANs) which use proprietary very-high-speed networks in a physically small area for a limited number of nodes and functionality [48, 85].

2.3 Communication Software Layers

Computer networks are usually designed in a highly-structured way, to simplify the design, development, and operation of the network, and in addition to allow (in theory) a relatively-smooth network evolution [8]. The basic modules of this structure are layers or protocols. Their purpose is to offer certain services to the higher layers while shielding those layers from the details of how the offered services are actually implemented. Each layer has to provide services to the layer above it.

Two main structured-scheme layer models exist: the TCP/IP protocol suite and a reference model called Open Systems Interconnection (OSI) from the International Organisation for Standardisation (ISO). Both models are similar in many aspects: they are both based on the concept of a stack of independent protocols, and the functionality of their layers is similar (but different). A user-application message must be processed at each layer of the stack in order to be sent to and received from the network.

The TCP/IP reference model is developed from experience with older protocols. It is a collection of complying protocols rather than a model. Most of its protocols are very effective and widely used. However, the specification was not separated from the implementation which can sometimes introduce difficulties with new network technologies [13, 26].

In the OSI model, the distinction between interfaces and protocols is clear

- Service: this states what a layer does
- Interface: this states how layers above it can access it
- Protocol: this is used to get a job done within a layer

In each case no implementation is defined, the number of layers the OSI model involves is large (seven) and the functionality of some of its layers is ill-defined. Implementations tend to be slow and often avoided because of the complexity of the protocol. This is mostly because when OSI was first specified, the TCP/IP protocols were already widely established [222] and

Table 2.1: Protocol stack comparison [155]

OSI Ref. Layer NO	OSI Layer Equivalent	TCP/IP Application	TCP/IP Examples
7	Application,		telnet, rsh, NFS
6	Session,		name services
5	Presentation		
4	Transport	Transport	TCP, UDP
3	Network	Internet	IP, ARP, ICMP
2	Data Link	Data Link	IEEE 802.2
1	Physical	Physical Network	Ethernet, etc

they provide adequate performance for the majority of applications. There was therefore little commercial interest in adopting a new more complex protocol and the OSI model is now often considered to be of theoretical interest.

2.3.1 The TCP/IP Stack

The TCP/IP protocol suite has become the universal network protocol suite standard and as such it has been used extensively in workstation clusters both experimental and also production platforms. The TCP/IP protocol suite has its origin in the ARPANET project [222]. Most of the existing LAN distributed systems and clusters use this protocol suite over a 10/100 Mbit/s Ethernet channel. The protocol suite has a four-level layer scheme which does not match the OSI layering hierarchy scheme particularly well, but it is nevertheless the most widely-used network protocol encountered in both LANs and NOWs. TCP/IP provides both a connectionless and a connection-oriented reliable byte stream service [209], because it was originally designed for WANs with relatively high-error transmission rate. The original philosophy behind TCP/IP was communication among autonomous machines rather than resource commonality [187].

In practice the TCP/IP protocol suite is a collection of networking protocols that conform to the Internet Protocol scheme. The Application Programming Interface (API) which the TCP/IP suite provides is sufficient to support many network and distributed system applications as shown in Figure 2.1. The four conceptual layers which are built above the hardware layer are:

The Data Link Layer: This layer accepts and delivers Internet Protocol (IP) packets. Different protocols are used in this layer depending on the type of physical network.

The Network Layer: This layer handles communication from one machine to another, enabling hosts to inject packets into any network and have them travel independently to their destination, (it also handles connection rendezvous, flow control, retransmission of lost data, data management, etc).

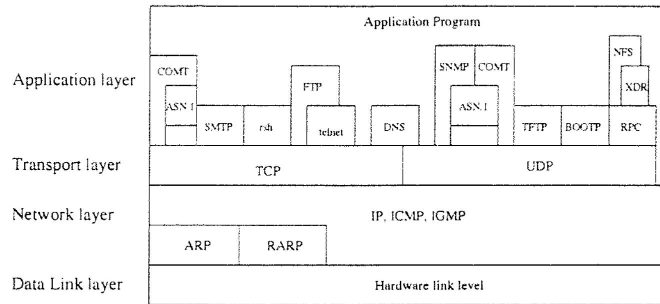

Figure 2.1: The TCP/IP protocol suite

The Internet Protocol [179] resides in the Network layer and it is the foundation of the TCP/IP architecture. It provides the fundamental service of a connectionless unreliable "best-efforts" packet delivery system. Its purpose is to standardise the basic unit of data transfer (datagram or packet) through the TCP/IP Internet[2] .

The Transport Layer: This layer provides reliable or unreliable communication between end-to-end application programs.

The transmission layer protocol [180, 24] can provide either an unreliable connectionless delivery service (User Datagram Protocol) or a reliable connection-oriented, stream delivery service (Transmission Control Protocol) over the transmission medium regardless of the underlying transmission rate, delay, error rate or reordering of packet delivery.

The Application Layer: This is the highest level and it contains protocols that implement user-level functions and applications software (e.g. telnet, File Transfer Protocol FTP, http, e-mail).

2.4 Analysing Communication Overhead

Optimisation of the communication subsystem performance can lead to substantial improvements on a workstation cluster overall performance. This section examines in detail the communication overhead issues encountered in workstation clusters. Traditional ways of eliminating communication overhead requires changes either to the Application Programming Interface (API), or the communication protocol, or the protocol implementation [187]. The first two approaches do not preserve compatibility either with older applications or with older protocols and applications require re-implementation e.g. [232, 231, 234]. The third approach requires changes to the protocol implementation, but applications preserve compatibility.

Communication support has traditionally not been considered to be an integrated part of an OS simply because the widespread adoption of networks is comparatively recent. The fundamental design objectives of traditional OS and protocol stacks at that time were reliability,

[2]The "Internet" is the global collection of connected networks and gateways that use the TCP/IP protocol suite and function as a single virtual network.

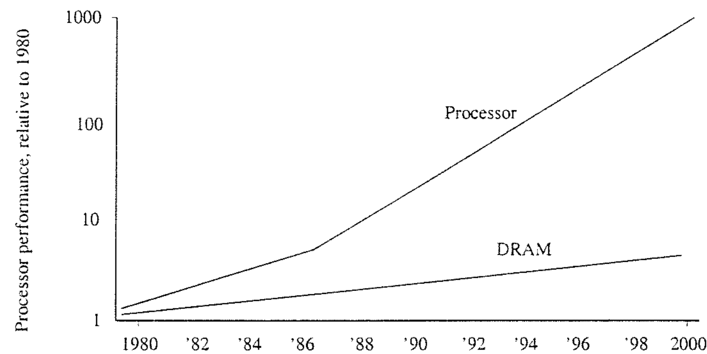

Figure 2.2: Relative performance difference between processors and DRAM (Moore's law) [176]

programmability, process protection and re-usability over relatively expensive and unreliable hardware resources. Most of the processing overhead resides in the OS and many implementations fail to achieve high throughput because they access data several times. Data handling requires at least one memory copy operation from the user workspace to the network interface.

Several years ago the bandwidth of the main memory and the disk I/O of a typical workstation was an order of magnitude faster than the physical network bandwidth. That difference in magnitude was invariably sufficient for the existing OS and communication protocol stacks to saturate network channels such as 10 Mbit/s Ethernet or even 100 Mbit/s LANs [183]. Despite hardware improvements, the memory access time and internal I/O bus bandwidth in a modern workstation has not increased significantly during this period (memory performance improvement is around 7% per annum while processor improvement approximates 50% per annum [76]) the main improvements in performance have come from caches and a better understanding of how compilers can exploit the potential of caches[3]. Thus the gap between the network bandwidth and the internal computer resources has been considerably reduced [183]. At the same time processors are running 30–50 times faster than DRAM which makes the task of hiding memory latency significantly more difficult [106].

Traditionally communication protocols have been designed on the assumption of an unreliable erroneous physical link, a packet sent over the network could be duplicated, lost, damaged, arbitrarily delayed or even dropped. TCP/IP, for example, incorporates features such as in-packet end-to-end checksum, packet delay and time-out policies as well as a packet fragmentation and reassembly scheme including out of order delivery and retransmission policies [187]. All these features are useful in WANs but in LANs and clusters of workstations they impose stages of redundancy and consume unnecessary computational power e.g. contemporary LAN network interfaces inevitably perform cyclic redundant check computation (CRC) per packet in hardware. As a result the transition of data in a multiple layered structure can be very costly

[3]A full cache miss on an Alpha 21164 can cost up to 180 ns/1.7 ns = 108 clock cycles x 4 or 432 instructions

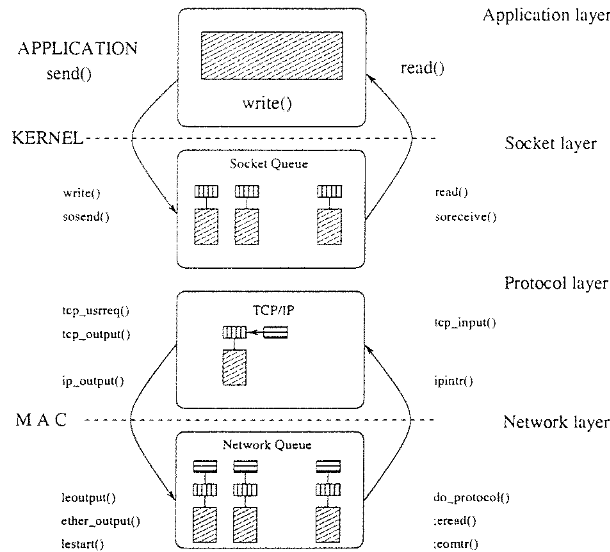

Figure 2.3: Conventional TCP/IP implementation

in terms of computing power, e.g. poor code locality, multiple memory-to-memory copies, different data abstraction between layers, packet headers and complicated memory management mechanisms all increase latency and reduce the effective bandwidth.

In a conventional implementation of TCP/IP a *send* operation involves the following stages of moving data: data from the application buffer are copied to the kernel buffer, then packet forming and calculation of the headers and the checksum takes place, finally packets are copied into the network interface for transmission. At the reception end the network driver copies an incoming packet into a kernel buffer where packet headers removed and calculation of the checksum takes place, before the *recv* operation copies data from the kernel buffer space to the application buffer space [55]. This requires extra context switching between applications and the kernel for each system call, additional copies between buffers and address spaces, and result in generally increased computational overhead [170].

Pasquale et. al. [175] analysed the software communication overhead of a TCP/IP protocol stack for a cluster of DECstation 5000/200 workstations used for the Sequoia 2000 project and they categorise functions commonly used by TCP/IP (and UDP/IP) protocol stacks as:

Checksum: checksum computation

DataMove: moving data to different memory locations

Mbuf: message buffering

ProtSpec: protocol specific operations, e.g. header fields computation

DataStruct: data structure manipulation

OpSys: OS overhead

ErrocChk: user and system error check

Other: operations too small to measure

These types of TCP/IP protocol overhead can be divided into data-touching operations, (i.e. data move and checksum) and non-data-touching operations. The cost of the first division scales linearly to the packet size and becomes the dominant overhead for large packets. The cost of non-data-touching operations is comparatively constant and dominates the overhead for small packets. Optimisation of the checksum computation on that network improved throughput by 37% and elimination of the checksum improved throughput by 74% [175].

Communication libraries and message-based applications often implement a redundant stream protocol on top of TCP/IP in order to become portable and platform independent [219]. Implementations of these libraries might be insensitive to the characteristics of a given interconnect, for example client-server applications might wait for an explicit reply-acknowledgement message to a request. In many cases it is common for a communication library to insert explicitly the message length of the stream, and at the receiving end to loop indefinitely to ensure acquisition of the entire message e.g. socket case [47].

2.4.1 Optimising the Communication Processing Overhead

Communication processing overhead in parallel systems and workstation clusters is often analysed into two parts, one per-segment cost and one per-byte cost [63, 186]. The fixed cost per segment could include various tasks of the OS such as interrupt mechanisms, buffer allocation, resetting I/O devices, waking up processes and resetting timers, etc. The per-byte cost is variable in handling data, e.g. a conventional TCP/IP implementation can have up to four memory operations. Data copied from the application buffer to the kernel buffer through the CPU requires two memory operations (read, write), then the calculation of the checksum requires another memory operation (read), and the final copy into the network interface requires one memory operation (DMA transfer).

Various communication models [210, 63, 186] have been developed in order to evaluate communication latency among processors in parallel systems. The total latency of a message according to [176] is:

$$Total\,latency = Sender\,overhead + Time\,of\,flight + \frac{Message\,Size}{Bandwidth} + Receiver\,overhead$$

Dongarra et. al. [63, 80] follow a linear approach considering latency t_n (n-byte message) as a constant start-up time t_s (constant per segment cost), and a variable per-byte time t_w and zero per-hop delay γ. The total latency of an n-byte message on h hops is given by equation 2.1. The message length at which half of the maximum bandwidth is achieved ($n_{1/2}$) is an important indication as well.

$$t_n = t_s + t_w n + (h-1)\gamma \tag{2.1}$$

where n is the size of the message. In our case $\gamma = 0$ therefore equation 2.1 is simplified to:

$$t_n = t_s + t_w n \qquad (2.2)$$

Jacobson et. al. [41] suggested prediction and caching techniques that can reduce the fixed per-segment cost and hence improve the performance of the protocol stack, e.g. they observed that most of the incoming TCP segments arrive in order and they have no out-of-band data, while most of the segments exhibit locality.

Reducing the variable time cost of a protocol requires the reduction of memory operations [55]. With the use of additional hardware support there are techniques that can eliminate some of the memory operations required e.g. some CPUs provide a capability of calculating the checksum while copying, i.e. in the *Copy on Write* technique the system makes the user data read-only during a send operation, hence data bytes are copied directly into the network interface. Another technique known as *Page Re-mapping* requires the maintenance of a buffer for the incoming packets. The network interface can split the header and data of incoming packets into separate buffers starting at memory page boundaries. The memory manager can re-map the corresponding data pages to the application without copying. With reference to [55] the *Single Copy* technique dedicates an area in memory which is shared between the processor and the network interface, on send or receive event data are copied into the dedicated area with prefixed headers. Moreover, CPU involvement for copying causes pollution of the process working set with further cache misses with additional performance degradation.

Such an implementation of the TCP/IP protocol stack should not limit the communication performance and can support very high transition speeds. Optimised implementations of the TCP/IP protocol can then move the communication bottleneck down to the network interface. Reported TCP/IP performances using these techniques have achieved throughput up to 200 Mbit/s [55]. Implementations of the IPv6 protocol are expected to take advantage of the simplified Internet packet header fields and reduce further computational overhead. In addition the new protocol can exploit its "jumbogram" features in order to send packages of larger than 64Kbyte with a minimum overhead cost.

2.4.2 High-speed Interconnection Networks

Over the past few years many new network technologies have been developed, but only a few of them have been adapted successfully for clusters of workstations and distributed computing systems. Among the key features a successful high-speed network technology should provide are low cost, high reliability, software availability and compatibility with existing standards. In order to preserve the integrity and the functionality of the TCP/IP suite, each new standard has to provide its own "Network layer" protocol. In this way changes to the underlying interconnection network are made transparent for existent applications. If a lightweight proprietary network protocol is used instead (to increase the throughput further) additional changes and re-compilation of the applications is also required.

Many current high-speed interconnection technologies in system area networks (SANs) provide hardware services with outstanding reliability. Such networks can use "optimistic"[4]

[4]Protocols which speculate and make optimistic assumption about the underlying network reliability [40].

lightweight proprietary communication protocols and introduce new communication modes (e.g. multicast, isochronous or asynchronous, etc) which can increase application throughput further. The most important new technologies which can be used in workstation clusters are reviewed below.

Network Switch Technology Network technologies that use a single shared medium topology suffer network bottlenecks and bandwidth shortages. Switch network technology can alleviate such network bottlenecks by providing high aggregated bandwidth, and increased throughput transparently. Network switch technology prevents unnecessary traffic crossing ports-segments and allow multiple simultaneous communication paths among its port-segments. Switches operate at layer 2 of the ISO reference model by forwarding data with low overhead cost [173]. Traffic is usually forward in cut-through mode for low latency .

Fast Ethernet Fast Ethernet is a variation of the IEEE 802.3 specification. The bit rate is increased an order of magnitude (to 100 Mbit/s) while retaining the same wiring systems, Medium Access Control (MAC) method and frame formats of the old standard. As a consequence the maximum segment distance is reduced down to 100 m. This standard is known as *100BaseT* [101]. Fast Ethernet is acknowledged to be the simplest way to upgrade an existing 10BaseT based cluster network. Currently Fast Ethernet interconnections provide an acceptable cost performance trade off solution for commodity workstation clusters.

FDDI Fiber Distributed Data Interface was developed by ANSI and is defined in ISO 9314 standard. FDDI is based on a 100Mbit/s ring topology and can span over a ring of 500 stations up to 100 km which makes it ideal for a backbone network [101]. FDDI has become well-established for particular applications where it can provide different functionality to Ethernet, despite a significantly higher cost.

Gigabit Ethernet This is an evolution of the Ethernet standard (IEEE 802.3) that scales Ethernet technology to the gigabit range (Gbit/s). The standard in full-duplex mode enables a 2 Gbit/s throughput on a fibre optic medium (IEEE 802.3x/z or 1000BASE-SX). The standard preserves backwards compatibility (with half-duplex mode CSMA/CD) but also provides extensions which increase its functionality e.g. routing, quality of service (QoS) [70]. Both Fast Ethernet and Gigabit Ethernet technologies use Ethernet switches to alleviate single bus congestion problems, increase the aggregate throughput [173] and improve the scalability of the network.

FCS The Fibre Channel industry Standard is an ANSI standard proposed for ISO adoption as well. FCS is a switched system that can simultaneously provide high-bandwidth utilization with distance insensitivity in both directions from 266 Mbit/s to over 4 Gbit/s transfer rate over a 10 Km distance. Both connection-oriented and connectionless classes of services are supported as well as broadcasting and multicasting including Internet Protocol (IP), SCSI, IPI, HIPPI-FP, and audio/video frames [138].

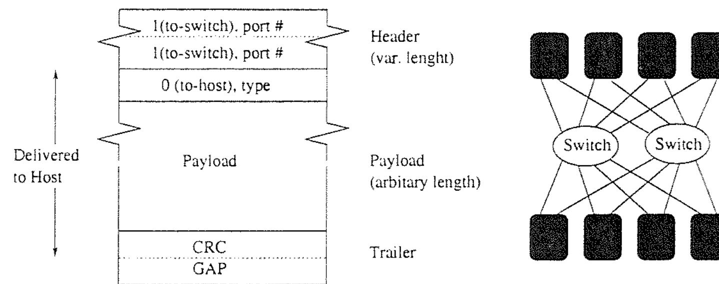

Figure 2.4: A typical Myrinet packet structure, leading header source bytes are interpreted as routing code of one byte with the most significant bit (MSB) set to 1

ATM The Asynchronous Transfer Mode protocol (ATM) is a common transmission protocol that has been internationally defined and agreed by both the computer and telecommunication communities. ATM is a cell-based network which provides transmission and switching support independent of the source media (data, images, voice, video). ATM uses a hybrid form of *circuit* and *packet switching* of fixed-size blocks, called *cells*, over virtual circuits as a compromise between data traffic and audio/video traffic, and QoS. It originated as a way to support Broadband Integrated Service Data Network (B-ISDN) with high data transfer rates (SONET[5], OC-1, OC-12, SDH[6]) [25, 44]. Data rates start at 155Mbit/s and potentially rise to 4.8 Gbit/s.

The cell switching can handle efficiently both point-to-point and multicasting communication modes. ATM implementation has three low-level layers. The *physical layer* is almost identical to layer 1 in the OSI model. The *ATM layer* deals with cells, (routing and transport) but does not provide recovery for lost or damaged cells (it covers layer 2 and partially layer 3 in the OSI layers). The *adaptation layer* handles assembly/disassembly of packets to cells or vice versa [221].

The ATM cell size is 53 bytes made up of a 5 byte header plus 48 byte protocol data unit (PDU) [101] which is a compromise between payload efficiency (around 90%) and low packetisation delay. Relay switches can process cell packets in parallel and then increase transmission speed [43]. ATM technology is widely adopted by all the major telecommunication carriers and hence provides WAN services. It has proved to be too expensive for most LAN applications, except where backbone functionality is required.

Myrinet Myrinet is a switched gigabit-per-second network technology developed by Myricom Inc for high-speed LAN to support parallel processing on NOWs. The design of Myrinet was based on the Caltech Mosaic and the USC/ISI ATOMIC [199, 198, 46] project which implemented the design of a high-speed LAN using MPP components.

The network consists of point-to-point links connecting hosts or switches. Each link is capable of full bi-directional 1.2Gbit/s bandwidth with low latency (of the order of microseconds) and very low error rates [117]. The leading byte of a Myrinet packet determines the outgoing

[5]Synchronous optical network
[6]Synchronous digital hierarchy

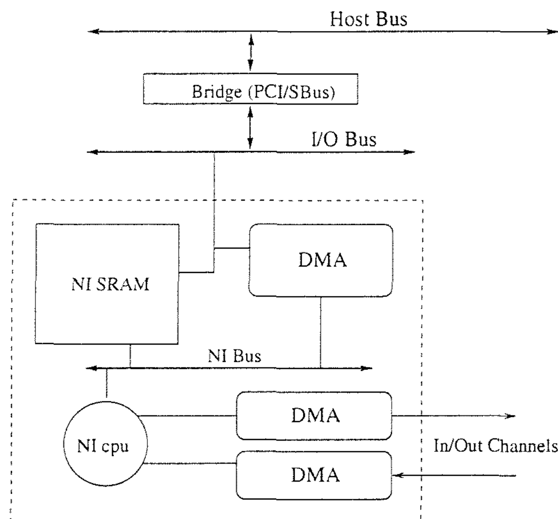

Figure 2.5: Myrinet NI block diagram.

port of the switch and is stripped off by the switch en route. At the host end the remaining leading header byte identifies the type of packet. Myrinet packets have arbitrary length payload and can encapsulate other types of packets, e.g. IP packets, without an adaptation layer. The CRC is computed for the entire packet and recomputed on each link (because the packet header is modified). The Myrinet network uses multi-port switches (4, 8, 16, 32-way port) of pipelined crossbar type with blocking cut-through (wormhole) routing similar to Intel Paragon and Cray T3D MPP systems [22, 45].

The network provides in-order delivery, error detection is done by hardware-computed CRC field and erroneous packets are dropped. The flow control mechanism used to block packets on busy channels is accomplished with acknowledged byte-control symbols (GO, STOP, etc) injected into the opposite-going channel of the link.

Each host interface card has its own programmable network interface processor (a 32-bit control processor called LANai), 1Mbyte of fast SRAM used to hold network buffers and instruction code for the network processor and three DMA engines, one for the outgoing channel, one for the incoming channel and the third one (via PCI or S-Bus bridge) for the host main memory. Data transfer between the host memory and the Myrinet interface can be done either in DMA mode or using Programmable I/O instructions mode [17].

Control and access of the network interface is based on a flexible scheme with a Myrinet Control Program (MCP), the device driver and the OS. In this way the interface is very flexible and can easily implement Data Layer (Level 2 of the ISO reference model) services for existing higher-level network protocols as well as providing an excellent infrastructure for experimental protocol implementations.

Scalable Coherent Interconnect (SCI) The SCI (ANSI/IEEE standard 1596-1992) is not a traditional network, rather a bus defining an electrical interconnect standard for internal

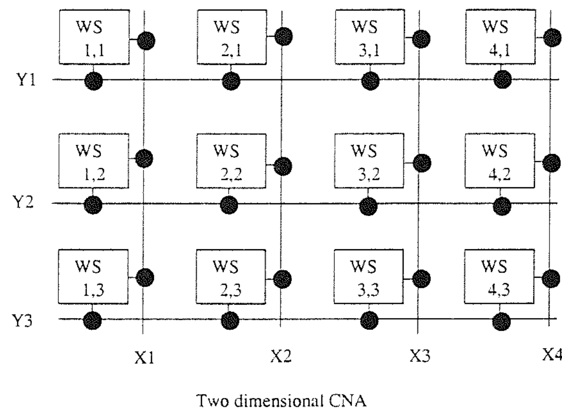

Figure 2.6: 2-D Concurrent Network Architecture source [111]

processor-memory connections within computers allowing 1 Gbyte/s transfers over distances of up to 10m and a fibre-optic serial interface that equals FCS full speed up to one kilometre [119]. SCI-based systems provide hardware supported Distributed Shared Memory (DSM) support with low-latency remote memory access (remote write of 100 ns and remote read of 5 μs on SMiLE PC cluster [112]).

Serial Express The 'SerialExpress' is a draft standard (IEEE P2100) based on a bus architecture similar to SCI and Serial Bus. Its aim is to provide a low-cost technology independent interconnect for system area networks [119]. The protocol supports peer-to-peer communication modes as well as isochronous or asynchronous traffic in real-time mode. SerialExpress is independent of the transmission media, and can operate with a low-cost serial-link technology (e.g. 1Gbit/s data rate).

Concurrent Network Architecture and Channel Bonding A technique to improve network bandwidth is the concurrent use of multiple network paths. Hipper et. al. [111] suggested an alternative network architecture for clusters that increases communication performance with the introduction of a structured network consisting of several parallel and independent LAN communication channels in a flexible topological structure. Communication between two nodes can be direct, e.g. sharing of a common communication channel, or via another node acting as a router. In this way the aggregated communication network throughput among nodes is increased with the number of independent networks. Consequently contention and scalability of the system is improved as well. The Beowulf class project [208] (see Chapter 3.6) has used a similar multiple Ethernet configuration scheme to increase communication throughput and scalability known as "channel bonding", this technique joins multiple low-cost networks into a single logical networks with higher bandwidth. Channel bonding and load balancing is implemented at the device queue layer below the IP protocol layer which makes it transparent to the application layer. This method was successfully tested with 10 Mbit/s and 100 Mbit/s Ethernet channels in many Beowulf clusters.

An experiment on a 16 node Beowulf cluster with two and three way 10Mbit/s Ethernet channels per node configurations achieved respectively throughput of 1.7 Mbyte/s (68% of peak) and 2.4 Mbyte/s (64% of peak) for 8Kbyte token exchange [208].

In this way the performance of the cluster is improved, especially for application algorithms that exploit the underlying network topology. Sophisticated network topologies such as hypercubes, mesh, or toroidal could be easily implemented and accommodate a large number of nodes. However, such a parallelism at the network level does not improve latency. In practice there is a compromise on processing nodes that serve as networking nodes between computation overhead of the extra routing and the remaining compute capability of the processor itself. Other disadvantages of such architecture are the multiple cost of multiple number of networking e.g. extra Network Interface Card (NIC) per workstation, and the limited number of interface slots, (e.g. for the PCI bus), available on workstations.

2.5 Case Study: Internetworking with Ethernet

The 10Mbit/s Ethernet, though relatively slow in comparison with new gigabit networking technologies such as Myrinet [22], is still the most-widely used LAN technology to interconnect local distributed systems and clusters of workstations in organisations. Additionally most of the clusters at the University of Southampton currently use 10 Mbit/s Ethernet for the interconnecting network. Thus in order to understand the interconnection of our clusters, a further study of the basic characteristics of an Ethernet interconnection network (such as latency and bandwidth) is required. The results will be used later as a basis to compare with communication libraries, such as MPI.

Networks can be characterised by two aspects, latency and bandwidth. Communication latency is the end-to-end transfer time of a message from user space to user space (see equation 2.2). For small messages the protocol computation is the dominant factor rather than the actual hardware latency. Throughput denotes the amount of data can be transmitted over a time period. In this case study, latency is denoted as the zero-size-message round-trip time and bandwidth the throughput achieved in the transmission of large messages. The maximum theoretical throughput of the 802.3 Ethernet standard is first calculated and then compared with the actual latency and bandwidth on an example network e.g. a cluster of Sun workstations. Parameters that can affect measurements in our tests are taken into account for each platform in order to analyse the results better.

Communication libraries in parallel systems are usually based on the underlying interconnection network (this will be explained in greater detail in chapter 4). A small ping-pong program [186] was written to make an estimation of the actual throughput of the OS and the network (a 10 Mbit/s Ethernet segment). Measurements were also taken on Sun (ULTRA SPARC) and Pentium-Pro workstations and Pentium-Pro workstations on a Fast Ethernet network. The software was designed to measure bandwidth and latency of the Berkeley sockets only. Figure 2.7 and Table 2.2 give numerical results for the latency and the bandwidth measured on various clusters using TCP/IP sockets. For small messages the start-up time t_s can be an order of magnitude larger than the actual time required for the transmission t_w (i.e. $t_s \gg t_w$

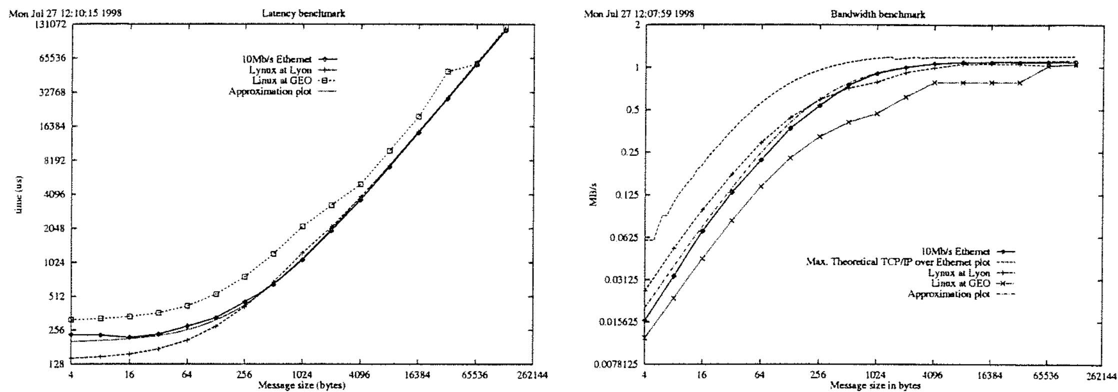

Figure 2.7: Communication bandwidth and latency measurements over 10 Mbit/s Ethernet, together with theoretical estimation

from $t_n = t_s + t_w n$ in section 2.4.1).

Applying a linear regression fitting on the experimental results of Fig. 2.7 using the model of equation 2.2 we can find the coefficients of t_s and t_m as shown in Table 2.2. A simplified approximation of the equation 2.2 coefficients can be derived from the Table A.1 in Appendix A which gives similar results for 10BaseT networks.

$$t_s \approx zero\,length\ message \tag{2.3}$$

$$t_w \approx per\,byte\,transmission\,cost \approx \frac{8(bit/byte)}{10(Mbit/sec)} \tag{2.4}$$

From Appendix A the average overhead n_{ov} for each Ethernet packet is approximately ≈ 55 bytes for each frame transmitted hence equation 2.2 becomes:

$$t_n = t_s + t_w(n_{ov} + n) = t_s + t_m(55 + n) \tag{2.5}$$

Applying a linear-fit regression for the Ultra-SPARC cluster measured points yields a zero-length latency of 200 μs and $t_w \approx 0.9$, thus equation 2.2 becomes: $150 + 0.9(55 + x)$. The transmission cost per byte in our approximation includes buffer handling management so its calculated value is bellow the real byte transmission on the channel which for a 10Mbit/sec Ethernet is 1.25 Mbyte/sec. Figure 2.7 shows the plots of this approximation compared with the measured data.

The non-deterministic nature of the Ethernet channel as well as the way the OS handles transmission and reception of network packets, has the potential to cause variations in measurements. In addition, Ethernet packet fragmentation affects the latency for packet sizes above the maximum Ethernet packet size (>1460 bytes).

The results illustrate a detectable difference between the actual performance and the theoretical one, this is due to the operating system overhead and the non-deterministic nature of the channel. For large size messages most of the clusters approach the maximum theoretical barrier

Table 2.2: Latency and bandwidth characteristics for different networks of workstations

Cluster Configuration		Latency (μs)	r_∞ (Mbyte/s)	$n_{1/2}$ (bytes)	t_s	t_w
Sun (Solaris)	UltraSPARC	233	1.084	>256	152	0.987
PC (Lyon)	Pentium Pro	144	1.042	<200	137	0.916
PC (GEO)	Pentium Pro	316	1.041	1K	1074	0.819
PC (FastEthernet)	Pentium Pro	90	5.40	<1.5K	196	0.175
SGI O2 (FastEth.)	R10000	368	12.06	<8K	496	0.079
150+0.9(55+x)		200	1.10	222	-	-

of the Ethernet channel. In terms of latency and half performance bandwidth point $n_{1/2}$[7] there are substantial differences among clusters. Latency and bandwidth both strongly depend on the hardware performance as well as the actual implementation of the network protocols.

The Solaris cluster for example gives relatively-smooth results which are closer to the theoretical limits than the other cluster configurations most of the time. The reason for this is the well-tuned network protocol implementation together with balanced underlying hardware achieved by Sun workstations. Conversely unbalanced PC-based systems with different software implementations can degrade the effective network performance.

Similar results could be achieved with a Fast Ethernet network cluster. The main difference with the 10Mbit/s Ethernet is the medium transmission rate which decreases the "time of flight" in equation 2.2 but does not change significantly start up processing overheads. Figure 2.8 shows the result of the latency and bandwidth measurements for the Fast Ethernet link. The Ethernet link of 10/100 Mbit/s. Effective bandwidth in tests is limited asymptotically by the Ethernet link barrier only for large size messages.

For short messages the computation overhead dominates the results. Saturation of the communication channel within the range of small messages is relatively low despite the use of fast processors [151] i.e. the half performance message length for Fast Ethernet configurations is relatively large and makes poor utilisation of the effective bandwidth for short messages. Remarkable performance variations can be seen for clusters of PCs due to their difference in hardware and software configurations. Least square parameter fitting of equation 2.2 is more accurate for short messages than large messages for network interfaces which use advanced hardware features i.e. DMA engines.

2.6 User-space Protocols

Advances in processor and network technology frequently improve bandwidth but not end-to-end latency, largely because the communication software overhead is several orders of magnitude larger than the hardware overhead [169]. In traditional network architectures the host processor

[7]The $n_{1/2}$ or *half performance point* is the message size at which the bandwidth is equal to the half of the maximum assymptotic bandwidth performance achieved on that system [114].

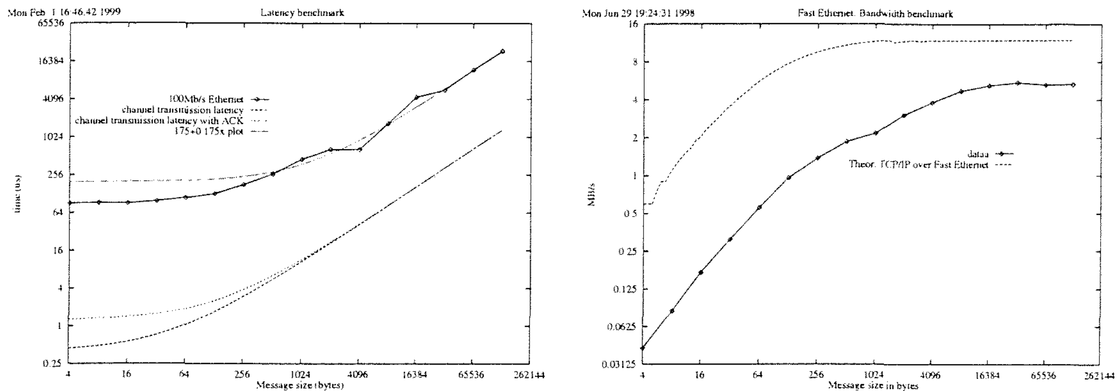

Figure 2.8: Communication bandwidth and latency measurements over 100 Mbit/s Ethernet, theoretical latency of $196 + 0.175 \cdot n$ approximation

and the OS control the communications hardware, inevitably introducing a serious bottleneck which constrains performance [55]. This results in additional context switching between applications and the OS for each system call and additional copy operations between address spaces and buffer management [170, 9].

The actual network traffic over a LAN on which workstations share local resources and control usually consists of many small packets. Extensive measurements even on network-bandwidth-intensive applications [176] demonstrated that 95% of the packets in the trace are less than 200 bytes, while the mean packet size is less than 400 bytes. For packets of this size the dominant transport cost is not the bandwidth but the set-up overhead. Small packet sizes with a latency overhead in the order of millisecond cannot be *hidden* using conventional programming techniques such as overlapping or pipelining. In other words, performance is effectively bounded by the interaction between the kernel and the user-space rather than the available communication bandwidth.

As a result, clusters of powerful workstations still suffer a degradation in performance even when a fast interconnection network is provided. In addition, the network protocols in common use are unable to exploit fully all of the hardware capability resulting in low bandwidth and high end-to-end latency. Parallel applications running on top of communication libraries (e.g. MPI, PVM, etc.) add an extra layer on top of the network communication stack, (see Fig. 2.9) [152].

Modern improved protocols for workstation clusters and LANs are designed to avoid the time-consuming communication path (application-kernel-network device) identified above [182, 187, 134, 229]. Additionally these protocols exploit advanced hardware capabilities, (e.g. network devices with co-processors and enhanced DMA engines such as Descriptor-Based DMA[8]) by moving as much functionality as possible into the hardware device. Users can write

[8]Descriptors are data blocks typically with 16- and 32-bit fields that serve as a simple instruction set for implementing DMA transfers. Hence the DMA engine is enhanced to execute efficiently and autonomously, i.e. without interrupting the CPU, sequences of data transfers.

The 3c905 NIC supports both Descriptor-Based DMA and CPU-driven DMA transfer modes.

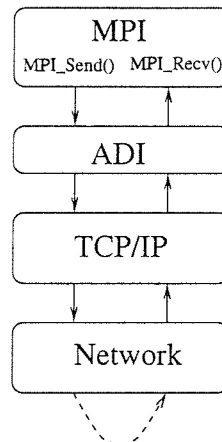

Figure 2.9: MPI on top of the TCP/IP protocol stack

communication libraries and applications which interact directly with the network interface avoiding any system calls or kernel interaction. In this way the processing overhead is considerably reduced providing both reduced latency and higher throughput [21]. Such an approach can however introduce a drawback in terms of functionality (i.e. reduced system security and integrity together with removal of the protected multiprogramming communication, because the network interface is shared now between the OS and network applications) [184, 182, 134, 236]. Contemporary OSs can overcome these disadvantages by providing virtual memory protection mechanisms or virtual network interfaces, but there is always a compromise between functionality and performance. There are several approaches for user-space protocols (sometimes referred as user-level protocols in the literature) which provide low-latency and high bandwidth on fast networks [9]. Most of them provide direct user access to the network interface support for SPMD models and use commodity workstations (e.g. PCs) together with a standard OS (e.g. NetBSD, Linux, NT). The following paragraphs present the current state of the art in fast protocols.

Basic Interface for Parallelism [183] (BIP) this approach implements a high-speed protocol Application Programming Interface (API) run on a Myrinet board. It eliminates all system calls by implementing zero-copy protocols at the user-space and provides data transfer only (in a FIFO order), while not providing protection or multiprogramming. Latencies of the order of a few microseconds are achieved and the network channel can be fully filled with data (126 Mbyte/s at user-space throughput). Section 2.7 in this chapter examines a BIP system in more detail.

U-Net [229] provides a network protocol stack at the user level, enabling applications to access high-speed communication devices directly. A virtual Network Interface with memory management capabilities provides protection among processes without any kernel intervention. The architecture is very flexible and can adapt traditional protocols such as TCP, UDP or even Active Messages efficiently. Implementations of the U-Net protocol

achieve performance close to the hardware limits for Fast Ethernet or ATM communication channels.

ActiveMessages(GenericAM,AM-II) [139] represent a *RISC* approach to *one sided* communications by providing a simple set of communication primitives based upon request and reply active messages, which provide a substrate for higher-level communication libraries or parallel-language compilers. In this system the header of each message contains control information for the user-space routine responsible for extracting the message from the network. Messages are delivered in FIFO order and there is an option for multiprogramming support. AM-II also provides "put" and "get" remote-memory communication primitives.

FastMessages(FM) [170] is a high-speed Active-Message-like system that delivers low latency and high bandwidth for short messages over a Myrinet network. FM provides in-order message delivery with flow control and packet retransmission. Each message carries a pointer to a function that consumes data at the receiver end. The latest versions of FM support the SPARCstation SBus as well as the PCI bus on a PC cluster together with support for multiprogramming.

Fast_Sockets [187] exports the Berkeley Sockets programming interface using a high-performance protocol which collapses and simplifies protocol layers by transferring some of the protocol knowledge required into user-space programming.

VirtualMemoryMappedCommunication(VMMC) [56] model allows an application process to access (directly or through user-space transfer operations) the memory of another process running on any node in the system within a protection domain, hence it can be used for both message-passing as well as shared memory implementations. The VMMC model supports protected user-space communication while multiprogramming is possible as well.

VirtualInterfaceArchitecture(VIA) [48, 67] is an attempt to standardise a user-space specification protocol for clusters and system-area networks (promoted by Intel, Microsoft and Compaq). VIA defines a set of functions and data structures with associated semantics for moving data among remote processes memory. Processes open connection-oriented Virtual Interfaces (VI) that represent handles into the network, which can be seen as an extension to the U-Net end points, on which messages are sent to or received from its remote VI. VIA provides direct transfers between local and remote memory similar to AM-II "puts" and "gets". Protection in VIA is ensured by each process specifying the available memory areas for remote DMA operations. A defined quality of service (QoS) can be supported as well, although reliability of communication is not mandatory.
There is still no NIC hardware support (end of 1999), since the release of the VIA 1.0 specification. Experimental kernel-emulated VIA support with improved characteristics for Fast Ethernet has been implemented.

All protocols discussed above have one common feature, communication interchange is done via the I/O programming mode at the host computers. This means that latency can be several cycles long because of the I/O commands needed (e.g. programming the NIC) and the only way to improve it is to reduce the number of instructions. For a fast interconnection network such as Myrinet latency in the order of 5 microseconds can be achieved. At this level additional functionality can add substantial burden and increase latency. Bilas et. al. [9] in their quantitative study of user-space communication discuss the difference between functionality and efficiency for fast protocols.

2.6.1 "Careful" Protocols

According to [158] careful protocols offer a reliable service which is equivalent to the reliability provided by the lower-level protocol service. Many new network technologies provide services with "negligible small" probability of lost, damaged, duplicated, or out-of-order packets, While the description 'negligible small' is application dependent, the option of enhancing reliability could be addressed at the end-to-end level. Under such an assumption the implementation of communication protocols is considerably simplified, because no retransmission, or duplicating-detection, or out-of-order schemes are required and the flow-control issue can be addressed with a simple request/response rendezvous scheme. This will provide interconnection with low latency features in a similar way to typical MPP interconnection, which is considered reliable by design.

2.6.2 Light Weight Protocols

Another interesting class of fast protocols is the "efficient OS support" approach sometimes known as "light weight protocols". The communication protocol again is carefully simplified and supported on a small set of flexible and efficient low-level communication primitives by the OS kernel [39, 38]. A LAN-span communication protocol can have a simple host naming scheme, it should also minimise temporary data movements and apply various pipelining techniques between consecutive communication stages. Intervention with the OS should also be minimised and use light-weight system calls. The notification policy upon message arrivals has to be efficient and adaptive between polling and interrupt based mechanisms [40, 158, 190].

High-level communication protocols are built efficiently on top of these communication primitives. The advantages of this kernel-level approach is enhanced flexibility of the network protocol but at the same time protective multi-user and multiprogramming features which can be supported on inexpensive commodity hardware (e.g. Fast Ethernet NI) in contrast to a restricted user-level approach. Examples of light-weight protocols are the GAMMA project, the Beowulf clusters, PARMA, etc.

Genoa Active Message MAchine (GAMMA) The GAMMA project is an example of a light-weight protocol currently running on a 100Base-T cluster of Pentium PCs running Linux. The communication mechanism is based on the concept of Active Ports, each process can activate and use up to 256 ports for output, input or input/output to send or receive messages.

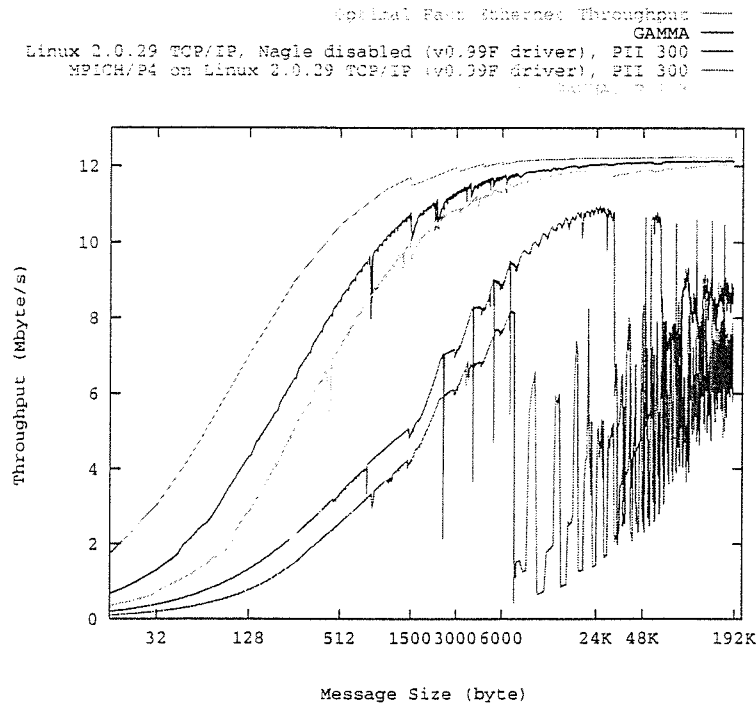

Figure 2.10: GAMMA throughput taken from [40]

Active Ports are implemented with light-weight calls, (i.e. system calls with no intervention of the scheduler upon return), as well as "fast interrupts" and are effectively between the kernel-level (at the NIC device driver) and the user-level communication library.

Message sending is accomplished with a zero copy mechanism that transfers data from the user space to the network interface by splitting the message into a sequence of Ethernet frames of size 60-1536 bytes. At the receiving end an interrupt handler is launched which copies the network interface receive queue to the memory space of the receiver process. Frame headers provide the necessary information needed for correct process space buffering [37]. The GAMMA protocol mechanism allows multi-user access of the communication path as well as the use and existence of GAMMA and IP datagrams.

Active Ports allow a zero-copy protocol implementation with the very low latency feature below 13 μs and maximum throughput of 12.2 Mbyte/s with half-bandwidth message size of just 192 bytes [39]. The Active Ports communication mechanism does not provide explicit acknowledgement and flow control scheme (which is usually not critical as the Ethernet channel is the slowest part of the communication path), Ethernet frames with an invalid CRC are discarded. However the protocol is allowed to pass error packets to the application layer to deal with.

Table 2.3: User space protocol characteristics

	AM-II	BIP-0.92	FM-2.02	VMMC-2
Comm. Model	RPC	Send/Recv	Send/Recv	Direct Depos.
Control/Data transfer	Combined	Data Only	Combined	Separate
Buf. Overflow	Prevented	Data Loss	Prevented	Impossible
Net Errors	Tolerated	Catastrophic	Catastrophic	Tolerated
Net. Manag.	Dynamic	Static	Static	Dynamic
Send Data	PIO	DMA+ v2p transl.	PIO	DMA+ v2p transl.
Recv Data	DMA+ copy	DMA+ v2p transl.	DMA+ copy	DMA+ v2p transl.
Translation	DMA	user	DMA	UTLB
Protection	Copy	None	None	PIO+copy
Notification	Polling	None	Polling	N/A
Latency	21 μs	6 μs	11 μs	11 μs
Bandwidth	31 Mbyte/s	121 Mbyte/s	78 Mbyte/s	97 Mbyte/s

2.6.3 Semantics of User-Space Network Protocols

Fast communication protocols are necessary for distributed and clustered systems that can deliver the real potential of the hardware performance at the application level. Existing communication protocols provide a level of functionality which is not necessary on LAN and SAN systems. At the top end of the application level the communication library services required can be summarised to:

- Message delivery between the sender and the receiver
- Message ordering (messages should be delivered in the order of transmission)
- Deadlock and overflow safety
- Reliable delivery

At the network side of the communication sub-system the required features are typically:

- Arbitrary delivery order
- Finite buffering
- Fault detection (but not a fault-tolerance mechanism)

The software messaging layer has to bridge the gap between communication services and hardware features in the most efficient way possible [122]. Hence the software protocol has to

sequence and reorder packets, address flow control and buffer management as well as acknowledging data, etc.

A common feature among user-space protocols is the diversion of OS from the critical communication path and the avoiding of redundant memory copying. The cost of memory copy is regarded as relatively high because memory speed and memory bus bandwidth have not improved significantly over the past few years [160, 176]. However, bypassing the OS removes as well the responsibility of the communication either towards the application level or towards the network interface. User-space protocols hence being simple have to use as much of the network interface hardware features balancing functionality and performance. User-space protocols in additional have to address several common network protocol design problems efficiently e.g. data transfer, reception of message, flow control, etc.

Data transfer for zero-copy protocols involves three stages, a host-to-interface transfer, an interface-to-interface transfer and an interface-to-host transfer. The host-to-interface transfer to the network interface or the I/O subsystem should have direct memory access capabilities [197, 18] as well as programmed I/O (copying). DMA techniques have the advantage of decoupling the CPU from the data transfer and also prevent pollution of the cache. Furthermore DMA transfer can enhance overlapped activities within a node. Host-to-interface transfers via DMA require the translation of the process' virtual address to the physical address on NI in a protected manner.

The address translation mechanism is a key requirement for such operation and the NI is expected to provide it. In addition coherency with the cache has to be supported without the involvement of the CPU [17]. Asynchronous DMA transfers may require locking of memory pages in their address space to avoid OS involvement. In order for the system to support multiple sends and receives the mechanism has to define message segment areas for each process to send or receive messages. The address translation mechanism has to be extended to *<segment_id, segment_off set>* and provide protection among multiple processes network access [197] or alternative use a scheme of Translation Look-aside Table (TLB) e.g. the U-Net/MM [235]. Many zero-copy protocols compensate the DMA start-up cost by using Programmable I/O for short messages (e.g. <256 bytes, and providing an adaptive cut-through message delivery mechanism [243]. Programmable I/O transfer does not require translation or protection mechanisms.

Control transfer mechanisms and notification mechanisms can be either by polling or by use of interrupts, depending on the network interface. The first method is fast which helps to keep resources busy, while although interrupt handling is generally more expensive in terms of time it allows multiprogramming. The network channel in fast networks is highly reliable so many user-space protocols do not provide retransmission, buffering or acknowledgement mechanisms. Erroneous packets are usually dropped or notify the application layer. At an extra cost some protocols can also provide reliable communication e.g. AM-II.

Regardless of the reliability of the system, overflow control is another issue network protocols are required to address. Many protocols rely on the network interface hardware flow control mechanism, other on the preallocation of buffers or the of a rendezvous-style communication where the receiver posts a receive request for large messages. Multicasting and broadcasting

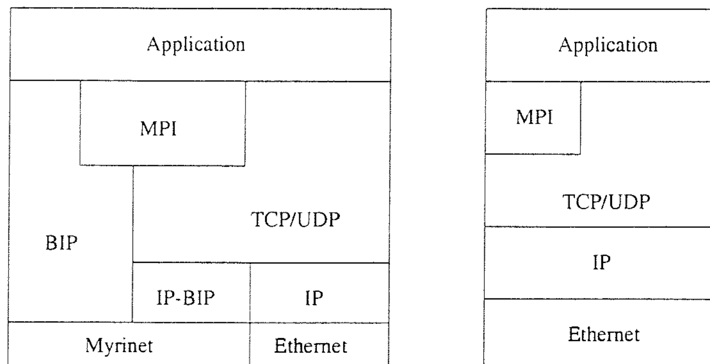

Figure 2.11: The BIP protocol stack approach (left) compared with the classical TCP/IP approach (right)

for these protocols is usually done at the software level. Network management of user-space protocol can be either dynamic or static.

2.7 The BIP Zero-Copy Protocol Approach

The Basic Interface for Parallelism is an API for System Area Networks, capable of exploiting the fast communication links among nodes provided by network technologies such as Myrinet [117]. The interface has been designed to deliver to the application layer the maximum performance achievable by the hardware. Important features of this protocol include direct interaction with the network board at the user-application level, elimination of system calls, efficient use of memory bandwidth and a zero-copy protocol [183].

The protocol takes advantage of the network-board processor, memory and DMA engine in order to perform fast pipelined data transfers. BIP messages can be routed through multiple Myrinet switches which provide services equivalent to the OSI 3 level. The Myrinet network has a very low intrinsic error rate, therefore in order to reduce overheads the BIP protocol itself does not provide an error correction mechanism. However a Myrinet error detection mechanism is provided and in principle could be used from the application end or higher protocols to provide limited error correction [182].

In order for BIP to achieve its maximum performance, the network board management of a node has to be dedicated to the application The NI registers and its memory regions have to be exposed to user-level access with no protection. Thus other applications cannot share or use the BIP protocol on the same node concurrently. However applications running on other nodes can share the Myrinet network. Hence the operation mode of the cluster is similar to the batch processing mode of MPPs. The NIC DMA engine is capable of addressing any location in memory for direct data transfers from and to user space. This capability means that BIP requires system support for coherency between DMA memory access and the processor cache, hence the OS together with the system bus has to be able to translate between physical and virtual addresses.

Table 2.4: Latency and bandwidth performance on a Myrinet cluster using BIP

Message Size (bytes)	Latency μs	Bandwidth Mbyte/s
0	11	-
128	20	8.5
256	48	5.7
512	54	10
1K	60	18
8K	142	57
32K	353	90
128K	1160	108
512K	4370	114

Although the Myrinet network can handle arbitrary long packets the BIP protocol uses an adaptive four-staged pipelined transmission mechanism to maximise the efficiency of its DMA engines along the communication path. Messages are fragmented into packets of equal size and each packet transmitted in sequence through the pipeline. The host processors at both ends are only involved during the transfer initialisation to provide storage information about the message. An adaptable transferring policy is used for short and long messages to maximise efficiency (memory copy, PIO, or DMA transfer).

BIP services are strongly oriented for parallel applications to provide an intermediate layer of functionality for higher-level protocols e.g. TCP and MPI. The BIP interface provides stand-alone communication primitives for blocking or non-blocking calls with "loose" rendezvous semantics, hence overlapped computations and communications are possible.

The current implementation of BIP (v 0.93) runs over a homogeneous cluster of six x86/Linux workstations, linked by a Myrinet network with a maximum throughput of 132 Mbyte/s (using the Myrinet/PCI board). The Myrinet switch is a wormhole switch contributing less than 100 ns latency overhead in the absence of contention. The BIP cluster test-bed is flexible enough (see Fig 2.11) to configure the communication API as:

1. TCP/IP API over Ethernet (Ethernet configuration)
2. TCP/BIP API over Myrinet (Myrinet configuration)
3. BIP API over Myrinet (BIP configuration)

User applications for the first two API modes run transparently on both configurations i.e. no change is required either in the MPI implementation or the MPI executables. In the third mode, where MPI runs directly on top of BIP, changes in the Abstract Device Interface are necessary hence all MPI programs require re-compilation with the new libraries. The easiest way to do this for the BIP protocol was to provide a new Channel Interface without invoking the kernel which ensures a zero-copy protocol at the MPI level.

Table 2.5: Ping-pong test results on various communication libraries

Configuration	min Latency	max BW	$n_{1/2}$
TCP/IP sockets	144 μs	1.06 Mbyte/s	300
TCP/BIP sockets	84 μs	23 Mbyte/s	1.5 K
BIP sockets	6 μs	121 Mbyte/s	3 K
MPI over TCP/IP	280 μs	1 Mbyte/s	300
MPI over TCP/BIP	171 μs	17.9 Mbyte/s	1.5 K
MPI over BIP	11 μs	114 Mbyte/s	8 K

Table 2.5 shows the latency and bandwidth graphs for different protocol stack configurations. A noticeable discontinuity at message sizes of 256 bytes reveals the different semantics between short and long messages transmission modes.

2.7.1 Future Trends of Network Subsystems

Over the past decade the concept of networking has become an increasingly fundamental part of computing: in the words of Sun[9] "the network is the computer". Despite the acceptance of the above trend network subsystems have not been perceived as an integrated computing subsystem from both the hardware and the software perspective.

According to Hill et. al. [160] the future of network interface is tightly coupled with memory systems and processing units and therefore NICs should be seen as a vital computer subsystem and not as an 'ordinary' peripheral. Cranor et. al. [50] describe an efficient network architecture interface that integrates communication functionality into the processor which can provide fast event processing and high performance data transfer. For the existing network architectures, the user-space protocols on clusters and distributed systems have demonstrated the potential to reduce the communication bottleneck and achieve high throughput. Table 2.3 shows the system I/O bus throughput could limit the effective network bandwidth (e.g. the 32bit PCI bus is restricted to a throughput of 132Mbyte/s while existing communication links are capable to deliver 1.28Gbit/s that is equivalent to 160Mbyte/s point-to-point data transmission rate [117, 23]). In addition, the choice for data handling and moving to and from the host memory via the PCI I/O bridge is becoming critical and can affect performance of common network traffic patterns [243]. As microprocessor and network technology advance towards Gigahertz clock rates and tens of Gbit/s bandwidth[10], the I/O bus bottleneck will become the next critical communication barrier. Mukherjee et. al. [161] propose that a network interface should be attached on the system bus (processor data path, the cache bus, or the memory bus) in a similar way to MPPs.

Access to the NIC will be treated as regular memory access rather using I/O operations through the OS. There are many advantages of attaching a device to the system/memory bus,

[9] Advertising slogan used by Sun Microsystems.

[10] Sonet (OC-192) has a throughput of 10 Gbit/s Wave Division Multiplexing (WDM) can deliver aggregate throughputs of 200 Gbit/s [44].

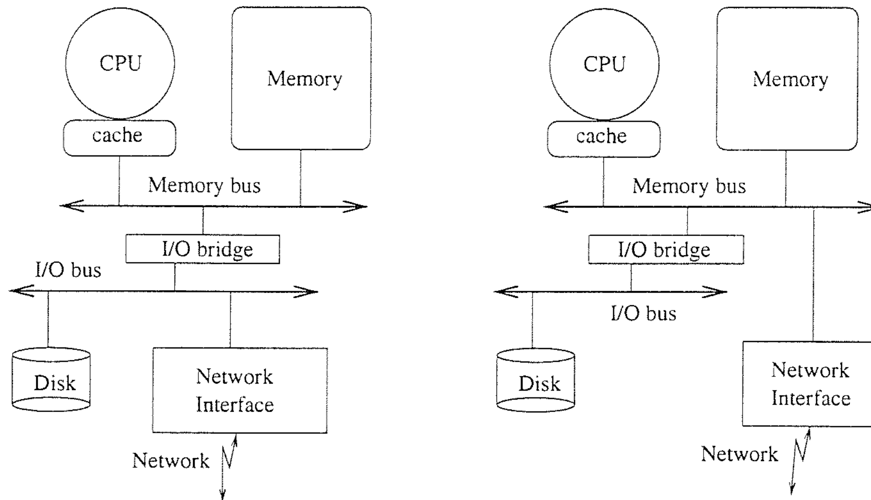

Figure 2.12: Network Interface attachments on a workstation system

latency and throughput of data transfer is becoming significantly better (latency at the order of *ns* bandwidth greater than 2.6 Gbyte/s are typical values system bus). Network protocols then can be simplified even more, as the NI will inherit all memory bus virtual-to-physical-address translation and protection, existing mechanisms together with cache coherency strategies to avoid side-effects [160, 36]. Host applications will then be able to directly access the network interface without compromising performance. Among the drawbacks of such a proposal are possible changes on the host (system and OS) to adapt the NI and the lack of standardisation among system buses.

One could argue that such a configuration of clusters and distributed systems is very similar to current MPPs. The difference is that MPPs will probably continue to be based on proprietary components while system area networks are likely to be based on standardised commodity COTS (Commodity Off The Shelf) components.

2.8 Summary

This chapter has examined the fundamental characteristics and semantics of intercommunication subsystems that are used in networks of workstation. "Traditional" communication protocols written with WANs and LANs features in mind introduce long overheads and fail to deliver high throughput on modern high-speed links. Ciaccio [40] points out that communication software is substantially older than network interfaces but relatively young compared with the history of operating systems.

A new generation of network protocols such as user-space, "careful", or light-weight protocols, built around the concept of system area networks and clusters, demonstrate improved communication performance characteristics at both network level and application level. Emerging network technologies such as Fast Ethernet, ATM and Gigabit Ethernet are now replacing the old 10Mbit/s based networks in LANs and will offer new communication features (e.g.

multicast, QoS, etc). In order to sustain high throughput among the nodes of a distributed system, high-speed network technologies require modern implementation of network protocols in order to exploit their advanced features. In addition, such protocols need to address any unnecessary interaction with the OS (e.g. user space protocols) and reduce the computational overhead. The use of fast protocols such as the BIP have demonstrated that substantial performance improvement is possible. Workstation clusters need to adopt these low latency and high bandwidth interconnection subsystem in order to perform as a viable competitive parallel platform.

Chapter 3

Clusters of Workstations

3.1 Introduction

This chapter reviews concepts and issues from Distributed Systems (DS) and parallel systems which are directly applicable on workstation clusters. In addition later sections of this chapter will classify the main attributes and characteristics of the workstation cluster platform compared with MPP systems which will assist later in understanding the underlying performance.

Advances in computer systems and computer network technologies over the last twenty years have gradually allowed different computers to share network resources and facilities (a concept known as "distributed computing"). A "distributed computing system" (DCS) is a collection of autonomous computers linked by a communication network with software that provides integrated computing facilities. Tannenbaum [221, 223] defines a distributed system as "a collection of independent computers that appear to the users of the system as a single computer".

There are several ways in which a distributed system can be classified. Using Flynn's [75] taxonomy a distributed system is certainly an MIMD machine, and according to the way nodes are coupled a distributed system can be either tightly coupled or loosely coupled [143, 32, 49] or according to their programming model client/server or processor pool [224].

Workstations clusters (otherwise known as networks of workstations or simply clusters), [6, 171] are a type of parallel (or distributed) system that consists of a collection of interconnected whole computers utilised as a single unified resource [178]. Clusters or Networks of Workstations (NOWs) provide a way to build powerful, cost-effective parallel machines by using standard off-the-shelf computers and networking technology [241]. NOWs use the same computational model as MPPs (message-passing) so they can provide an alternative test-bed platform for parallel applications.

As a distributed system, clusters have the same difficulties as a DCS has in terms of software tools and standards. Administration of a cluster is individual for each machine of the system which is neither time-efficient nor cost-effective. Software applications and tools for clusters are limited and require the support of a robust programming model (e.g. message-passing) and a run-time system. Essential issues and concepts of distributed systems become directly

applicable and could determine the performance of clusters as well.

The following section examine issues of distributed systems such as remote procedure calls, synchronisation, the client-server paradigm and distributed computing tools which are also fundamental for clusters or workstations.

3.2 Basic Distributed Computing Primitives and Concepts

In a non-distributed computing model, all components of a user application (such as user interface, computational function and storage) are integrated on the same node. A distributed computing model can transparently migrate these parts onto different computational nodes [188]. In this way a better resource sharing and usage management can be achieved among the users of the system [49]. A DCS is composed of the hardware (which can be heterogeneous or homogeneous), the software (the Operating System, distributed or not, tools, utilities, etc.) and the network subsystem which interconnects all nodes together. According to Lamport [129] a distributed computation is determined by the type and the relative order of *events* occurring at the processes. Events are specified as: a *sent* event which causes a message to be send, a *receive* event causes a message to be received and update the local state and an *internal* event which cause only a change of the local process state.

Over the last few years, along with hardware component improvements, most of the elements of distributed systems software have also became standardised [188]. The key attributes of a distributed operating system are defined by Coulouris et. al. as [5, 49]:

- Transparency
- Resource sharing, coordination
- Support of an arbitrary number of systems, scalability (processors and processes)
- Openness, modular design of physical architecture (homogeneous or heterogeneous)
- Message passing facility through a shared communication system
- Concurrency and system control for all distributed hosts
- Fault tolerance

Most of these characteristics are also key points for workstation clusters. A fundamental issue in any distributed system is transparency as specified by the Advanced Networks Systems Architecture (ANSA) [7, 49]. The system has to appear to be a single computer to both application programs and users. Tasks should be executed consistently and effectively regardless of the location of the hardware, the software, or the system's structure.

A DCS has to cope with any changes made "on the fly" therefore the issue of flexibility is important as well [221, 206]. Scalability implies that a system should not be restricted to a small number of nodes and the potential should exist to extend to a large number of nodes without any substantial difficulties or performance degradation. Scalability should also apply to

application software. Increasing the number of nodes should increase the reliability of the system (or at least not decrease it). Among the drawbacks of DCS are the additional complexity of the software and the absence of an accepted distributed operating system. Sharing resources over the network adds extra overhead for applications which could decrease their performance. The use of threads and caching is one technique that has the potential to reduce this performance loss [188]. Distributed systems in many institutes and companies have now effectively replaced old-fashion centralised mainframe computers, as their modular concept provides better availability and scalability.

3.2.1 High Level Communication Primitives and Concepts

Communication among nodes is a vital part of any distributed system and workstation cluster. Chapter 2 discussed the lowest part of the communication subsystem while this chapter will review the highest part of the communication subsystem along with the mechanisms a distributed system uses for its interprocess communication. At this higher level communication primitives for DS and workstation clusters can be either blocking or non-blocking, buffered or unbuffered [11, 49].

In blocking communication mode the process which sends a message is blocked (i.e. suspended) until the corresponding receive is executed and only then is data transferred [162, 228]. The main advantages of blocking calls are simplicity and determinism as data is transferred only when both the source and destination memory addresses are known and there is no need for buffering. The exchange of a message represents a synchronisation point for a programming model but could also lead to deadlock. Use of light-weight processes or threads can enhance parallelism e.g. threads can be used as a concurrency mechanism with a low system overhead [188]. When a node is blocked because of a send or receive operation, computation on the same node can continue on another thread. Context switching within threads provides low-overhead synchronisation but concurrency control becomes the explicit responsibility of the application programmer [189].

In non-blocking mode (asynchronous), none of the processes is blocked during a send or a receive process. For a non-blocking send, for example, control is returned as soon the call has been submitted to the underlying system and for a non-blocking receive the call returns whether data are available or not. The underlying message layer then takes care of buffering and queing the message until a receive accepts the data [228]. A communication scheme that involves at least one non-blocking call is known as asynchronous. The implementation of non-blocking communications is more difficult because issues such as message queue management and send/receive buffer management consistency arise. In an unbuffered primitive mode a receive call at the receiver end prepares the kernel where the receiving data will be stored. In the case that a send() happens before a receive() call the kernel has not been informed (from the receive() call) that there is a process is waiting for a message to arrive so the incoming message is discarded as the kernel has no indication of which process to send it. In general asynchronous communication can enhance parallelism e.g. latency hiding techniques which can separate and overlap computation and communication parts [212].

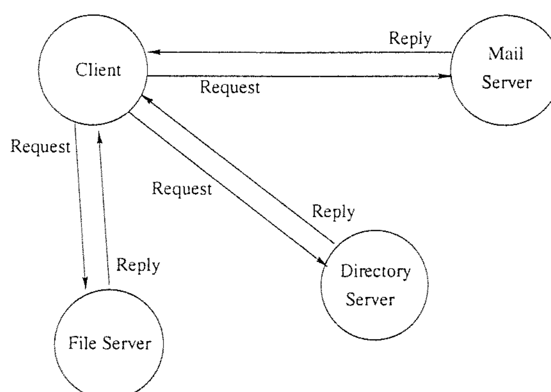

Figure 3.1: Client Server model

In buffered primitive communication mode, a process that intends to receive messages requests the kernel to create a mailbox for it. The incoming messages are stored in the mailbox and the process can use the receive() function to read messages from the mailbox [203, 79].

3.2.2 Client-Server Model

The client-server model provides the basis upon which many distributed application are constructed. The client-server model is not used often in parallel computations, however workstation clusters use it indirectly for run-time system support. Clients and servers are relative terms and refer to software subsystems rather than hardware components [188], hence they could be either in separate machines and communicate over the network or in the same machine using the message passing facilities of the OS (e.g. pipes). In general a server maintains data objects and defines operations on them which typically can be invoked on the server site to manipulate data and exported to clients [11]. A server is any program that offers a well-defined service that can be accessed over a network and a client is any program that sends a request to a server and expects a response.

In this model the client initiates an activity by passing a message/request to the server, then the server processes that request remotely and passes the reply back to the client. Figure 3.1 depicts the fundamental concept of the client-server model. The client-server model is simple and efficient in concept and can be implemented using a single request/reply protocol on a variety of software and hardware platforms [19].

3.2.3 Remote Procedure Call

The Remote Procedure Call (RPC) is the *de facto* industry-standard communication mechanism used for constructing distributed programs and applications. Key features of RPC are request-reply protocol behaviour, UDP/TCP transport, standardised data representation via the eXtended Data Representation (XDR) protocol and authentication support. It defines a well-understood high-level language definition which provides a general-purpose model for interprocess communication (IPC) in a transparent way across a network [19, 237]. The RPC paradigm shields the programmer from the details of the communication network applications

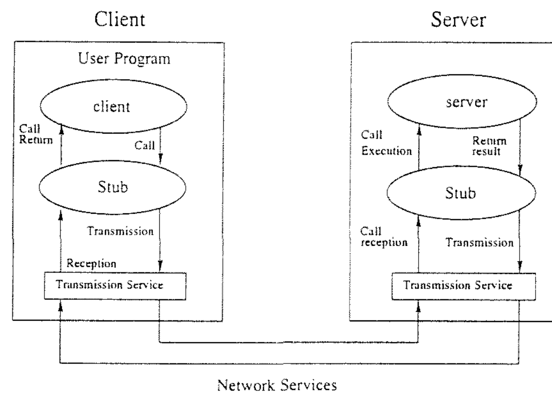

Figure 3.2: The RPC mechanism

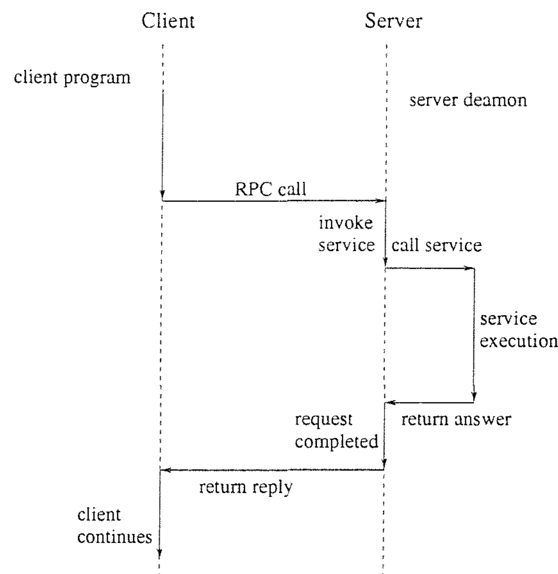

Figure 3.3: Network communication with RPC calls [217]

can thus be portable and more robust, although RPC does not provide explicit error checking and recovery mechanisms.

An RPC facility is built on top of a transport-level service (e.g. UDP). In this way RPC solves the problem of heterogeneity between different peers as well as application standardisation [15]. At the programming language-level an RPC is an ordinary function call that passes all its arguments to the RPC protocol [215]. The RPC mechanism is illustrated in Figure 3.2 which shows the flow of control, when a local (client) machine invokes an RPC. The calling process is suspended and the execution of the call takes place on the remote machine (server) in a different address space after which the server returns the result back to the client and the execution of the program continues on the local machine [16].

The Sun RPC Interface Definition Language enables a programmer to define the functional interface to an RPC program and the interface compiler by using the *rpcgen* tool [217]. An RPC-based application has the following components: the *compile time* which includes programming

language, interface description language client and server stub structure and generation, the *binding* protocol, and the three protocols employed at *call-time* i.e. transport protocol, control protocol and data representation. Parameters can be passed by value but not by reference (although some implementations can pass parameters by reference using a more complicated mechanism). The implementation of an RPC involves the use of several network protocol layers and memory copying accesses hence its performance is often an order of magnitude slower than local procedure calls [11, 49]. The RPC mechanism was been widely and successfully adopted mainly due to its simplicity and generality which provides transparency in both homogeneous and heterogeneous environments.

Other distributed application communication mechanisms include active objects that can migrate autonomously among nodes known as mobile agents. This communication mechanism and can be built in a platform independent language such as Java or a script language such as TCL and enhance interoperability in heterogeneous networks. The mobile agent paradigm is asynchronous and does not block computation on the client site. Hence, the mobile agent model can be efficient and provide fault tolerance [71, 123].

3.2.4 Coordination, Synchronisation, Concurrency control

All these terms are frequently used in distributed and parallel computing. Synchronisation refers to the need for one process to wait until another process has completed an activity [142]. Coordination, i.e. synchronisation and concurrency control among nodes, are crucial aspects for the functionality of both DCS and parallel systems [128, 131]. Parallel computing and distributed systems extend the concept of a sequential coordination further for interprocess coordination and concurrency control of resources located in different nodes [3] because parallel computation cannot proceed without internode coordination.

Synchronisation and concurrency control for single processor systems can use classical UNIX Inter-Process Communication (IPC) techniques such as signals, pipes, semaphores or monitors (e.g. signals as WAIT, SIGNAL, locks, interrupts) or System V IPC mechanisms for shared memory and message queues which are adequate to ensure synchronisation and avoid deadlocks among processes in a single address space [11, 206, 190]. For distributed memory systems (e.g. networks of workstations) where there is no basis for sharing memory areas, internode coordination typically uses a message-passing mechanism. Basic communication primitives such as send and receive are used to synchronise nodes because the transmitting processor knows when the message is sent and the receiving process knows when a message arrives.

Synchronisation methods used in a single CPU system cannot be expected to operate in a DCS environment, therefore new techniques have to be developed, e.g. global synchronisation or barrier synchronisation. Distributed systems have similar deadlock and synchronisation problems as centralised systems in addition to additional potential for deadlocks arising from their communication network. Detection and handling of deadlocks can use the same methods as single computer deadlock handling methods use (locks, timestamps, global time, virtual communication channels) [221, 110]. Methods of mutual exclusion and locks are usually built on top of these communication primitives [49, 221, 130].

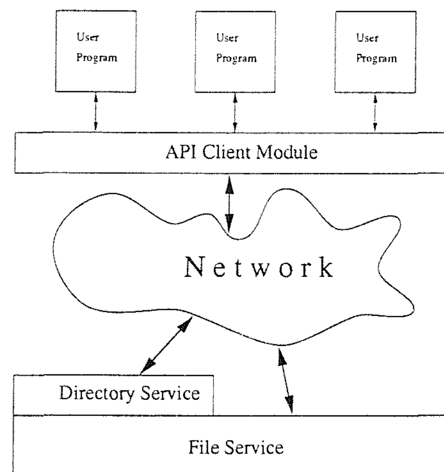

Figure 3.4: Components of a distributed file system

3.2.5 Distributed File System Concepts

Directly analog to a single-processor filing system, a distributed filing system is a basic component in any DCS or parallel system and can be seen as an extension of the classical filing systems. Clients, servers and storage devices are all spread among a distributed system [49, 224]. An important requirement of the filing system is that it should provide flexibility and scalability. A distributed file service (DFS) and a distributed directory service are also required to provide additional properties and features such as naming and pathname resolution concurrency control transparency, etc. [11, 193, 194].

A DFS can be divided into three fundamental services: the file service (concerned with implementing operations on the contents of files), the directory service (which provides operations creating/deleting naming directories or mapping names), and the client module service which integrates computer directory and file services for each client under a single Application Programming Interface (API). A common file system for distributed and parallel systems is important as it can reduce significantly the replication of resources (files) and the administration cost of the system. For example the user nodes in Figure 3.4 can share the same configuration files using the distributed file system.

The Network File System (NFS) is a well known distributed file sharing standard, initially developed by Sun Microsystems [216]. The NFS is based on a client-server model and provides transparent access to remote files, with each computer able to act as both client and server. The importance of a shared file system in a distributed system and workstation clusters is twofold, it minimises the need of a filing resource and at the same time minimises administration costs providing a flexible and shared configuration environment among the system nodes.

3.3 Multicomputers

According to Foster [80] a multicomputer is a Distributed Memory (DM) MPP architecture in which interconnected computing elements have their own memory space. Any node can send a

message to any other node. Each node-computer executes its own program in the classical "von Neumann" model but it can equally access local memory or memory on other nodes remotely via message-passing mechanisms over an interconnection network. Distributed-Memory Multiple Instruction Multiple Data (MIMD) machines fall into this category[1], have workstation clusters can implement the same form of multicomputer parallel machine based on the concept of the message-passing computational model.

Multicomputer MPP systems usually have very distinctive structure characteristics. Their internode message passing network operates in an intra-computer environment at extremely high data rates with a very low error rate. The network topology selected (for example hypercube, 2D mesh) usually avoids cyclic dependencies and deadlocks. Hence simple and aggressive communication primitives can be implemented i.e. cut-through routing, flow control, etc. The architecture of typical MPP multicomputers can scale up to potentially thousands of nodes while still avoiding potential communications bottlenecks and hence increase the computational power of the system in an efficient fashion.

3.4 Clusters as Parallel Computing Platforms

As pointed out in Chapter 1, clusters provide a way to build powerful, cost-effective parallel machines using standard off-the-shelf computers and networking technology [241]. Pfister [178] defines a cluster as a type of parallel or distributed system that consists of a collection of interconnected whole computers utilised as a single unified resource. The main reasons for the current interest in clusters are the improved price/performance ratio of workstations, the development of high-speed off-the-shelf interconnection networks and the establishment of software tools and programming models (e.g. message-passing).

In addition the current trend in computing is in favour of workstation clusters [6, 178, 81] as in future developments are expected to provide not only improved workstations (VLSI, DRAM memory, disk capacity [176, 107]), but also faster networking technologies (such as ATM, Myrinet, Gigabit Ethernet [70]), together with better DCS software tools and standards (TCP/IP, OSI, Distributed Computing Environment DCE, Distributed Management Environment DME, ISO/Reference Model for Open Distributed Processing [20], etc). Another potential advantage of clusters is the availability of their key components which in addition provides scalability and cycle harvesting capability as well (e.g. a distributed system can be easily configured as a cluster of workstations to exploit unused cycles which are "free"). The increasing popularity of NOWs establishes a parallel processing paradigm known as "network based computing" [171]. The availability of clusters can improve as well the response time of applications over a batch system (e.g. an MPP scheduler with a long queue of submitted jobs).

Clusters of workstations in comparison with MPPs have inherent problems and difficulties arising from their distributed nature. Very often, clusters have to operate in an "shared" environment with heterogeneous machines and an OS, tools and facilities that were originally

[1] Shared-Memory MIMD and Rarallel Random Access Machine (PRAM) architectures fall under the multiprocessor category [80].

designed for distributed systems (in comparison MPPs will have their own dedicated and well-tuned OS and tools). This generality and the heterogeneity of clusters software inevitable affects the performance available to application. In order for clusters to be established as a parallel platform with MPP-like performance, issues such as internode communication, programming models, programming environments, resource management and performance evaluation all need to be adequately addressed [171].

For example the communication bottleneck is not a matter of a simple replacement of the communication links. Design of efficient messaging layers and user space communication protocols is required as well as architectural and OS support for multiple communication methods (e.g. collective communication, multicast, broadcast, network topologies asymmetric bandwidth networks, etc.). There is also a need for efficient implementation and support of programming models for the distributed memory paradigm (e.g. PVM and MPI) on clusters. The message-passing paradigm as a concept is straightforward to understand and its implementations are efficient on both clusters and MPPs. In contrast the development of message-passing programs is significantly more difficult compared with the development of shared-memory and sequential-computer programs as it requires explicit programmer handling of communication and synchronisation between nodes, domain decomposition, etc. Sophisticated programming environments and tools (e.g. debuggers, performance monitoring, "graphical languages") with high-level abstraction mechanisms are required to simplify the development of parallel applications. A productive programming environment is necessary to allow programmers to develop their parallel applications easily. Software applications and algorithms tailored for clusters (i.e. latency tolerance algorithms) are still limited and require the support of a robust programming model.

Clusters as distributed systems have the same characteristics as a DCS has in terms of software tools and standards. However, there is lack of a cluster configuration standard, each cluster being built using different configurations. The system should provide run-time support for the resource management and automate the facilities of load balancing, job scheduling and if possible fault tolerance. Prediction and performance evaluation of clusters is necessary as it will assist to assess the usefulness of current systems and provide valuable information to design better systems in the future.

3.4.1 Cluster Hardware Aspects and Structures

Clusters differ from Symmetrical Multi-Processor (SMP) or other parallel computers, in the sense that they are composed of complete computers in contrast to SMPs which replicate parts of a computer, (e.g. processors). Massively Parallel Multicomputers such as the IBM SP series replicate whole modules of workstations such as processors, memories and I/O systems, making distinctions with clusters even more difficult. Despite the similarities between clusters and massively parallel systems there are fundamental conceptual differences.

Parallel systems of SMPs and MPPs may use commodity components (such as CPUs) and utilise a "bottom-up" fine-grain performance orientated approach to build a system. The performance of these parallel systems is highly optimised both from the hardware perspectives

Table 3.1: Parallel Systems, Clusters and Distributed Systems comparison

Characteristic	Parallel Systems	Clusters	Distributed
Number of nodes	large	medium	large
Performance metric	turnaround time	throughput turnaround	response time
Node individualisation:	none	none	required
Communication standard	proprietary nonstandard	proprietary/ standard	strict standards
Inter-node security	nonexistent	varies	required
Node OS	homogeneous	homogeneous/ heterogeneous	homogeneous/ heterogeneous
Runtime System	Proprietary non-standard	general standard	general standard
Runtime support	vendor granted	–	–
H/W Availability	vendor limited	open	open

and software issues. An important characteristic of these systems is that they are sometimes built to address a specific class of applications. Support for these systems usually comes from the vendor side and could cover software/hardware or even application/algorithmic issues of the system. Despite the potential advantages this approach has several disadvantages including high-cost, decreasing number of vendors, variability of architecture types across vendors or even between successive generation of a given vendor and sometimes inadequate software environments.

On the other hand clusters of workstations usually use commodity components and have to deal with their design generality among various abstraction layers which usually leads to a "middle-up" coarse-grain approach to parallelism. Performance targets then become more difficult to meet and often depend on the ability and experience of the cluster designer or administrator. Furthermore, support and development of the system is usually limited to the application developer. This is a major point of difference between workstation clusters and MPPs which is often misunderstood. The "knowledge" and support of an MPP system resides mainly on the vendor side while in workstation clusters that knowledge resides in the cluster manager and application programmers/users [196].

Scalability features are available for both workstation clusters and distributed systems because they are composed of whole computers. Cluster nodes can be added or replace (upgrade) existing ones without disturbing the system at a minimal cost, while changes to MPPs are usually difficult and expensive. Portability of parallel applications has now become an important issue and parallel applications originally developed for other parallel systems might use algorithms which are not performance efficient when ported to cluster of workstations architecture (e.g. latency tolerant algorithms).

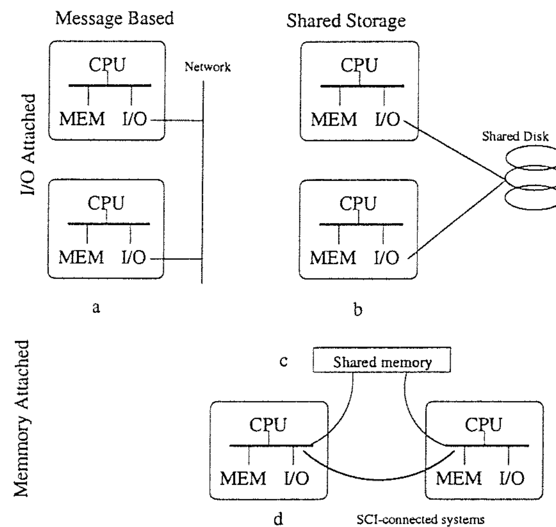

Figure 3.5: Categories of Cluster hardware

Distinctions between NOWs and distributed systems are also difficult. The key point here is the "Single System Image". For example nodes in a distributed system retain their own individual identities despite the transparency of the system. On the other hand, clusters are viewed from outside as anonymous (e.g. the processors of an SMP system). There is no requirement to access node A or B of a cluster, rather the concept is to access and use the cluster as a whole integrated system. Table 3.1 summarises important differences among MPPs, NOWs and distributed systems.

A first classification attempt among clusters is the communication network, whether it is dedicated or not (i.e. exposed to the outside view or enclosed within the cluster itself [178]). Sharing the cluster interconnection network with other public communication facilities is also possible. The cluster can make use of all idling workstations over a campus (scavenging), but this also implies the use of message-based standardised communication, which usually has a high overhead in terms of messages, increased latency and lack of network security.

A dedicated intra-cluster communication system with low overhead and increased level of security can be used as well. The communication medium and the method in which computers are attached to it are two other orthogonal characteristics in which the computational model classifies clusters (see Figure 3.5) [178].

- **I/O Attached** Communication is performed by using I/O operations usually initiated by the OS. Using I/O operations message latency could be relatively high compared to memory operations.

  Message Based could use LAN, FDDI, ATM, or any other network technology. Scalability and portability with message-passing is difficult, although heterogeneity is possible.

  SharedStorage The shared storage system is a shared disk system. All the nodes in the cluster have direct access of the disks on which shared data are placed. The storage

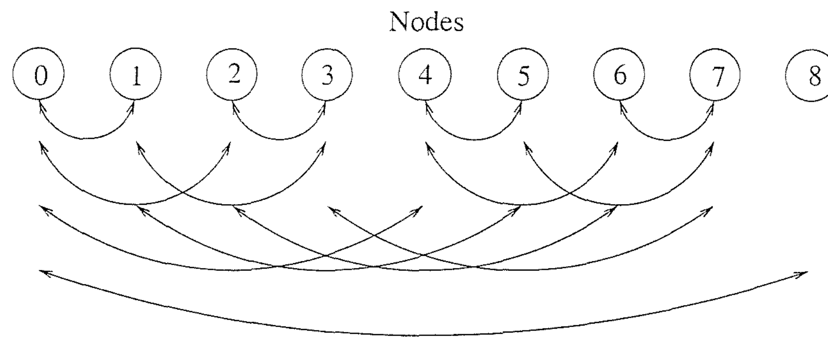

Figure 6.3: The underlying *MPI_SentRecv* calls of a butterfly communication pattern of a 9-node communicator *MPICH MPI_Barrier* routine

Data Movement-I Test

The broadcast and multicast one-to-all operations are built on top of single peer-to-peer primitives. The broadcast latency test measures the time from when the root node initiates the multicast command until the time the last processor receives the message and completes the call. In practice this measurement is not easy and traditional "ping" techniques cannot be used because the root node is not guaranteed to participate in all steps of the call, there is no global clock and it is not possible to identify the last node [167]. The SCOPE broadcast test uses a simplified combination of broadcast and synchronisation calls:

```
synch_barrier();
start := GetTime();
 For I:= 1 To N Do
   broadcast(message);
   synch_barrier();
 End Do;
end := GetTime();
time := (start-end-comp_overhead())/N;
```

This test measures the latency of the cluster while performing a combination of broadcast and barrier synchronisation calls. The MPICH implementation of the broadcast call is based on a recursive subdivision (binary tree) algorithm, such that data flows top-down from the root to the leaves. The root sends to the process *size/2* which becomes the root process for that subtree and so on, which means that in theory such a call is completed within $\lceil \log_2 p \rceil$ steps. Optimisation of the code attempts to switch the binary tree algorithm into a linear broadcast (chain) when the subtree size becomes too small. For long messages a pipelining technique which splits the message into smaller blocks can be used along with non blocking operations.

Figure 6.4 illustrates the order of the individual peer-to-peer calls required for the accomplishment of a broadcast call on an 8 node cluster. The logarithmic nature of the call on a switched technology network can be approximated by the following equation:

$$t_{bcast}(n,p) = t_{oc} + t_{init} \cdot p + (t_s + t_w n) \lceil \log_2 p \rceil \tag{6.16}$$

system is required to provide an ownership mechanism of segments of the storage. Examples of such systems are, DEC Open VMS Cluster, IBM Sysplex[2][166, 124], etc. The computational model is close to the traditional uniprocessor model and load balancing is adequate.

- **Memory Attached** Communication is performed by processor-native memory attached *load* and *store* operations. Communication performance is better than the I/O attachment but their implementation in both hardware and software is difficult.

  MessageBased There are no current examples in this category.

  SharedStorage The shared block of storage is the memory, it could either be separate from the individual storage provided for nodes, or it could be contained in the individual nodes. The SCI bus based clusters is an example of this category, e.g. the IBM POWER/4 [14] system.

3.4.2 Communication Requirements

The vast majority of clusters use a bus technology such as Ethernet for the internode connection. In these cases throughput and especially latency of the network is two or more orders of magnitude slower than the internal data busses of the nodes. This difference in magnitude can cause a serious bottleneck, especially for applications with intensive I/O among the nodes. The usual way to overcome this potential bottleneck is either to separate the computational (t_{calc}) and communication (t_{com}) part of a calculation t_{th}, by rescheduling the application to overlap the communication part with the computation part (or alternatively to reschedule the application in some way to minimise the internode communication requirements) [212, 213].

$$t_{th} = t_{calc} + t_{com} \tag{3.1}$$

t_{com} can be analysed further according to equation 2.2 to:

$$t_{com} = t_s + n \cdot t_m \tag{3.2}$$

Overlapping commutation and communication parts of equation 3.1 the runtime t_{th} becomes:

$$t_{th} = max(t_{calc}, t_{com}) \tag{3.3}$$

An ideal internode communication bandwidth N_{net} for a cluster with n nodes should be equal to the bandwidth of all the I/O sources $N_i(I/O)$ that the fastest node has, although that bandwidth is beyond effective utilisation.

$$N_{net} = \sum_{i=1}^{n} N_i(I/O) \tag{3.4}$$

[2]IBM S/390 Sysplex cluster architectures currently support scalable commercial applications such as on-line transaction processing (OLTP) and parallel database systems.

Table 3.2: ASCI machines summary

Features	Intel ASCI Red	IBM Blue Pacific	SGI Blue Mountain
Processor type	200MHz PentiumPro	500MHz Power3	SN1
Performance/node	200 Mflop/s	800 Mflop/s	1 Gflop/s
Peak performance	1.8 Tflop/s	3.2 Tflop/s	>3 Tflop/s
Number of nodes	9216	4096 (512x8)	3072
System architecture	DM message -passing	Cluster of SMPs with DM	Cluster of SMP with DSM
Memory	594 Gbyte	2.5 Tbyte	500 Gbyte
Link Bandwidth	800 Mbyte/s	800 Mbyte/s	1560 Mbyte/s

3.5 The ASCI Project

The Accelerated Strategic Computing Initiative known as ASCI was launched in 1996 by the Department of Energy to build Teraflops supercomputer systems. It is an ambitious plan which calls for a threefold increase in computing performance every 18 months over a 10-year period [42]. Currently there are three ASCI machines installed in three labs: the ASCI Red in Sandia National Laboratory by Intel, with 1.8 Tflops peak capability, the ASCI Blue Pacific at Lawrence Livermore is an IBM PowerPC system of 512 nodes eight CPU per node with 3 Tflops and the ASCI Blue Mountain at Los Alamos by SGI/Cray which is a 3000 node.

Very often the ASCI project machines are referred as MPPs systems but in practice they have several similarities with clusters as well. All machines use off-the-shelf high-end CPUs to scale up to Tflops level. Technological challenges that an ASCI machines needs to address is the high-performance interconnection which has to be scalable to a very large number of nodes, software issues e.g. distributed OS parallel programming and high-performance I/O aspects.

3.6 The Beowulf Class Cluster Computers

The Beowulf project is an example of high performance cluster which has emerged as a viable path to scalable computing systems for scientific and engineering applications. The Beowulf cluster was introduced at NASA Goddard Space Flight Center's Center of Excellence in Space Data and Information Sciences (CESDIS) for the need of the Earth and Space Sciences project (ESS) in 1994 [208]. The first Beowulf cluster was build around 16 Intel 80486 DX4 processor systems connected by channel-bonded Ethernet. The Beowulf architecture has no custom components and is a fully COTS (Commodity Off The Shelf) configured system. The concept of Beowulf clusters was further promoted by the rapid evolution of mass market commodity technology i.e. microprocessor price/performance improvement, network technology advances such as private system area networking, availability of freely software such as the Linux OS as well as GNU software and the "standardization" of message passing libraries such as PVM

and MPI. The availability of the software source code is important because it enables custom modification to facilitate parallel computation [54]. As a result, Beowulf class cluster computers range from several node clusters to several hundred node clusters.

Communication between processors on Beowulf is achieved through standard Unix network protocols over Fast Ethernet networks internal to Beowulf, which proved to be both straightforward and cost effective. Beowulf-class cluster computers provide scalability ranging from several-node clusters to several-hundred-node clusters, e.g. a two-level fat-tree-like topology comprising 240 nodes of 16-way Fast Ethernet switches [192]. Communication throughput is limited by the performance characteristics of the Ethernet and the system software managing message-passing. Beowulf is capable of increasing communication bandwidth by routing packets over multiple Ethernet segments ("horizontal", "vertical"), each node then acting as a software router in order to allow not-adjacent nodes to communicate. Hawick et. al. in their paper [105] discussing some of the key issues for designing a Beowulf system at the beginning of 1999 conclude that PC compute nodes with Intel processors running at 350-400MHz clock speed interconnected with Fast Ethernet gives the best price/performance ratio.

In order to establish and support Beowulf clusters as a viable alternative parallel platform, the Beowulf project incorporated research into applications and algorithms suitable for clusters (i.e. latency tolerant algorithms). In addition the cluster configuration has to reflect and match specific application requirements as well as to adapt rapid technological advances. Other areas that the Beowulf project has focused on are: system software support and tools (debuggers, tuning tools, scheduler, etc), load balancing, low-level programming interfaces, scalability, heterogeneous computing. A commercial distribution of Beowulf software has been released in 1998 by RedHat in cooperation with NASA CESDIS known as "Extreme-Linux" [153].

Today many Beowulf clusters demonstrate remarkable performance ratings running scientific applications, many Gordon Bell Performance Prizes include Beowulf workstation clusters, e.g. Goddard Space Flight Center 10.1 sustained GigaFlops with a PPM (Piecewise Parabolic Method) code on 199 CPUs (11/17/97), Caltech/JPL collaboration 10.9 GigaFlops with an n-body problem on over 120 CPUs (11/18/97) [207].

3.7 The NOW project in Berkeley

The Berkeley Network of Workstations (NOW) project is another example of workstation clusters that demonstrates that it is viable to build inexpensive large scalable parallel computing systems [51]. The cluster consists of 105 SUN Ultra 170 workstations with two interconnection networks. A Myrinet network provides the high-speed communication within the cluster while a switched-Ethernet network into an ATM backbone provides scalable external access to the cluster. The topology of the Myrinet network is a variant of a fat-tree to create a system with uniform bandwidth between nodes using thirteen 8-port 160Mbyte/s bidirectional network switches.

The software architecture of the NOW project is based on a complete version Solaris OS, run on each node, with extension to interfaces that support global operations over the cluster such as the GLUnix process management layer, memory management and file system support

Table 3.3: Important software development areas for Beowulf class clusters

Software/Tool Area	Existing or Under Development
Application Development	Utilities *sh*, *csh*, *grep*, *sed*, *diff*, *make*, etc.
	Languages f77, C, C++, linker, HPF, f90, etc.
Debugging/Tuning Tools	Debugger
	Profiling tools, Tracing Tools
	Hardware Performance & Monitoring Tools
Low-level Programming Interfaces	Message Passing MPI, PVM
	POSIX Threads
	Math Libraries FFT, BLAS, etc
	Parallel I/O MPI-2 I/O
OS Services	Networking TCP/IP
	Filing System >4Gbyte support
	Job Queuing and Scheduling e.g. EASY
	Accounting, Quotas and Limits Enforcement
Ensemble Management Tools	Common Boot/Install Configuration Package
	System Monitoring
	Parallel *rsh*, *ps*, *kill*, etc.
Documentation	How to build a Beowulf

(xFS). The basic communication primitives in this project are based on Active Messages which provides application-direct, protected user-level access to the network.

The higher-level communication library of the NOW project is MPI which is based on a modified MPICH implementation built with a customised abstraction device interface (ADI) for active messages. The NOW project has successfully shown that a large cluster of workstation can obtain low-latency and high bandwidth communication over a range of parallel applications. Current research challenges of the project is the integration of communication and scheduling within a time-shared resource [51].

3.8 The Clusters of the University Campus

Building a cluster of workstation nowadays is relatively straightforward. Several high-performance workstations with a fast interconnection network will suffice to implement a minimum system. A large number of clusters have been assembled over the past few years in research institutes and universities, but in general each of those clusters is different, creating difficulties for standardisation and the deployment of software products or tools (e.g. administrative and monitoring tools, job submission, load balancing, etc.). One of the most well-published clusters of workstations is the Farm at T.J. Watson Research Center which consists of 22 high-performance workstations [178]. The cluster has one *master (terminal) server* which is a gateway to the outside world, 19 *compute servers*, and two *file servers*. A terminal server on the master workstation selects

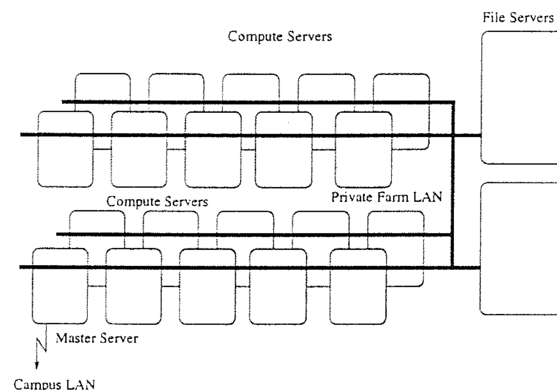

Figure 3.6: The Farm

the least loaded of the workstations to run a job.

The Fermi National Accelerator Laboratory runs another of the largest workstation cluster with 400 Silicon Graphics and IBM workstations. This system is primarily used to execute parallel applications which are individual jobs that simultaneously use several of the clustered machines. This type of overt parallelism is known as *serial program, parallel sub-system* (SPPS).

This case study examines and demonstrates some important characteristics of the clusters of workstations used in this university campus. There are two orthogonal axes that university clusters of workstations are divided into, according their homogeneity, and according the way they are attached on the network (single segment topology or switching). The low speed of the Ethernet network restricts the available bandwidth to 1 Mbyte/sec and the latency is high due to the layered structure of the standard network protocols. At the same time other users can use the nodes of the cluster as workstations, affecting further the performance of the system. Under these circumstances the campus clusters can be used only for coarse grain parallelism computation during off-peak (non-working) hours. Despite these disadvantages clusters can be used on an experimental basis and to provide a test-bed to study the basic characteristics of clusters. A promising configuration of clusters is that of Figure 3.7 (c) in which a second fast interconnection network is used among the nodes. Applications use the fast link for message passing and coordination while the interconnection and access to the outside world goes through a standard interconnection network network. In this way nodes can use a proprietary network protocol which can improve the communication bottleneck by improving significantly latency and bandwidth [38]. Efficient implementation of such protocols requires additional changes to the run time system as well.

A common characteristic of these clusters is the shared file system which is based on NFS. The file server can be either directly attached to those clusters or indirectly via the campus backbone network. Applications software for homogeneous networks is common for all workstations. In contrast to heterogeneous networks, separate application executables and configuration files must be preserved for each different computer architecture. Application development and installation then becomes more difficult because portability and inter-operability must be preserved. In a heterogeneous network there are no restrictions about the node architecture and each time the best performing workstations for an application can be chosen. Performance

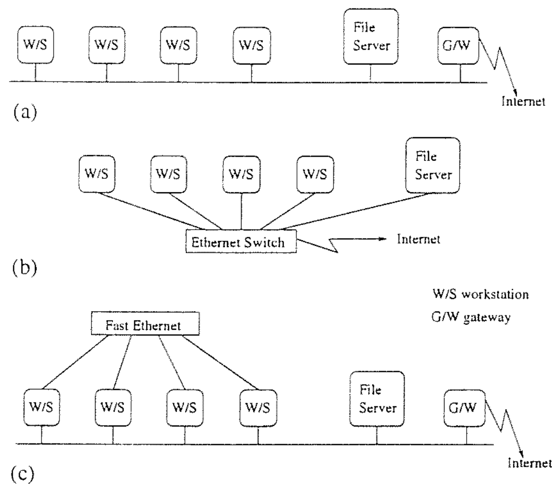

Figure 3.7: Classification of clusters of workstations in the campus

Table 3.4: Different cluster configurations

Cluster Type	Network Topology	Workstations/OS
Heterogeneous	Ethernet Switch	SPARC-x/SunOS
		IBM/AIX
		PC/LINUX
Heterogeneous	Single segment	SPARC/Solaris
		SGI/IRIX
Homogeneous	Single segment	SPARC-4/Solaris
Homogeneous	Fast Ethernet	SGI/IRIX
Homogeneous	Fast Ethernet	LINUX

measurements or estimation analysis is more difficult due to the node heterogeneity.

Single-segment Ethernet topology networks suffer from poor performance because of their single bus congestion problems, especially for applications with high internode traffic [150]. Switched Ethernet clusters can reduce significantly congestion problems with a careful scheduling of the application internode traffic. Table 3.4 shows configuration information of the available clusters at the University of Southampton in 1998.

3.9 Summary

This chapter has examined and reviewed the key characteristics of workstation cluster from the DS and the MPPs prospective. Such system provide a flexible parallel platform which can incorporate characteristics from both DS and MPPs at the same time.

In its simplest form a workstation cluster is a DS. The majority of workstation clusters are built around an I/O attached message-passing based communication mechanism which can

combine the advantages and simplicity of DS with the message-passing computational model of MPPs. Software tools designed for distributed systems are now mature enough to provide useful services for clusters but are not suitable to deliver MPP-like performance because these tools are primarily designed for functionality and reliability.

Software requirements for performance on clusters are more strict than distributed system requirements. Software support also should not put any limits on the performance and should exploit hardware advanced features of new workstations and high-speed networks. The examples of the Beowulf and Berkeley NOW projects have shown that properly configured clusters of workstation can deliver acceptable HPC services.

Chapter 4

Message-Passing and MPI

4.1 Introduction

This chapter investigates the role of the message-passing model as the inherent computational model of clusters. This is important because it will provide a better understanding and a basis for the analysis of cluster performance as it will be examined in later chapters [201]. One of the fundamental components of workstation clusters as depicted in Fig. 1.2 in Chapter 1 is the establishment of the message-passing computational model which was originally developed for parallel systems.

The mechanism used by nodes to exchange data in the message-passing model is based on send and receive primitives. Each process controls access to its own space in memory and the only way to move data from one process space to another is via messages [203, 79, 168, 140]. Computation is performed through one or more processes, on the same or different nodes, which communicate and coordinate via messages. Sharing resources and synchronisation among processes is achieved by sending messages. Message passing can be asynchronous (e.g. usually non-blocking) or synchronous (blocking).

The concept of message-passing is straightforward and in principle it is an expansion of the existing sequential computational model which run on the same sequential hardware platforms (e.g. the von Neumann model). Implementations of the message-passing model on parallel machines and clusters can be built with the provision of message-passing library calls to existing sequential languages [126] with good scalability and efficiency.

The Message Passing Interface (MPI) standard was designed to replace a large variety of existing message-passing systems which extend on a wide variety of parallel platforms. Implementations of MPI such as MPICH [95], CHIMP [2], LAM [29] run efficiently over both MPPs and clusters. As a consequence the portability of MPI has reduced the differences between the various parallel platforms available and established MPI as the de-facto standard for message-passing.

The rest of this chapter is divided into four sections. The first section reviews briefly PVM and BSP then examines the concepts and semantics of the MPI message-passing computational model. The next section reviews the implementation of these concepts on MPICH. A review of

the architectural structure of the implementation reveals the trade-off between portability and efficiency, which is essential to understand and analyse the performance behaviour of clusters. Finally there is a case study of the MPI (MPICH) computational environment for clusters of workstations. An implementation of a matrix multiplication algorithm run on both homogeneous and heterogeneous clusters of workstations is used as a test-bed demonstration for further study of the clusters parallel platform.

4.2 The Parallel Virtual Machine

The "Parallel Virtual Machine" (PVM) interface, developed at Oak Ridge National Laboratory, was the first message-passing environment to establish a standard model for parallel programming on distributed computing systems [82]. PVM was originally developed under the assumption of a heterogeneous distributed system in which the underlying network could be slow and unreliable e.g. BSD sockets [34]. The PVM system provides an advanced API capable of managing processes dynamically and a run time system to provide application management. A PVM system is an integrated set of software tools and libraries that enables a collection of heterogeneous computer systems to be used cooperatively as a parallel virtual machine for concurrent or parallel computation. The basic principles of a PVM system include:

- User-configured host pool
- Translucent access to hardware
- Process-based computation with an explicit message-passing model (e.g. data and functional parallelism)
- Heterogeneity and multiprocessor support.

The PVM system comprises the pvmvirtuald daemon that resides on all the computers/nodes making up the virtual machine together with the library of interface routines which provides the message-passing functionality together with a facility to spawn processes and coordinate tasks. Currently the PVM system has been ported to a wide range or hardware and software platforms including MPP systems as well as support and bindings for programming languages such as C, C++ and Fortran.

4.3 Bulk Synchronous Parallelism

Bulk Synchronous Parallelism (BSP) is an architecture and platform-independent structured programming model for general-purpose parallelism. The BSP programming model was introduced by Valiant [225] in 1990. In the BSP model each node of a parallel machine has its own local memory space and interconnection network that can route packets of some fixed size between nodes. The model reduces an overall computation into a series of supersteps, each containing computation and/or communication followed by a global synchronisation among all the processors of the parallel system [88]. The concurrency model assumes that no process can

proceed to the next superstep unless all processes have completed the current superstep. Each superstep is subdivided into three ordered steps:

- local computation step on each node/process, which uses only variables stored locally.
- communication step, in which processes exchange data among themselves.
- a barrier synchronisation step, in which processes are blocked until the communication phase is entirely completed. The exchanged data become locally available at this point and computation proceeds to the next superstep.

Values sent through the communication network are not guaranteed to arrive until the end of the current superstep. The overall concept of the BSP model is straightforward and it is considered as a direct enhancement of the sequential programming model. BSP programs can be written in one of two ways:

1. Direct BSP programming, in which the programmer takes full responsibility for the two-level memory model to ensure that data to be operated upon reside in local memory at the beginning of every superstep.
2. An automatic style of BSP programming, in which a lower-level entity maintains the illusion of a single memory.

BSP can be efficiently applied to a variety of parallel platforms and the performance of a program can be predicted from the text of the program together with a few global parameters of the target architecture. These parameters are the number of processors p, the ratio of communication throughput to processor throughput g, the computing speed in flop/s s and the time required to barrier synchronise all processors l. The cost of a single superstep then is the sum of the maximum cost of the local computation on each processor plus the cost of the global communication on an h-relation and the cost of the barrier synchronisation at the end of the superstep:

$$cost\,of\,a\,superstep\,S_i = MAX(w_i) + MAX(h_i g) + l$$

where w denotes the amount of local computation load and h is the number of data elements sent or received by a processor. The total execution time of an algorithm is obtained by adding the times of each separate superstep S_i [225]. Evaluation of the parameter g is not straightforward, at BSP network traffic occurs in bursts at the end of each superstep, that, it can be approximated by the concept of network "permeability" when all processes send and receive messages simultaneously in a random order rather a linear approach. BSP provides an effective and usable model, unlike unstructured parallel programs implemented on message-passing and shared memory models.

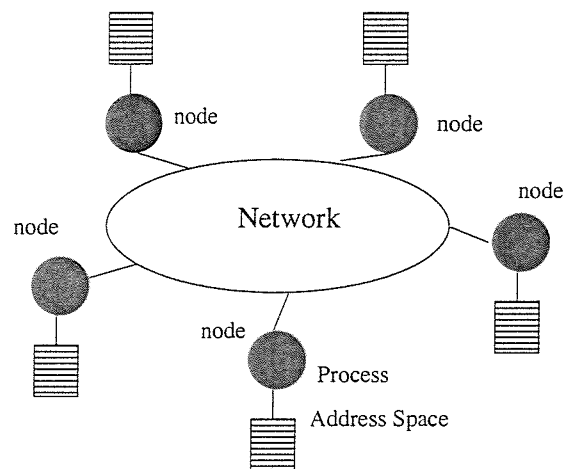

Figure 4.1: The message passing model

4.4 The Message Passing Interface, Concepts and Semantics

At the start of the 1990's many software tools and packages for distributed memory systems were available. The need for a standard arose because most of those packages had shortcomings either in portability, performance or were incomplete. The basic features of the proposed standard were discussed in an workshop held in April 1992 in Virginia and a preliminary draft proposal developed [64]. In November 1992 a draft proposal (MPI-1) was published and the MPI Forum was established, the final version of MPI being released in May 94. In a meeting in December 1995 MPI 2 extensions ware discussed, a year later (November 96) a draft of MPI 2 was released, and the final version of the draft was released in June 1997 [33].

The Message Passing paradigm is well-understood and can be efficiently applied to parallel programming. Before the advent of a standard, a variety of message passing implementations and libraries prevented application portability. Code developed for a specific machine using specific message passing libraries could not run on another platform [108], while portable message passing libraries were often not efficient or complete.

In order to provide language independence MPI runs on top of an underlying communication protocol, (for example TCP/IP), providing portability for both heterogeneous and homogeneous environments. The standard does not specify explicit memory operations, program construction tools, debugging facilities, or any other specific functions that could affect portability. The Message Passing Interface is thus a specification for message passing libraries designed to be a standard for distributed-memory, message-passing, parallel computing systems. MPICH [95], CHIMP [2], LAM [29], Unify [35] are implementations of the MPI standard for a variety of systems with binding libraries for programming languages such as C and Fortran or even Java [156, 87] together with other tools and facilities.

For the MPI programming model a computation comprises one or more processes that communicate, by calling library routines, to send and receive messages among the processes.

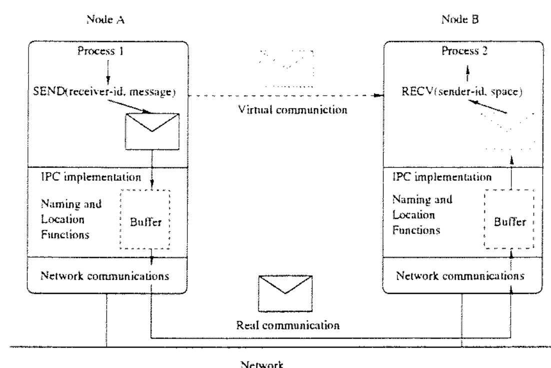

Figure 4.2: Distributed IPC functions are similar to IPC functions on a single computer, although networking, naming and location functions have to be included as well [11]

A fixed set of processes is created at program initialisation, then each process being capable of executing different programs. Hence SPMD-style programs can be directly ported using MPI and in addition limited porting of MPMD style programs is possible [80].

4.4.1 MPI Procedures and Semantics

The notion of MPI is based on the concepts of process, group and communicator. An MPI program can have one or more processes running on one or more nodes. Each process is autonomous and has its own memory space. The MPI standard does not specify the process execution model, hence processes can be sequential or multi-threaded. A group is an ordered set of processes with their own unique identifiers (handles). Each process in a group is identified by its "rank" integer. Groups are always associated with communicators. In order for MPI to ensure safe communication between members of the same group (e.g. to avoid misinterpretation, messages being over-written), the concept of the context, in which a message is passed, is introduced. A communicator is a mechanism which combines together the concepts of group and context. Hence a communicator identifies both the process group and context in which the operation is to be performed. MPI also provides another mechanism to distinguish messages used for different purposes known as the tag, which is an integer assigned by the programmer to identify a message uniquely. The concept of communicator mechanisms in MPI can provide sufficient information hiding that is needed to support modular programming (e.g. parallel decomposition) as well as application-oriented topologies (virtual topologies).

The message passing model is by default non-deterministic i.e. the arrival order of two messages sent from two different processes to a third one is not defined [80]. However the *source* and *tag* specifiers in MPI calls guarantee that two messages sent from the same process to another process will arrive in order. MPI messages consist of two basic parts: the actual data to be sent/received (its format is: *buf*, *count*, *datatype*), and an envelope of information (its format is: *source/destination*, *tag*, *communicator*) that helps to route the data [79, 168]. Heterogeneity is supported through a user-defined datatype mechanism.

Communication between processes within a group can be either point-to-point or collective. The point-to-point call is the basic communication mechanism between a pair of processes with

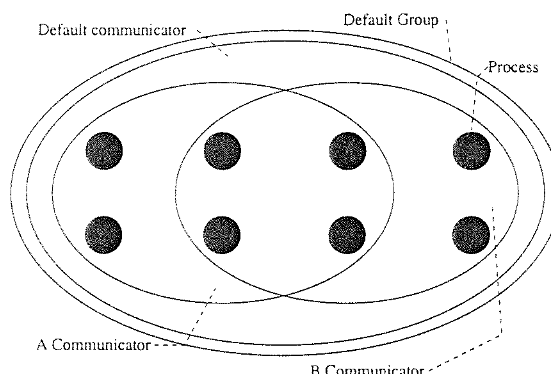

Figure 4.3: A communicator identifies the process group and context in which the operation is to be performed

```
                      data                      envelope
MPI_Send( *buff, count, datatype, | dest, tag, comm );

MPI_Recv( *buff, count, datatype, | dest, tag, comm , *status);
```

Figure 4.4: MPI calls format: the data part plus the envelope part

the type of the call being either blocking or non-blocking. Most of the MPI constructs are built around this mechanism. Group or collective calls are blocking calls and require the participation of all the processes of the group (usually implemented using single peer-to-peer calls). Finally communication modes between the sender and the receiver ends can be either synchronous (e.g. "hand-shake" mode) or asynchronous (e.g. through a probe mechanism) [77].

The first release of the MPI standard (known as MPI-1) kept the standard as simple as possible and avoided adding features such as dynamic resource management, programming tools, debugging facilities, I/O functions, explicit thread support, or other communication operations (e.g. shared-memory like operations, one-sided) [96]. Most of these features are addressed with the second release of the standard MPI-2 (July 1997) [78].

4.5 MPICH

MPICH is a current implementation of the MPI standard, which is both a research tool and a software development project [95]. MPICH was developed at the same time as the MPI standard was proposed, with the objective of providing a test-bed implementation. This enabled problems with the specification to be discovered quicker and was beneficial for the standard, which rapidly became more robust and effective.

In the early stages of MPICH development was based on precursor systems such as p4 [30], Chameleon [93] and Zipcode [202]. The actual MPICH implementations are in the public domain.

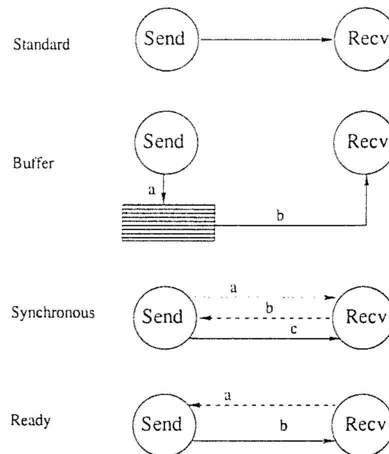

Figure 4.5: MPI communication types

4.5.1 The Architecture of MPICH

The designers of MPICH tried from the beginning to separate the interface between the MPI library and the underlying message passing hardware. In addition, the complicated aspects of MPI such as communicator management, derived datatypes, or topologies are separated from the underlying communications mechanism [96] to hide any peculiarities of a particular system. The layer of code that interfaces with the communication device, (i.e. the lower layer) is inevitably individual for each specific hardware platform and provides hardware-dependent access to communication and synchronisation primitives [154]. This lower layer can be either a native communication subsystem (e.g. for parallel systems), or another message-passing subsystem such as p4, Chameleon, etc. This scheme provides efficiency, universality and flexibility for future upgrades [94, 99]. New device implementations from third party vendors and others can be integrated easily as well as facilitating experimentation with new devices [89, 103].

The Abstract Device Interface (ADI) The central mechanism behind MPICH is a specification called the Abstract Device Interface (ADI) [95, 92, 90, 91]. ADI is a set of functions or macro definitions in terms of which MPI standard functions may be expressed, to preserve portability. Each hardware platform needs its own ADI to exploit all vendor specific features that are provided.

ADI provides four sets of functions:

- Specifying a message direction (send/receive)
- Moving data between the API and the message-passing hardware
- Managing lists of pending messages
- Providing information about the execution environment

The Channel Interface (CI) The very low level of the ADI implementation is a thin layer known as the channel interface [90], with the functionality of this layer limited to data

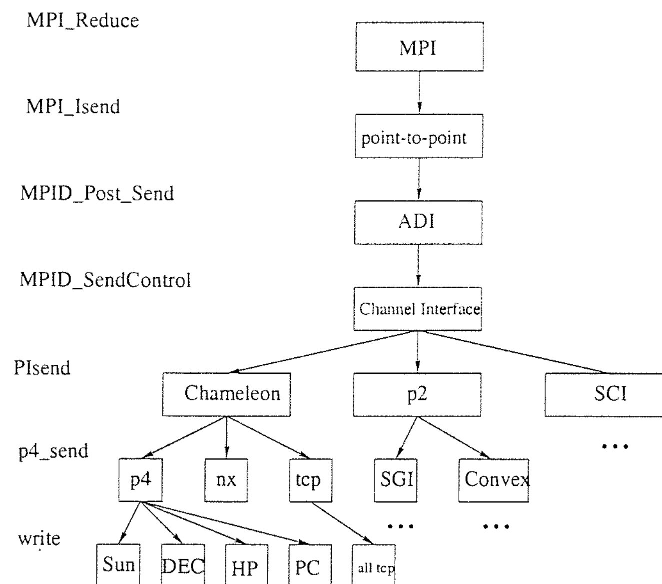

Figure 4.6: Upper and lower layers of MPICH

transfer only. Its basic implementation requires 3 functions to control envelope information *(MPID_ SendControl, MPID_ RecvAnyControl, MPID_ ControlMsgAvail)*, and 2 other functions to send and receive data to/from the channel (*MPID_ SendChannel, MPID_ RecvFromChannel*).

The data exchange mechanisms the channel interface provides are:

Eager Data is sent to the destination immediately (default choice)

Rendezvous The data are sent when a receive is posted that matches the message

Get Data is read directly by the receiver (similar to *memcopy*)

4.6 Case Study: Matrix Multiplication

A case study of a matrix multiplication algorithm implementation using MPI in presented in Appendix C. The purpose of this case study was to gain experience with MPI semantics and developing a simple application using the MPI programming environment and run it over various clusters available on the university campus.

4.7 MPI-2 and Parallel I/O

Implementations of the MPI standard shortly after its release were used very successfully on a variety of platforms by both academic and industrial users. The success of MPI was considerable and it became established as a de facto "standard". Together with the success of MPI-1 there was also a demand for extra features that MPI-1 did not support such as dynamic process management, parallel I/O operations and bindings for other languages. These requests eventually resulted in the MPI-2 standard. The MPI Forum reconvened during 1995 to address new

functionality and to develop an MPI-2 standard. MPI-2 is a superset of the MPI standard which was released in June 1997. The new standard includes C++ and Fortran-90 bindings, external interfaces, extended collective operations (e.g. non-blocking collective calls), and language inter-operability (distinct platforms integration). Extensions to the message-passing model include dynamic process management, one-sided operations, parallel I/O, real-time extensions, external interfaces, etc. [78].

This section examines and discuss briefly some of the new features MPI-2 introduced and which have the potential to affect the computational model of NOWs. For example the dynamic process management feature modifies the computational model towards to a multi-computer model in which each node can start and terminate processes, as well as sending and receiving messages [80]. Extensions of the dynamic process potential will enable two separate applications to interact in a client/server-like style. Inter-operability among different platforms is another important issue MPI-2 addresses. Other features of the new standard are one-sided communication which introduces a shared-memory like operation, parallel I/O features (providing potential for many applications especially for Grand Challenge problems) and the real-time support provision (e.g. time-driven, event-driven, priority-driven) which is expected to enable and merge a new category of real-time applications into parallel computing.

Implementations which fully-support the MPI-2 standard so far have not been released, although preliminary implementations that partially-support MPI-2 are available. Hence real estimations and evaluations of MPI-2 is not yet possible. To achieve these requirements MPI-2 has to remain an interface communication library without managing the runtime environment, it should interact transparently with the OS while not assuming any responsibilities from the OS. Applications already developed on MPI-1 now are facing the challenge of MPI-2 which should balance the trade-off between functionality, portability and performance. Hence application and system developers are interested in estimates or predictions of the impact of MPI-2 on their products. This section introduces briefly the most important features of the new standard.

4.7.1 Dynamic Processes

This is probably the most important feature introduced in the new standard. The MPI-2 process model allows the creation and cooperative termination of processes after an MPI application has started. An MPI application may now start new processes through an interface to an external process manager e.g. CMOST, POE, p4. The MPI_SPAWN call starts MPI processes and establishes communication with them through a communicator. Applications can require a variable number of processes and can use as many processes as required but if necessary then some processes can be returned. An important class of MPMD applications, requiring process control, are supported directly from the standard. In particular for NOWs this is a very important feature as it will enable clusters to use more efficiently overall system resources i.e. workstation idle cycles.

In addition to the direct MPMD programming style support, MPI-2 enables communication to be established between two independently started applications. MPI will create an inter-communicator in which the local and remote groups are the original sets of processes. This

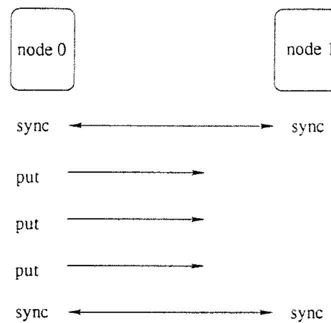

Figure 4.7: One sided communication access (put)

is a collective asymmetric process because processes of one group indicate a willingness to accept connections (such as servers do) and the other group of processes connects to the server (client), so *application*-1 could be a kind of "server" while *application*-2 could be a "client". MPI-2 provides the functionality to support this "client/server" mechanism (MPI calls for accept, listen, connect, etc) therefore client/server applications can use an interface for two parts of a cooperating application.

4.7.2 Single-Sided Communications

Sometimes processes, in applications with irregular dynamic distributed data patterns, have all the knowledge for a call on one side from/to a "window" which another process has made available for one sided access. These processes on their own should be able to initiate communication with other processes, which does not require execution of matching calls at both ends. MPI-2 provides an extended communication mechanism, Remote Memory Access (RMA), by allowing the process to specify all communication parameters, both for the sending side and for the receiving side.

Each process can compute what data it needs to access or update at other processes. However processes may not know which data in their own memory needs to be accessed or updated by remote processes or perhaps they do not know the identity of these processes. RMA operations are initialised by specifying for each process a memory window that is made accessible to accesses by remote processes.

Message-passing communication achieves two effects: communication of data from sender to receiver; and synchronization of sender with receiver. The RMA design separates these two functions and provides three communication calls: remote write (put), remote read (get), and remote update using a window managing mechanism to achieve one-sided communication. The RMA mechanism can take advantage of fast communication mechanisms providing by various hardware platforms such as shared memory, DMA engines, put/get operations, MPPs, etc.

4.7.3 MPI-2 and Parallel I/O

MPI I/O is not the first specification of parallel I/O system, rather the first portable and broadly-accepted parallel I/O specification that has been proposed [78, 74]. MPI specifies

how the data should be laid out in a virtual file structure (the view) but does not specify the physical layout within a file. Specification of the physical file structure is avoided because it will be system specific and hence it will restrict portability. In order to optimise I/O performance and file layout MPI can pass specific control information via *info* objects. MPI I/O is based on the UNIX portable file system model interface with extensions for:

- group, collective data access
- disk directed I/O
- asynchronous I/O, accesses with a stride, etc

Additional features include:

- Basic file inter-operability between systems
- user directed optimisation via portable *hint*
- non-blocking data access

I/O is layerable on top of the MPI-2 external interface. Appendix D gives more details about the MPI-I/O concepts and semantics.

4.8 Summary

This chapter has examined the fundamental issues of the workstation cluster message-passing computational model. This review aims to provide a better understanding of the workstation clusters behaviour at the higher application levels. The MPI is the prominent message-passing library used in clusters and MPPs. In this model computation is performed by one or more autonomous processes which communicate and coordinate among themselves via message-passing mechanisms.

The MPI standard proposed in November 1992 unifies a large variety of existing message-passing systems on many different platforms. MPI supports peer-to-peer communication modes as well as collective communication. The original MPI standard can be used for both SPMD and MPMD (modular programming) programming styles, while the second release (MPI-2) includes enhanced functionality features such as dynamic process management, I/O operations, etc.

An experimental case-study of a matrix multiplication application was used in a range of experiments over a variety of homogeneous and heterogeneous clusters of workstations using MPI. The results demonstrated a high degree of portability and efficiency of the MPI standard, although the communication network on clusters was found sometimes to cause potential bottlenecks. The absence of a standard programming environment and programming tools as well as runtime support facilities was identified as a noticeable disadvantage.

Chapter 5

Benchmarks

5.1 Introduction

This part of the thesis will examine the performance evaluation issues of workstation clusters to be assessed by the proposed SCOPE evaluation tool. In order to understand and evaluate the performance behaviour of workstation clusters, it is necessary to review existing benchmark suites, examine their basic characteristics and demonstrate the requirement for a benchmark suite specifically tailored for clusters. The next two chapters will provide a brief introduction into HPC benchmark suites and examine the proposed workstation clusters performance evaluation benchmark suite.

5.2 The Requirement for Benchmarks

The measurement and understanding of computer system performance has been important since the first computers were built. An accurate measurement of computer system performance will enable people to assess computer systems and provide valuable information not only for system designers but also system managers, vendors and purchasers.

The term *computer performance* as it stands is ambiguous because different people can give different interpretations of its meaning. System designers want to assess performance of a system in order to test and understand its behaviour and possibly improve it or design better systems in future. In a similar way, salespeople are keen to know *how good* is the performance of their system in order to promote sales, e.g. the price/performance ratio, (e.g. Million Instructions Per Second MIPS, Clock cycles Per Instruction CPI). System managers need to have some kind of estimated *performance* features (e.g. throughput) of computer systems that they plan to purchase or use. Furthermore cluster managers need performance evaluation tools in order to evaluate and match the best cluster modular components off the shelf. Finally computer users need to know how fast their application might run on that system (e.g. response time), or which programming style will best take advantage of the system features and lead to optimum execution time for their application.

From the above paragraphs it is clear that *performance* expressed only as a single feature

of merit will frequently produce misleading results. The complexity and diversity of modern general-purpose computer systems in terms of both hardware architecture and software issues makes performance assess merit even more challenging. The nature of a single task or program made up of a number of programs running sequentially is not adequate to provide a rigorous single performance metric for a computer system. A solution to the weakness of a single workload task, commonly referred as a benchmark, is the introduction of a set of specific tasks-benchmarks each of which will assess different single performance characteristics of a computer system. Such sets of benchmarks are known as *benchmark suites*. Accordingly the performance characterisation of a computer system is expressed as a set of individual system component performance measurements, e.g. floating point unit, memory subsystem, communication subsystem, OS, compilers and so on. A common feature on which all performance metrics are based on is the time measurement of a task, e.g. execution time, throughput, etc. The performance of a computer is defined as the speed with which performs a well-defined task [176]. The performance of an X computer executing a task B will be then:

$$Performance_{X,B} = Speed(B,X) = \frac{1}{Execution\,time_{X,B}} \tag{5.1}$$

Consequently we can compare quantitatively two different computers X and Y which execute the same task T in x_B and y_B time respectively as:

$$\frac{Performance_{X,B}}{Performance_{Y,B}} = \frac{y_B}{x_B} = n \tag{5.2}$$

which means, in other words, that computer X is n times faster than computer Y. The first expression of equation 5.1 gives the absolute performance of computer X executing a task B while the second one (equation 5.2) is a relative performance between computers X and Y.

The plethora of benchmarks, workloads and the rapid evolution of computers over the past twenty years has often led to confusion and misinterpretations of benchmarking results which has sometimes led to a vigorous debate on the value of benchmarks within the computer community. Scientific benchmark suites need to provide essential and clear information for the performance evaluation and analysis of computer systems if they are to be useful.

5.3 Benchmark Objectives

The objectives of benchmark suites throughout the history of computing from single-computer systems to scalable parallel computer systems, remains the same: to evaluate the performance of a computer system and if possible to rank computer systems with respect to their suitability for a certain task [114]. Scientific benchmarks should not only provide a single representation of computer systems performance, but they should also provide further information about the way system components affect the overall system performance and an understanding of the internal behaviour of computer systems. Thus effective computer performance evaluation has to be expressed as a function of many interrelated considerations. Accurate characterisation of a benchmark is not only a matter of understanding the test program, it considers also the size of the problem, the algorithm, the ability of the compiler, the OS and the computer

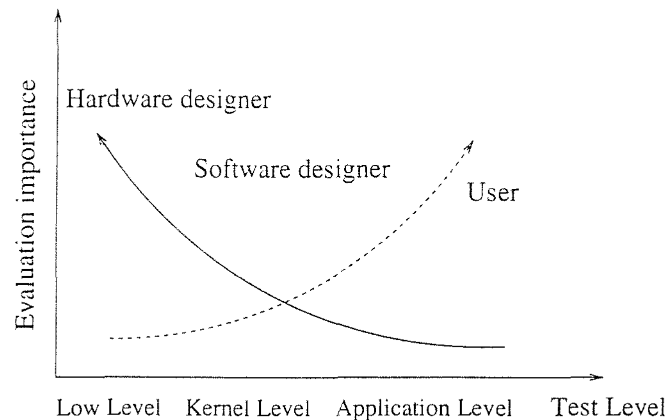

Figure 5.1: Different benchmark levels could have different importance evaluation interest

architecture. Constrains in hardware and software which will affect performance have to be taken into account, as well as the level of software tuning, e.g. compiler flags mainly for kernel and application level benchmarks. Van der Steen [227] defines all these considerations as the *workload*[1]. With reference to Dongarra et. al. [61] assessing performance evaluation properly will enable:

- An aid to designers of future and existing architectures
- An aid in reasonable characterisation of system capabilities
- Promote software development for efficient utilisation of existing architectures

Successful benchmark suites such as PARKBENCH [186] have followed a hierarchical bottom-up approach towards computer performance. On each stage or level of tests different system components are examined and evaluated. Initial stage tests use small programs to measure low-level machine performance characteristics, further stage tests evaluate larger system component performance until a level of loading which approximates to a real workload is reached. On each level of tests knowledge gained in previous stages is used to analyse performance. Various groups of people are also expected to have a different evaluation interests at each level of the benchmark tests as Fig. 5.1 shows.

According to Hockney [114] the accepted principles of rigorous inquiry for scientific benchmarks include:

- Objective experimental measurements and written reports
- Effective controls employed in experiments to isolate key metrics
- Carefully documenting environmental factors that might affect experimental results
- Providing enough detail in written reports to permit reproducibility of results

[1]Van der Stenn defines as a workload as the set of application that is representative for use of a certain computer system at a certain time and a certain place (system configuration time and place).

- Employing standard unambiguous notation
- Comparing results with other results in the literature
- Developing mathematical models that accurately model the behaviour being studied
- Validating these models with additional experiments and studies
- Reasoning from these models to explore fundamental underlying principles

In a similar way scientific benchmarks have to go beyond quantitative system evaluation, (e.g. speed, cost/productivity, etc.) and provide qualitative evaluation as well which is also important. This means that computer systems have to be evaluated for the context in which they are going to be used, e.g. application workload, compatibility, availability of software, interoperability and so forth. In addition scientific benchmarks should be "open" and promote innovative new hardware systems or software techniques.

5.4 Typical Benchmarking Metrics

Benchmarks are programs designed to run on a computer system to produce a relative measurement of their performance which can then be expressed and interpreted in various ways. In the past there has been considerable confusion over the definition of certain performance metrics which has been misapplied and misused [200]. The speed of a CPU was often rated in terms of MIPS (Millions of Instructions Per Second) or CPI (Clock Cycles Per Instruction). The high complexity of modern CPUs with pipelining and multiple-instruction issues incorporating super-scalar architectures together with modern compiler capabilities[2] means that such terms are at least ambiguous and elusive. Many benchmarking projects had failed to setup universally accepted standards, defined methodology or a result reporting scheme. In parallel systems performance metrics can be more complicated as many of them require additional parameters such as the number of processors p or the algorithm complexity $O(N, p)$.

Over the past several years there has been a considerable attempt to standardise and establish the computer benchmarking field on a scientific basis. In most cases the proposed benchmark metrics are based on a function of time and the workload characterisation type, e.g. $T(B)$. The PARKBENCH methodology addressed the problem of benchmarking metrics and proposed a well-defined set of units and standard symbols for expressing benchmark results which comply with the SI standard [114].

The most fundamental metric of a benchmark is time, the wall-clock elapsed time which can be measured on an external clock has a universal meaning and is supported with more or less resolution by all computer platforms. A timing expression of the type $T(N, p)$ for parallel machines will relate the elapsed time of an N size problem running on p processors. A typical performance metric derived directly from the execution time result is known as the Temporal Performance of a problem:

$$R_T(N, p) = T^{-1}(N, p) \tag{5.3}$$

[2] e.g. rearrangement of the instruction stream to avoid stalling processor pipelines, etc.

Table 5.1: Units and symbols used in PARKBENCH follow the extension of the SI system

Symbol/Unit	Meaning
flop	floating point operation
instr	instruction of any kind
intop	integer operation
vecop	vector operation
send	message send operation
iter	iteration of loop
mref	memory references
barr	barrier operation
b	binary digit (bit)
B	Byte
sol	solution or single execution of benchmark
w	computer word

McMahon in Livermore Loops [148] normalised the cost of primitive computer operations into benchmark floating-point operation (flop) count as:

1 flop add, subtract, multiply operations

4 flop divide, square root operations

8 flop exponential, sine, etc operations

1 flop conditional operations

The number of operations N together with the elapsed time required to solve a program $F(N)$ is frequently used as a performance metric to express the rate at which a hardware platform performs an operation. Benchmark performance then can be expressed as a function of *flop* operations and the elapsed time is expressed in units of Mflop/s as:

$$R_B(N,p) = \frac{F_B(N)}{T(N,p)} \tag{5.4}$$

For parallel benchmarks speed-up, efficiency and performance per node metrics are frequently used. Speed-up in general is defined as sequential execution time over parallel execution time. In benchmarks speedup is often expressed as the ratio of the time required for the benchmark to run on a uniprocessor implementation *T(N)* and the time required to run the parallel implementation on p nodes.

$$Speedup(N,p) = \frac{T(N,1)}{T(N,p)} \equiv \frac{T_1}{T_p} \tag{5.5}$$

In a similar way efficiency is defined as the fraction of time that processors spend doing useful work, it characterizes the effectiveness at which an algorithm uses the computational

resources of a parallel computer. Efficiency is defined as the ratio of speedup to the number of nodes:

$$Efficiency(N,p) = \frac{Speedup(N,p)}{p} \tag{5.6}$$

The definition of such related metrics is somehow ambiguous, either because the uniprocessor implementation of the performed algorithm does not exist or it could be considerable different from the parallel algorithm. Sahni et. al [191, 214] explicitly defines five types of speed-up: relative, real, absolute, asymptotic and asymptotic relative. *Absolute* related metrics are defined with respect to the uniprocessor time for the best-known algorithm, in practice the definition of the best-known algorithm for each problem is a difficult task. In *relative* speedup and efficiency the related metric is defined with respect to the parallel algorithm executing on a single processor. Hence relative metrics cannot be used to make comparisons on different systems. Despite that, relevant performance metrics can be used to study the individual performance characteristics of a parallel system providing that the uniprocessor benchmark of *T(N)* is clearly specified.

Another term very often used in benchmarking is scalability, according to Gordon Bell there are several types of scalability such as generation scalable, reliability scaling as well as problem and machine scalability. The ratio of maximum and minimum performance rates of different workloads on a system is called *speciality ratio*, for example if this rate is close to unity for a specific platform this means that all problems would be computed at close to the advertised rate. A benchmark suite even of a few single tests can produce several results (e.g. tests have to run a number of times for different parameters). Presentation of a large number of measurements in a concise and meaningful way is not straightforward, it usually requires results to be combined, typically with arithmetic and harmonic means and graphs.

The large number of hardware platforms and different software applications has overwhelmed the amount of benchmark results over the years and revealed the need for a public domain benchmark result database [132]. Netlib is an example of such a Performance Database Server (PDS) developed and maintained at the University of Tennessee and Oak Ridge National Laboratory. Benchmark data and distributions acquired from industry and academia are accumulated, classified and made available for public access [132].

5.5 Existing Benchmarks

A brief description of universally-accepted existing benchmarks is provided below, together with a discussion of key characteristics as assessment of potential usability for clusters. This survey does not include early work such as the Whetstones and Dhrystones benchmarks [233] (which introduced the concept of a synthetic benchmark), or the Gentzsch kernels in 1984 [84].

5.5.1 The Livermore Loops

Livermore Loops and kernels introduced the idea of an abstract workload [230, 73]. The benchmark introduced the concept of adjusting weight factors for each test to meet the needs of a

specific computer system. Results were then presented using the harmonic mean. However to assign such weights is problematic and because of this little used in practice.

5.5.2 The LINPACK Benchmark.

This benchmark [62] has started as a class of test programs for the software LINPACK project developed in 1979. The original class of problems was to solve linear systems whose matrices are general, banded, symmetric indefinite, symmetric positive definite, triangular, and tridiagonal square with sizes of N = 100, 300, and 1000 using the LINPACK numerically intensive routines SGEFA and SGESL. Results are given in Mflop/s

$$R_B(N;p) = \frac{F_B(N)}{T(N;p)} \tag{5.7}$$

where the number of operations is given as R_{peak} by:

$$F_B(N;p) = \frac{2}{3}N^3 + 2N^2 \tag{5.8}$$

and the system configuration has to be stated precisely, e.g. date, system, compiler, compiler options. Later on two other lists of results were added: a list presented speedup and efficiency information (the execution time of problems run on a single-node and multi-node are reported), and secondly the results of arbitrarily large size N problems added to enable parallel machines to achieve high performance, R_{max} and $N_{1/2}$ values are reported. The ratio of the optimised code over the original Fortran performance can show the effectiveness of the compiler. LINPACK benchmark results are regularly published over the Internet for a wide variety of hardware platforms. LINPACK tests tend to measure the peak performance of a system, which is usually different to the overall system performance [65].

5.5.3 The PARKBENCH Benchmark

PARKBENCH Benchmark [186] is an attempt to establish an acceptable set of parallel benchmarks for users and vendors of parallel systems, as well to set standards for benchmarking methodology, metrics and result-reporting. PARKBENCH includes codes of other benchmarks which suited its methodology, e.g. parts of the Genesis [113] and NAS benchmarks are included. The PARKBENCH committee accepted the wall clock time measurement of a program, for this reason it introduces tests TICK1,2 to validate real wall-clock time measurements. PARKBENCH tests have a hierarchical structure within three levels, Low-level, Kernel benchmarks and Compact applications.

Low-level tests measure performance parameters that characterise basic architecture and compiler software features. Low-level tests can be split into single-node tests and basic interprocessor communication properties of the system. Single-node tests establish the clock resolution, node computational intensity, memory bottlenecks, LINPACK and Livermore Loops benchmark tests can be used as well to evaluate completely the performance of a logic single node. Communication benchmarks test basic communication primitives such as peer-to-peer latency and bandwidth, synchronisation speed, etc.

Table 5.2: PARKBENCH tests

Low level		
RINF1	do loops for r_∞ and $n_{1/2}$	
POLY1	Computational intensity $f, f_{1/2}$	
POLY2	out of cache comp. intensity	
COMMS1	one way point to point comm.	Byte/s
COMMS2	two way point to point comm.	Byte/s
COMMS3	All to all communication	
POLY3	$f_{1/2}$over the comm. network	
SYNCH1	Synchronisation speed	barrier/s
Kernel Benchmarks		
K1	Matrix multiplication	
K2	Matrix transpose	
K3	SCALPACK routines	
K4	QR decomposition	
K5	Matrix tridiagonilisation	
K6	MG from NAS	
K7	3D FFT	
K8	CG from NAS	
K9	IS from NAS	
K10	paper & pencil I/O test	
Compact Applications		
CA1	PSTSWM	
CA2,3,4	LU, SP, BT from NAS	

Kernel benchmarks include a wide range of computation intensive type problems. Some of the benchmarks are taken from already existing benchmark suites such as Genesis and NAS. Kernel benchmarks include codes from matrix-matrix multiplication, Fourier transforms, partially differential equations, etc.

Compact applications are "reduced" versions of complete applications. Applications for the compact application suite are complete applications that produce results of research interest. The application codes have been extensively tested and validated on a wide range of parallel architectures and well documented.

5.5.4 The NAS Benchmark

NAS Benchmarks have been developed at NASA Ames Research Centre initially to assess various performance aspects of parallel computers for various NASA problems. These programs consist of five core benchmarks and three pseudo-applications. The programs were derived from routines generally used in computational fluid dynamics applications. The 5 core benchmarks

are: Embarrassingly parallel (EP), Multigrid (MG), Conjugate gradient (CG), 3-D FFT PDE (FT), and Integer sort (IS). The 3 pseudo-applications are: LU solver (LU), Pentadiagonal solver (SP), and Block tridiagonal solver (BT). The Benchmarks can run at 5 different problem sizes (S,W,A,B,C).

EP Embarrassingly parallel benchmark performs 2D statistics from a large number of Gaussian pseudo-random numbers, there is no communication among nodes (to measure the total floating point performance of a parallel system).

MG Simplified Multigrid kernel solving a 3D Poisson PDE, used for communication performance measurements

CG Conjugate Gradient computes the eigenvalue of a large sparse symmetric positive definite matrix.

FT Computes a 3D PDE using FFTs, testing long distance communication.

IS Integer sort program, integer arithmetic performance and communication systems are stressed.

LU Compact application performs an SSOR scheme to solve a 5x5 block lower and upper triangular system.

SP Compact application, no-diagonally dominant scalar pentadiagonal equations solved with a 5xr block size.

BT Compact application, contains a block triangular solver, similar to SP but different communication patterns.

Tests were precisely defined problems in a "paper and pencil" way, but no implementation was given. Hence the responsibility of the benchmark implementation was moved to system developers. Vendors were free to develop the tests in the best possible way according their system capabilities. Source and information about the NAS benchmarks can be found at the NASA NAS web site.

The original "paper-and-pencil" approach had some disadvantages as it could be seen as a benchmark of the "bench-markers", some vendors could skip parts of a calculation which were not interesting by using lookup tables, etc. The NPB version 2.0 released in 1995 includes a model implementation using a message passing model based on the MPI communication library. The NPB version contains 5 kernel (core) benchmarks and 3 compact CFD applications (pseudo-applications). Tests can be executed in three classes, A for small size, B and C for large size. Execution time for class A problems are normalised to the time required to run the tests on one processor on a Cray Y-MP, and class B equivalent normalised to Cray C-90.

5.5.5 The EuroBen Benchmark

The EuroBen benchmark [227] started in 1990 with the EuroBen Group. The objective of the EuroBen is to uncover and express the "performance fingerprint" of high performance computers

by developing a benchmark that will yield a good understanding and a performance profile on a wide range of different architecture systems.

The structure of EuroBen benchmark is split into four modules [226]. Contents of module "1" include test programs for single node basic operation performance, memory subsystem performance, as well as basic internode communication point to point tests. Module "2" contains simple basic algorithms tests such as matrix-vector product, linear system solution, eigenvalue and FFT problems. In module "3" there are tests encountered already in the first two modules which are more representative in scientific applications such as complex FFTs, very large very sparse matrix-vector multiplication, block relaxation problems, etc. Finally module "4", currently not active, was planned for application programs.

5.5.6 The Perfect Club Benchmark

The Perfect Club benchmark suite was primarily targeted for "supercomputers" obtaining metrics about CPU time, wall clock time and Mflop/s for compact application-level tests. Compact application tests from 13 scientific programs cover application areas such as computational fluid dynamics (CFD), chemical and physical modeling, engineering design and signal processing. Application tests intended to represent intensive supercomputer scientific workload could run in two modes, an application code with no alternation (baseline runs) and vendor-optimised application codes. Results report the harmonic mean of the MFLOPS (Millions of FLoating-point Operations Per Second) rate for each given program. The number of FLOPS for each program was determined by the number of floating point instructions executed in the CRAY X-MP using the CRAY X-MP performance monitor. Unfortunately the complexity of the applications and lack of application analysis made understanding the measured performance difficult.

5.5.7 The SPEC Benchmark

The System Performance Evaluation Cooperative (SPEC) Benchmarks started as a consortium of HPC vendors in an attempt to establish a well-accepted and used benchmark set of metrics [204]. The SPEC benchmark suite consists of a set of eight integer and ten floating point public-domain, non-trivial programs running under real conditions sufficiently large to stress computationally any system. Results are normalised to DEC VAX 11/750 and for the new release SPEC95 results are normalised to Sun SPARCstation 10/40. SPEC results report only the geometric mean of the testing programs as the relative performance figure, known as SPECint95 and SPECfp95. The lack of low-level evaluation indicates that this benchmark does not provide clear estimates about the basic hardware characteristics of a system [69]. The commercial importance of the SPEC benchmarks has often motivated vendors to add benchmark-specific flags into compilers for specific benchmarks including SPEC tests. Sometimes these transformations could result in incorrect code or even slow-down the performance of other applications [177]. The SPEC committee introduced a base-line performance measurement in order to eliminate such problems. Performance results from these benchmarks are made publicly available.

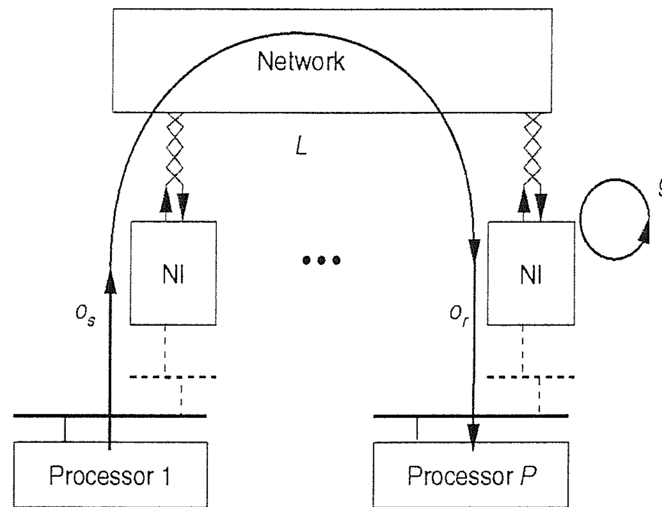

Figure 5.2: The LogP abstract model parameters; source [53].

5.5.8 The LogP Model

The LogP model originally was introduced as a realistic model for parallel algorithm design [52]. It describes an abstract machine configuration of key resource performance characteristics but does not rely on structured machine dependent details [144]. The cost model for a message passing parallel system is characterised by four parameters, three of them describe the time to perform an individual point-to-point operation and the other parameter provides a crude description of computing capability:

L: the latency or delay, is defined as the amount of time, for a small message, to be sent from the sender processor to the receiver processor on the network or the time needed to be processed by the network hardware.

o: the overhead, is defined as the time spent by the two CPUs engaged in sending or receiving a message. The overhead parameter can be described sometimes as a sender-side overhead o_s or receiving-side overhead o_r.

g: the gap, is defined as the minimum time interval between two consecutive message transmissions or consecutive message receptions at a processor side, the reciprocal $1/g$ corresponds to the available communication bandwidth.

P: processor is the number of processors/memory modules utilised.

The LogP model makes the assumption that the underlying network has a finite capacity which allows at most $\lceil L/g \rceil$ messages to be transmitted from one processor to any other processor at any time. If a processor attempts to transit a message that would exceed that network capacity the processor stalls until the message can be sent without exceeding that limit.

The simplest point-to-point communication operation thus requires a time of $L + 2o$ or $o_s + L + o_r$. The round trip time or a request-response operation will require: $2(L + 2o)$ [27].

Transferring n sequential small messages rapidly from one processor to another will require time: $o + (n-1)g + L + o_r$.

Parameter characterisation requires the use of a series of micro-benchmarks and careful analysis of the resulted graphs [53, 27]. Extracting the individual LogP parameters for a system is not always straightforward because they represent abstract quantities which might not map well into hardware structural characteristics, e.g. timing limitations cannot distinguish detailed measured parameters in many aggressive user-space communication protocols used in workstation clusters [18].

5.5.9 Other Benchmarks

Other benchmark categories such as proprietary benchmarks exist specifically tuned to evaluate the performance of commercial hardware features or applications and therefore cannot be used beyond their intended scope as standard benchmarks. The TPC benchmark suite is an example of a commercial data processing and database benchmark which primarily gives information about the throughput of commercial database transaction systems.

5.6 Comparison and Assessments

The benchmarking suites presented briefly in the previous section are classified into various groups according to their characteristics, e.g. the way they approach performance evaluation, or their internal structure. Historically benchmarks were developed to evaluate performance characteristics of supercomputers, where performance characterisation was important. Benchmarking requirements for architectures with advanced features are demanding because performance tuning on such systems is often critical [157].

According to the targeting hardware platform benchmark suites can be divided into two orthogonal characteristics: uniprocessor versus multiprocessor and vector processors versus scalar processors. Parallel processor benchmarks often invoke uniprocessor benchmark suites in order to analyse the performance of a single logic processor of their system. Benchmarks written for vector processors systems fail to provide a clear estimation of scalar processor systems performance because performance characteristics for vector pipelines and scalar systems with cache are very different. In a similar way performance characteristics for shared memory and distributed memory systems are significantly different [141].

The policy by which a benchmark suite approaches performance evaluation is another important characteristic of benchmarks. A hierarchical bottom-up approach of a system performance is obtained by splitting down the system components, analysing their performance and moving towards the top of the hierarchy.

Low-level versus high-level and synthetic versus application-level benchmarks terms are used frequently in the literature. Low-level tests are synthetic programs designed to measure basic architectural features of a computer system, e.g. the memory bandwidth, pipeline speed, etc. Tests of this level are simple to construct, port to a platform, or analyse. Results should provide the peak hardware performance of the system regardless of higher-level software capa-

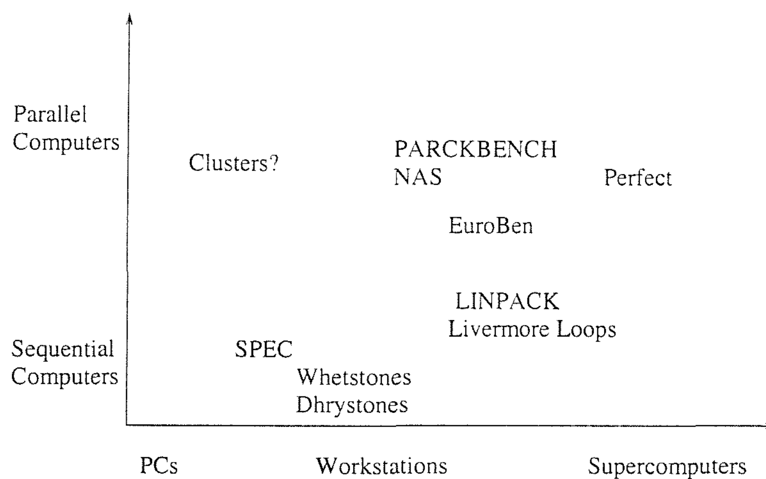

Figure 5.3: Benchmarking classification areas uniprocessor versus. multiprocessor and vector processor versus. scalar processor architecture

bilities. In many benchmarks low-level tests are extended to measure individual components and features of a computer system such as communication subsystems, processor performance, etc. Analysis of low-level tests will provide valuable information for the analysis of higher level tests.

Kernel-level tests are usually synthetic, although sometimes could be modified parts of applications or algorithms, i.e. such as found in a scientific subroutine libraries, introducing the concept of an abstract workload. Synthetic kernel benchmarks attempt to match the average frequency of operations performed for a large set of applications. Software aspects such as compiler technology clearly have a significant impact on performance. Results of these tests can give an indication of real working conditions or sustained performance but also could create misconceptions about the overall performance load.

Application-level tests can be either full scientific applications or stripped-down versions of real applications. Results from these tests have to avoid generalisation and can be expected to give the most accurate picture of a computing system performance only if low-level benchmarks performance is previously understood. Tests of this class have a number of disadvantages such as portability, analysis of the results is difficult and the danger of measuring programming style instead of performance.

5.7 Shortcomings of Existing Benchmarks

The characterisation of workload is probably the most fundamental key aspect of benchmarks. The influence of the specific underlying hardware features and the software compiler capabilities very often can alter significantly the workload. Benchmarks that fail on this characterisation will usually fail to meet their stated objectives as well. For higher-level benchmarks such as kernel-level and application-level tests, compiler tuning capabilities become a critical issue. This was the reason why many benchmarking suites introduced base-line tests, such as the

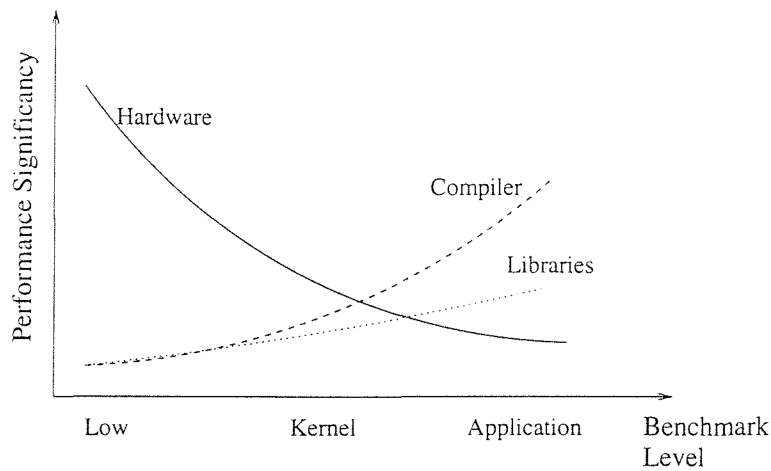

Figure 5.4: The importance of efficiency from low-level to higher-level tests from hardware to software issues

SPEC suite where no compiler optimisations are permitted, or an optimised open-line suite of tests. Hennessy and Patterson [107] refer to several cases in which Whetstone and Dhrystone benchmarks have produced overestimated results because certain compilers detected specific patterns and then optimised or simplified specific benchmark codes. Similar misleading results can be produced with benchmarks based on MIPS metrics [200] either because the workload has a different set of instructions or the size of the workload is different (e.g. it does not fit within a specific size inside the memory hierarchy).

Performance information provided by application level benchmarks is representative only for those particular programs. Hence the information obtained is sometimes inadequate to provide insight and predict the performance of other applications. Inter-platform comparison is also difficult for algorithmic and application level benchmarks as fairness issues concerning the underlying hardware architecture might arise. This is a common case in parallel systems where specific implementations of algorithms and applications might not adapt very well to the underlying system characteristics. The 'Paper-and-pencil' benchmarks introduced by NAS solve this problem in a way that allows parallel algorithms to vary between platforms by specifying only the numerical algorithm leaving the cost of the required implementation either to the user or the vendor. Most of the examined benchmark suites in this review fail to provide a concise performance evaluation over the wide range of HPC platforms.

5.8 Summary

The measurement and understanding of computer system performance has always been an important subject for computer developers and users. The main objective of a computer benchmark is to evaluate the performance behaviour of a computer system with respect to its suitability for a certain task-workload. However single workload benchmarks are inadequate to provide accurate performance evaluation for modern computers.

A collection of workload benchmarks which target computer performance behaviour at

different system levels known as benchmark suite can eliminate the single workload limitation but at the same time they introduce a variety of different performance metrics. In practice, existing benchmark suites such as LINPACK or NAS cannot provide a concise performance evaluation over the wide range of HPC platforms. The following chapters will examine the need for a tailored benchmark suite and its main objectives for workstation cluster platforms.

Chapter 6

SCOPE: a Tailored Benchmark Suite for Clusters of Workstations

6.1 The Requirement for a Tailored Release

The main target of a standard benchmark suite is to obtain general knowledge about the performance of a system over an application spectrum as wide as possible. Nevertheless benchmarks have certain limitations and drawbacks because all they measure is how fast the specific benchmark programs run whereas the performance of other applications is inevitably uncertain. More importantly, benchmarking requirements for HPC and parallel systems are higher. Full scale evaluation for such systems, to quantify and compare objectively, is difficult because of the wide range of their design space and demands for fine tuning performance settings [228].

Clusters of workstations have emerged as a parallel platform which has many similarities with MPPs but at the same time strong quantitative and qualitative differences from other parallel platforms. Although in the past few years differences between MPPs and clusters of workstations have tended to merge, MPPs have several potential advantages over clusters of workstations. The size and the quality of available resources per node is in favour of MPPs. For example the communication subsystem and the I/O bandwidth of MPPs have better characteristics from any distributed network system used for clusters in both terms of peak performance and sustained load. The memory hierarchy in MPPs usually is larger and has better performance characteristics than any commodity workstations memory hierarchy system [135]. Finally clusters cannot operate efficiently with tight global node synchronisation.

Another key point which can affect performance dramatically in HPC systems is the underlying software technology and the run-time system support. MPPs have to run usually in a batch mode environment while clusters of workstations usually run in multi-user interactive environments. Most of MPPs are released with highly-optimised compilers and programming libraries mostly written in the Fortran programming language. On the other hand the majority of workstation clusters are based on general-purpose software, typically GNU or freeware software, Linux, Free-BSD and C programming language with compilers which often do not provide

Table 6.1: Differences between MPPs and cluster

Characteristics	Cluster	MPP
cache size	512-1Mbyte	>2Mbyte
memory size	<64Mbyte	>256Mbyte
interconnection	Mbit/s	Mbyte/s
support/tools	-	exist
compilers	non aggressive	aggressive

specific optimisations or specific hardware feature support[1]. These differences in programming languages and runtime environment make the porting and running of benchmarks for MPPs on workstation clusters a cumbrous task.

Finally the lack of standardisation among the architecture of clusters is another key point of performance diversity. Despite the use of Commodity Off The Shelf (COTS) components the classification of clusters of workstations is rather loose and virtually every single cluster is built with its own individual architecture/configuration reflecting the nature of the targeted application. Frequently the behaviour of the underlying interconnection network used in clusters is often unpredictable and subjective to long delays and packet loss. Consequently, there is a need to examine closer the performance behaviour of the interconnected network for each cluster. On the contrary the interconnection network on MPPs is usually regarded by benchmarks as an opaque subsystem which is fast, efficient and reliable.

Existing HPC benchmark suites for message-passing systems are designed primarily for Distributed Memory or Shared Memory MPP systems rather than clusters. Most of these benchmarks, in principle, run also on clusters of workstations simply because clusters support the identical programming model as MPPs. Although theoretically the above condition is sufficient for an MPP benchmark to run on a workstation cluster, it does not necessarily provide useful information and understanding about specific performance characteristics of clusters of workstations. The size of a benchmark workload designed for MPP in general does not fit within the limited amount of resources of a commodity workstation cluster node. Therefore many of the tests used in existing benchmarks are not suitable for networks of workstations.

For those reasons existing benchmark suites misinterpret the measured performance of clusters and they often fail to provide meaningful performance information. This means that conceptual issues in the performance measurement of workstation clusters are confused and misunderstood. This chapter proposes a new benchmark suite called Specific Cluster Optimisation and Performance Evaluation (SCOPE) which will address performance evaluation issues specifically for workstation cluster characteristics such as interconnection network, message-passing calls efficiency, heterogeneity and provide cluster administrators and programmers with a useful performance evaluation tool.

[1]For example the GNU GCC compilers (especially on CISC architectures) have a rather conservative inter-procedural alias analysis, while specific RISC-architecture commercial compilers such as GEM from DEC for Alpha processors could be more aggressive and produce highly-optimised code [159].

6.1.1 SCOPE Requirements and Objectives

The initiative of this chapter is to propose and define a benchmark set of performance tests suitable for clusters of workstations which is straightforward to understand and use. The objectives of the SCOPE benchmark suite is to provide a comprehensive set of benchmarks that is generally accepted by both cluster users and cluster managers. The benchmark suite will assist cluster designers to understand the behaviour of workstation clusters and hence develop better systems out of COTS as well as provide application programmers with a comprehensive tool and a methodology to develop and tune parallel applications for clusters of workstations more efficiently[220].

The SCOPE benchmark suite will make use of well-established benchmark tests together with new tests to address performance evaluation and modeling aspects for workstation clusters at the single-node level, internode communication-level and programming model level. The SCOPE benchmark introduces a comparison between network-level and communication-library level tests, as well as the notion of the performance evaluation of message-passing routines and operations within the context of an algorithm execution. The benchmark suite has to be broad enough to accommodate variations of clustered systems as well as to stimulate experimentation with homogeneous or heterogeneous clusters of multi-node uni-processor or multi-processor (SMP) nodes [31].

The proposed SCOPE benchmark suite requirements and objectives are summarised as follow:

- Establish a simple and comprehensive set of benchmarks suitable for clusters of workstation systems that will provide meaningful information to cluster users and administrators.
    - Easy to use by both administrators and programmers
    - Realistic comparisons taking into account system configuration and resource availability
    - Representable workload for typical algorithms and applications
- Small in size with a low overhead. This implies that tests will not run for an excessively long time
- To adhere to existing standards for benchmarking, methodology and result-reporting
- To provide support for heterogeneous clusters
- To provide support for SMP multiprocessor clusters
- To run on MPP systems in order and provide comparative results
- SCOPE benchmark methodology will be expanded to provide support for cluster designers-administrators and cluster users, application programmers at non-privileged root mode:
    - User: wall-clock turn-around time of application.

- Numerical analyst: Speed-up, algorithm complexity, scalability prediction i.e. show the scalability of a current algorithm on a current system.
- Computer management: throughput, average turn-around time, utilisation, efficiency.

6.2 The Structure of the SCOPE Benchmark

The construction of the proposed cluster benchmark suite is consistent with the PARKBENCH [186] and Dongarra [61] benchmark recommendations and methodology which has established a scientific discipline based on a well-defined measures, units and workload characterisation. The philosophy of this benchmark suite is to provide system designers and programmers with information about the key characteristics of clusters they need to know: single-node level performance, interconnection-level performance, computational-model-level performance. According to the EuroBench and the PARKBENCH methodology, tests are classified into: low-level where communication primitives are tested, kernel-level in which computational model primitives are tested and application-level where compact applications stress the system. In addition the SCOPE benchmark introduces primitive network-level tests as well as tests at the algorithmic level along with kernel-level tests. Hence tests of the SCOPE benchmark are grouped into four categories following a hierarchical complexity structure:

1. Individual single node performance benchmarks i.e. include basic architectural benchmark tests. User-specific or architecture specific tests can be used as well to test and evaluate individual characteristics of nodes, for example I/O support. Well-established single node performance benchmark suites would be acceptable candidates.
   - LINPACK, SPEC95 benchmark suites can be used
   - User/architecture-specific individual node benchmarks e.g. STREAM [147], lmbench [205]
2. Low-level tests which include two subcategories:
   (a) Underlying communication network tests (at least two nodes must be involved) which examine the raw underlying network performance. The idea of these tests is to evaluate the performance at the low-level interconnection network, often called network raw performance, on top of which message-passing communication libraries reside.
      - Latency test (ping-pong)
      - Bandwidth (a bi-section bandwidth test may be required according to the network topology)

      Clusters with sophisticated network topologies and routing nodes will require extra tests for intermediate node latency and bi-section bandwidth evaluation. Such measurements are easily derived from the basic point-to-point latency and bandwidth features.

(b) Basic communication library tests (two or more nodes involved). These are classical tests of message-passing libraries based on a ping-pong principle.

- Peer to peer tests
  - Latency
  - Bandwidth
- Collective tests
  - Synchronise
  - Broadcast
  - Reduce
  - All to all

3. Kernel-level tests include two subcategories:

(a) Basic message-passing algorithmic kernel-like level operations and communication patterns are tested (two or more nodes involved). Some of these tests are already examined as low-level tests, at this level message-passing operations are examined more realistically from the context of an algorithm and not the artificial environment of a low-level ping-pong test.

- Shift operation
- Gather operation
- Scatter operation
- Broadcast operation

(b) Kernel-level tests include common algorithms used in many SPMD and data decomposition style parallel applications

- Sorting algorithm
- Relaxation algorithm
- Matrix multiplication

4. Application-level tests, these tests have not been implemented at this current stage.

6.3 The SCOPE Benchmark Methodology

The remainder of this chapter describes the methodology of the SCOPE benchmark in terms of methods, procedures, metrics and result presentation.

6.3.1 Benchmark Specification

The lack of standardisation and the diversity of cluster configurations has imposed restrictions over the provided usability of benchmark codes either on low-level or higher-level tests. To increase the portability and future upgrade-ability of SCOPE tests over different OS, supported

libraries and network protocols, the benchmark codes and tests will be provided also with a "paper and pencil" specification similar to that introduced by NAS. However, in this case there is a concern about the difficulty to distinguish between machine and programmer capabilities. One of the primary concern of the SCOPE suite is to reveal the real capabilities of a system and by covering or hiding system pitfalls (this should be a primary concern of an application developer).

Each test specification will contain an introduction to the objectives of the test and a detailed description of it. Specifications describe input parameters, output data format, timing procedures and benchmark-specific metrics, e.g. compiler issues. Implementation restrictions or other benchmark-specific characteristic tests also have to be described.

6.3.2 Performance Metrics

Based on existing benchmarks, methodology units and symbols for the SCOPE are adopted from section 5.4 and Table 5.1 are as follows:

- Central measure: wall-clock time $T(N, p)$
  - Measurements will use either a timer based on *gettimeofday()* calls and its derivatives used in communication libraries for example *MPI_ Wtime()* call or use of hardware specific assembly routines, with very low overhead. The last case enables new generation processors to make use of their precise internal timers which can simplify benchmarking and extend further benchmark and profiling usability.
- Tests will be repeated several times to overcome variable clock resolution and avoid other independent processor activity or noise of a multiuser-multitasking environment. Results will present the minimum time, the average time and the median time. The side effect of repeating a test several times is known as the cache *warm-up* effect which provides improved results when the test eventually resides in the system cache. Result fluctuation in long run tests is relatively small compared to the short run tests, hence there is no need to repeat them as many times as short run time tests. Results are stored in a text mode during or at the end of the execution of the benchmark. This will make result post-processing analysis easier for other tools.
- Derived measures:
  - *flops* = number of operations/time (operation count not known or defined)
  - *Speed-up* = (time for 1 processor)/(time for a P number of processors) see equation 5.5. We refer only to "relative speed-up" which is based on the execution time of the parallel algorithm running on a single node/process of the target computer [120]. Interpretation and analysis of speed-up and efficiency measurements should be used as a performance metric independent of runtime [191, 214].
  - *Efficiency*=Speed-up/number of processors (see equation 5.6) an indication of the utilisation of system resources[2].

  - *Effective bandwidth* this is a metric defined and used, in this thesis, for analysing collective calls results. As effective bandwidth is defined as the ratio of the aggregated message size $n*p$ (i.e. payload only) of all data transfers taking place among the processors within the period of this collective call, over the execution time of this call.

$$B_{eff} = \frac{\sum_{i=0}^{p=1} n_i}{t(n,p)} \tag{6.1}$$

    where p is the number of processors participating in this collective call and n is the payload involved in the operation.

- Analysis of the results of the SCOPE benchmarks will use the following approach
  - Text mode (tabular presentation) whenever is straightforward to make comparisons on certain points
  - Graphs provides a concise representation for performance results
  - Modeling and curve fitting if possible for machine-specific features

6.3.3 Software Requirements of SCOPE

Considering the variety of cluster platforms the use of standard software packages is an essential requirement. The wide use of public domain packages will be used for baseline tests as well as for some analysis and presentation of results.

- C compiler (optional FORTRAN compiler)
- Network libraries e.g. TCP/IP sockets, etc.
- MPI communication libraries (PVM could also be used)
- Shell scripts, tools necessary for the analysis and presentation of results e.g. *gnuplot*, *Perl*, *awk*. Analysis and presentation of results is not necessarily performed on the target platform.

6.3.4 Implementation Rules and Optimisation

The role of sophisticated compilers to achieve the maximum performance from advanced processor features is crucial. The ability of a compiler to exploit hardware features, e.g. the available of processor registers, has a strong impact on the performance outcome especially for kernel-level and application-level tests [159]. It is essential to ensure that compiler optimisations do not introduce any code elimination, alter the workload or produce erroneous results. Processors that support instruction level parallelism (ILP) have the potential to change the instruction order of critical parts of a test program and alter the workload. For this kind of side-effects an

[2] Both speed-up and efficiency are relative metrics in this context and should be interpreted according to the runtime.

anti-warping serialisation can be applied on critical parts of code to preserve the right execution order of the workload.

In order to establish a common performance baseline set, each test has as a default optimisation (the *-O2* level) and any other optimisation or compiler flags will be stated as an optimised performance metric/measurement. The later case will provide developers and programmers the chance to assess and experiment with the performance of non-standard features e.g. in experimental test mode.

6.3.5 Time Measurement and Considerations

Time is a fundamental metric for every computer benchmark and the basis of many tools for code optimisation. The timing of a task can be expressed in terms of user CPU time and system CPU time. User time is the amount of CPU time a program itself took to execute while system time is the amount of CPU time that OS routines took place to service requests made by the program. This time measurement technique in multiprocessing and multitasking systems, such as clusters of workstations, is not suitable for benchmarking purposes because it ignores any other kind of CPU activity which takes place during the test would yield unrealistic results. For the SCOPE benchmark suite time measurement will always refer to the clock time it took for a program to load, execute and exit (sometimes this is called wall-clock time, response time or elapsed time) which is a universally-accepted time measurement method.

The accuracy and precision of benchmark results depends directly on the timer accuracy used to take the measurements. Standard clock timer calls such as *gettimeofday*() defined in various system timer libraries return the current time typically with a resolution of the order of a few *ms*, which is a common figure for UNIX platforms. In practice the poor resolution of these calls restricts their usability for timing events that last for seconds or minutes. The *MPI_ Wtime*() communication library routine, used in many benchmarks to implement real-time stopwatches, has based its portability on such timer calls to profile program performance.

Existing profiling or timing mechanisms based on such routines have several disadvantages because measurement techniques are subject to noise and long overheads due to the system calls involved. In addition such techniques introduce cache and memory pollution. In a multitasking OS, measurement fluctuations can also occur under heavy-load conditions as well. Fine-grained measurement is often not possible because of the poor clock resolution. To overcome poor resolution problems and time a segment of program that requires little time to run we can repeatedly execute that segment of code within a loop many times. The resulting time can then be divided by the number of times the segment was executed to obtain a realistic average elapsed time. Unfortunately this technique, of enclosing the program segment within a loop, can also produce misguiding or unrealistic results because cache warm-up effects can take place. Moreover sophisticated preprocessors and compilers are able to detect that no useful computation is being performed and eliminate the code entirely. Therefore even for a single piece of code, profiling and benchmarking can become tricky and complicated as additional loop overhead estimation and subtraction from the final result is required.

For example, during the measurement of a test the time overhead of the *gettimeofday*()

Table 6.2: Timing Registers of modern processors, for more information see Appendix B

Processor	Register	width	instr.
Pentium Pro	TSC	64	rdtsc
Pentium II/III	TSC	64	rdtsc
DEC Alpha	PCC	32	rpcc
HP-PA 1.1	CR16	-	mfctl
R10000/R12000	?	?	?
Ultra Sparc I-III	TICK	32	rdtick
RS6000/PowerPC	MFTB	64	mftb

call is the time taking to exit the call at the start of the timing interval plus the time to enter *gettimeofday*() at the end of the timing interval. Loop overhead calculations in practice requires the timing of two separate loops, one loop with a single instance of the expression *T1* and the second loop with two instances of the expression *T2* as:

$$T_1 = N_1(loop_ov + expr) \quad (6.2)$$

$$T_2 = N_2(loop_ov + 2expr) \quad (6.3)$$

The loop overhead and the timing estimation of the expression according to the above equations 6.3 becomes:

$$loop_ov = \frac{2T_1}{N_1} - \frac{T_2}{N_2} \quad (6.4)$$

$$expr = \frac{T_2 - T_1 - loop_ov(N_2 - N_1)}{2N_2 - N_1} \quad (6.5)$$

The proposed SCOPE benchmark timing procedure has the following structure:

1. `start timing`
2. `perform benchmark computations`
3. `end timing`
4. `measure overhead`
5. `compute benchmark run time`

The above structure implies that computed time is greater than the timer resolution and the measured overhead otherwise the benchmark computation execution stage has to be repeated over a number of iterations.

The majority of modern processor architectures, see Table 6.2, provide time stamp counters (TSC) built into the processor in order to facilitate hardware performance profiling. These registers are incremented at the clock frequency of the CPU or an integer fraction of it. TSCs are long enough (64 bit[3]) to provide a guaranteed monotonically-increasing timestamp which

[3] On a 400 MHz processor clock frequency, this would give a time between register overflows of: 2^{64} cycles * (1 second/400,000,000 cycles) or over 1000 years

can be accessed easily with an inline assembly-language statement [118]. On a system reset, the Time Stamp Counter (TSC) is set to zero. To access these counters, programmers can use a simple 'read timestamp counter' instruction. The number of counts can be easily transformed to time given the frequency of the processor clock:

$$time = \frac{number\,of\,clock\,cycles}{processor\,clock\,speed} \tag{6.6}$$

The advantage of *register timers* is the very low overhead of their profiling mechanism, typically a few clock cycles only and the resolution accuracy is on the order of *nanoseconds*. The disadvantage of this mechanism is portability on different hardware platforms, this is because assembly language calls are involved which are processor dependent. However, portability should not be a problem because the development of such inline assembly calls is trivial and the number of different processors used for clusters of workstations is small and limited (see Table 6.2). Processor clock speed estimation in a similar way is the number of clock ticks divided by the number of elapsed time (Appendix B has an example). In superscalar processors with instruction level parallelism (ILP) and out-of-order execution support this profiling technique might require some need extra serialising instructions (anti-warping) to ensure the right execution order of the critical timer source code. The following lines of code give an example of how to use timer assembly calls on an Intel Pentium processor.

```
unsigned long timer_start, timer_end;
    double duration;
    . . .
    asm("mftb %0":"=r"(start_t):);    /* read timestamp counter */
    . . .
    . . .                             /* perform the operation */
    . . .
    asm("mftb %0":"=r"(end_t):);      /* read timestamp counter */
    duration = (end_t-start_t)/MHz;   /* in microseconds        */
     . . .
```

6.4 SCOPE Single Node Tests

The following sections will present a detailed overview of the SCOPE benchmark suite tests following the hierarchical structure presented in section 6.2. Single node tests are intended to measure the performance of a single node-workstation of a cluster, such kind of benchmarks are known as *basic architectural benchmarks* [114, 227]. Workstation characteristics such as the arithmetic unit performance (e.g. pipeline), disk and memory subsystem performance should be tested at this level (e.g. memcopy, or NCAR [102] memory bandwidth tests, etc). It is essential that benchmarks do not exceed resource availability. Heterogeneous clusters of workstations have to execute these basic architectural benchmarks for every different type of node they incorporate.

Several well-established benchmarks that provide information on the hardware/software performance of workstations can be used as the SCOPE single node tests. RINF1 and POLY1/2 from PARKBENCH, LINPACK, Livermore Loops, SPEC95, STREAM and lmbench are good examples of single node hardware performance tests. At this stage the proposed benchmark suite does not provide additional single node tests.

6.5 SCOPE Low-level Tests

The computational model of distributed memory systems (DM) is based on the ability of system nodes to communicate among themselves. Therefore the communication subsystem performance on DM systems is an important issue which can affect the overall performance of the system substantially. Parallel benchmark suites for DM systems have realised this importance and provide various types of low-level benchmarks that load and test the communication subsystem. The COMMS benchmark set from PARKBENCH [114] and Genesis [1] benchmark suites are classical examples of *ping-pong* type communication benchmarks (similar type benchmarks are included in the EuroBen suite module 1e).

Communication tests usually run at the level of computational model, e.g. the message-passing level and assume that the efficiency of underlying communication subsystem is always granted. In clusters of workstations this assumption is not always true, for example the underlying network technologies presented in Chapter 3.1 exhibited large variations in both performance and efficiency when used in clusters. Hence classical *ping-pong* benchmarks at the computational model level do not provide meaningful information for the communication bottleneck or other communication performance tradeoffs in clusters of workstations. For this reason, in order to emphasise the importance of the internode communication the SCOPE low-level communication tests introduce an additional set of network-level tests to the low-level set:

- The network level test stresses the network subsystem and is designed to measure the "raw" performance of the network protocol e.g. latency and bandwidth of the underlying Ethernet network.
- Communication library tests will provide information about the communication library performance delivered at the application level. Tests in this category will follow a hierarchical structure.

The primary objective of low-level network and communication benchmarks of SCOPE is to load and test the communication subsystem. In order to achieve this objective low-level test use techniques such as cache warm-up, buffer alignment, preposting receive calls, etc. A performance comparison between network-level and communication-level measurements is possible providing that the extra overhead of the message-passing call is taken into account e.g. for MPI/LAM this overhead is 32 bytes.

6.5.1 The Underlying Network-level Performance Tests

Network protocol developers do not always have a standard way of testing the performance of their protocols. As a result many network benchmarks were presented with contradicting metrics that do not clarify exactly the methodology of their tests, e.g. latency and bandwidth might refer to round trip time (ping-pong) or alternatively data-hose tests (ping only). Other optimisations which can strongly influence the outcome of results are memory page size, buffer alignment to memory pages, prepending receive calls, etc. One of the key objectives for this level of tests is to establish the necessary methodology required for such tests which aim to standardise and evaluate point-to-point communication performance at the underlying network level.

Performance results for latency and bandwidth at this level will show clearly the capabilities of the underlying network subsystem. A direct performance comparison between the hardware specifications, network level tests and the computational model level test is now possible, which will provide valuable performance information within the multi-layered structure of the cluster subsystems.

Portability at this level of tests in SCOPE is not always guaranteed because different network protocols often provide different APIs (for example compare the performance between the TCP/IP socket interface with the Active Messages interface). However, tests at this level are relatively simple as they test communication primitives between two nodes and do not involve complex blocking or non-blocking communication modes. A *ping-pong* loop which measures one-way round-trip time (RTT) between two adjacent nodes will suffice to provide latency and bandwidth information for the underlying network subsystem. In their simplest way a node (master) sends a variable length M message to another node (slave) and the slave immediately returns that message back to the master. The following pseudocode illustrates this ping-pong principle:

```
MASTER-NODE                                  SLAVE-NODE


    Initialise                               Initialise
    start := GetTime();
     for I := 1 To N Do                      for I := 1 To N Do
        Send(message);        ------->         Recv(message);
        Recv(message);        <-------         Send(message);
     end Do                                  end Do
    stop := GetTime();
    print (stop - start) / N;
```

After a number of repetitions time can be collected outside the repetition loop. The minimum send-receive time divided by two for zero-length message is reported as latency. Data rate or bandwidth is calculated from the number of bytes sent divided by half the round-trip time [63].

$$Latency(N) = t_{RTT}/2 \quad (6.7)$$

$$Bandwidth(N) = 2N/t_{RTT} \quad (6.8)$$

Various communication models [63, 186] have been developed in order to evaluate communication among processors in parallel systems. Applying the linear approach of [63, 186] with a start-up time α (constant per segment cost) and a variable per-byte cost β, the time required for the underlying network to send/receive a message t_n^u is given by[4]:

$$t_n^u = \alpha + \beta n \tag{6.9}$$

In analogy to Hockney's model for point-to-point communication operations $\beta = 1/r_\infty^u$ and $\alpha = n_{1/2}^u/r_\infty^u$ where r_∞^u is the asymptotic underlying network bandwidth which is measured for an infinite message length and $n_{1/2}^u$ is the half performance length that is the message length required to achieve half of the asymptotic bandwidth equation 6.9 will become:

$$t_n^u = (n_{1/2}^u + n)/r_\infty^u \tag{6.10}$$

The message length at which half the maximum bandwidth is achieved ($n_{1/2}$) is an important indication because it demonstrates the capability of the system to exchange short messages effectively. Parameters that can influence tests and measurements have to be considered for each platform in order to analyse the results better, i.e. measurements on non-dedicated clusters have to assess the effect of interference with other workload realistically.

6.5.2 Low-level Communication Library Tests

This level of tests targets communication performance measured at the programming model or at th application level. Results in this level are equivalent to the COMMS benchmark set of PARKBENCH and Genesis benchmark suites. Tests at this level can be divided into two groups, peer-to-peer tests which involve two adjacent nodes and measure latency and bandwidth and collective communication tests which require participation of more than two nodes.

Low-level communication library tests stress (in the case of NOWs) the network sub-system as well as other individual system characteristics e.g. CPU, memory, etc. Use of low-level tests, together with network-level counterpart tests, will provide a clear picture of the underlying system which can be useful to understand the behaviour of the system from the administrators and programmers point of view. Low-level communication library tests are portable and can also run in MPPs, e.g. IBM SP2, which also provides some useful results for better assessment and analysis of the performance of clusters of workstations.

6.5.3 Peer-to-Peer Tests

These tests are almost identical to network-level latency and bandwidth test. The main difference is the level at which they measure performance, in the former case it is at the network subsystem end and in the later one performance is measured at the communication library level.

The peer-to-peer latency test measures the time a node takes to send a sequence of messages to another node and receive back the echo, while the bandwidth test measures the time of a sequence of back-to-back messages sent from one node to another. In both tests receive

[4]Sometimes α is refered as start-up time t_s and β is refered as per byte cost t_w.

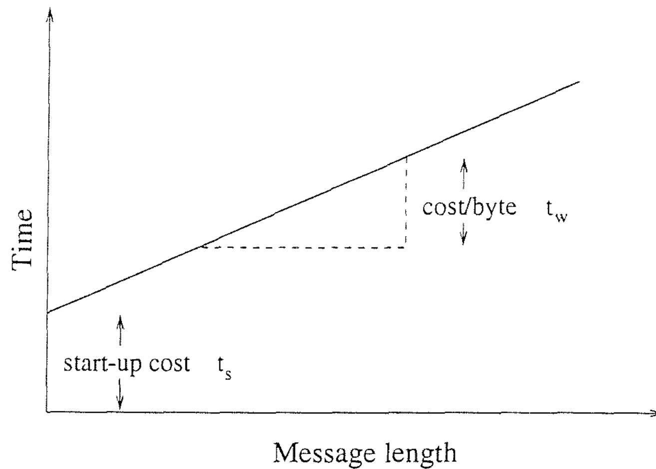

Figure 6.1: A simplified communication cost model $t = t_s + t_w n$

operations were posted before the send ones [134, 133]. *MPI_ Send* and *MPI_ Recv* functions are used for the latency test and *MPI_ Send*, *MPI_ Irecv* and *MPI_ Waitall* used for the bandwidth test. Latency and bandwidth performance can be expressed as a function of the message size, Hockney's parameters r_∞ and $n_{1/2}$ are directly applicable here.

$$t = (n + n_{1/2})/r_\infty \tag{6.11}$$

where the communication rate is:

$$r = \frac{r_\infty}{1 + n_{1/2}/n} \tag{6.12}$$

and the startup time is:

$$t_0 = n_{1/2}/r_\infty \tag{6.13}$$

In SCOPE tests, the message size always refers to the payload. Each test is repeated many times in order to avoid any clock jitter, first-time and warm-up effects.

Latency and Bandwidth Test

This benchmark is a straightforward ping-pong loop with a SEND - RECEIVE and RECEIVE - SEND MPI functions between two peers [186, 167]. The test can run for both blocking and non-blocking MPI communication modes. For the non-blocking tests there is no computational load overlapping on the nodes, so the results between blocking and non-blocking modes are not significantly different [165, 145, 66].

```
Initialise
start:= GetTime();
 for I:= 1 to N Do
   if(rank == root)
      Send(message);
      Receive(message);
```

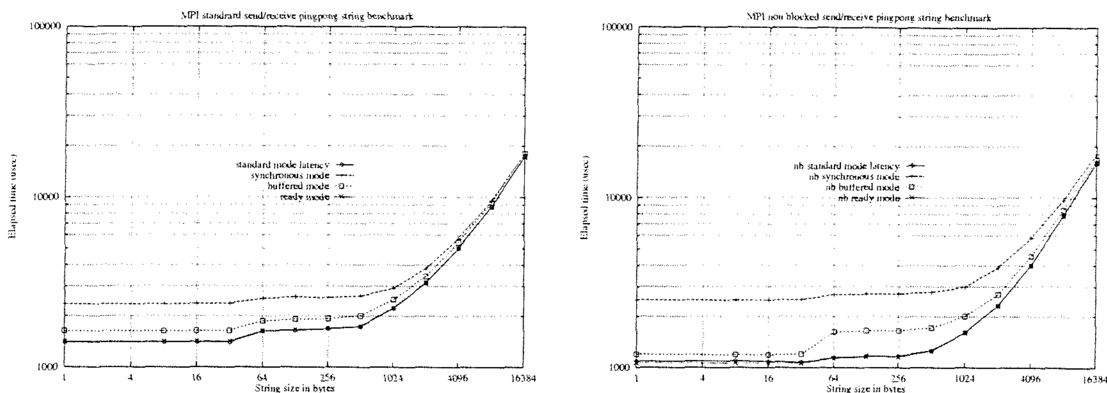

Figure 6.2: Latency of blocking and non-blocking MPI communication modes

```
    else
       Receive(message);
       Send(message);
    end if;
  end Do;
end := GetTime();
```

Elapsed time is measured either with the system function *gettimeofday()* or the *MPI_ Wtime()*. An example of the test run over two Sun SPARC II workstations connected with 10 Mbit/s Ethernet follows. The results for both blocking and non-blocking MPI communication modes are illustrated in Figure 6.2. The absolute values of these results are not so important as the difference between the two communication mode primitives (blocking and non-blocking). Ready and standard modes are faster than buffered and synchronous modes for both the non-blocking and blocking modes. Non-blocking communication tests are faster than the correspondent blocking mode calls (except the synchronous mode).

6.5.4 Collective Calls Test

Collective communication routines differ from point-to-point communication routines in several ways. They require coordinated communication within a group of processes which usually involves more than two nodes. MPI-1 collective calls are blocking, thus their implementation requires a lock mechanism usually implemented inside a protective communicator at the initialisation phase of each collective call which results in long initialisation overheads. Another common feature of these calls is their implementation on top of single peer-to-peer calls, therefore their performance is based on the efficiency of the algorithm implemented (e.g. binomial tree), the peer-to-peer call performance, the group size (p) and the underlying network architecture. For instance binomial tree-like algorithms require $\lceil \log_2 p \rceil$ steps for a collective all-to-one or one-to-all calls. In practice the current implementation of MPICH [95] uses a combination of binomial and sequential trees known as a *block-based binomial tree*. The structure of the tree is defined for the MPICH implementation by the MPIR_BCAST_BLOCK_SIZE variable which

is 1 for clusters of workstations and 3 for the SP2 systems.

The time for each collective operation routine is expressed in general as a function of the message size and the group size, i.e. the number of nodes participating in that call $T(n,p)$. Proposed collective routine tests for SCOPE are also suitable for Collective Communication Library implementations such as (MPI-CCL) [28] which utilise explicitly the native LAN broadcast mechanism for collective operations. Analysis of the results on these implementations differs according to the implementation routine mechanism. Collective calls can provide a good scalability measurement of the systems capability to sustain positive speedup in proportion to the number of nodes. Collective routines can be divided into three sub-classes:

- synchronisation (i.e. barrier call)
- data movement-I (i.e. broadcast call)
- global computation (i.e. reduce operation call)
- data movement-II (i.e. all-to-all call)

The barrier call tests only the synchronisation primitives of a system (there is no data movement among nodes). The other collective calls require data movement.

Synchronisation Call Test

The barrier call is a collective synchronisation routine, each node is blocked until all the nodes within the group have reached this barrier call. The SCOPE synchronisation call test measures the latency of the MPI_Barrier call $t_{bar}(p)$ for groups of different number of nodes. The MPICH implementation of the call is based on a butterfly communication structure algorithm with *log p* steps often represented as a hypercube. Practical issues require the implementation of a locking mechanism for each process with the initialisation of a protective communicator in which synchronisation with zero payload peer-to-peer calls will take place. Figure 6.3 illustrates the complexity of the underlying communication pattern which takes place for a barrier synchronisation call of a 9-node communicator using the MPICH implementation. The number of peer-to-peer calls required from nodes to exchange for this call is given by:

$$2(\lfloor \log_2 p \rfloor \cdot 2^{\lfloor \log_2 p \rfloor - 1} + p - 2^{\lfloor \log_2 p \rfloor}) \tag{6.14}$$

A simplified model approximation for a multiple channel communication network of the barrier synchronisation call can be given by 6.15:

$$t_{bar}(p) = (t_{oc} + t_{init} \cdot p) + t_s \lceil \log_2 p \rceil \tag{6.15}$$

where $t_{bar}(p)$ is the barrier synchronisation call latency for p nodes, t_{oc} is a constant startup cost and t_{init} is the variable initialisation overhead cost per process, the last term of the above equation represents the logarithmic part of the communication cost.

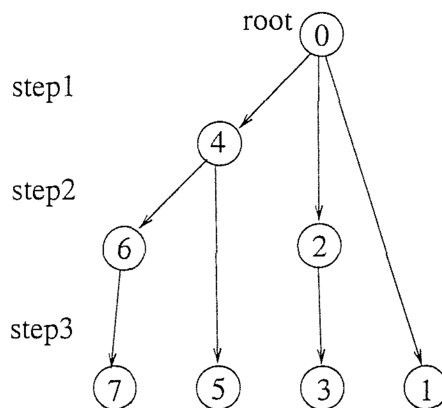

Figure 6.4: MPICH Broadcast call communication pattern involved on an 8 node communicator

(where $t_{bcast}(n,p)$ is the required time for the broadcast of a message size n on p nodes). The protective communicator initialisation phase is represented by the $t_{oc} + t_{init} \cdot p$ term. Each single message-transfer step takes $t_s + t_w n$ although in practice these steps are not clearly distinguished for short messages where noise and the start-up time t_s dominates the latency of the broadcast call.

Global Computation Test

In a global computation test the latency of a reduce routine is measured as a function of the group size and the message size. A reduce operation call is a global computation (or combine operation) collective call in which data flows from bottom-up, from the leaves to the root of the tree. In addition, nodes have to access locally the transferred data in order to calculate the partial results of the combine operation. For this reason the cost of a reduce operation is relatively higher than a broadcast call. In practice a latency measurement test for a reduce call is straight-forward. The implementation of any reduce operation involves the *root* process in all of its execution steps, so a single *ping* test is sufficient to provide the necessary measurements for this collective call. The combine operation for the SCOPE reduce test is a single-clock-cycle logical operation (MPI_LOR).

The MPICH algorithm for the reduce operation is relative to the binary pattern of the rank of each process, i.e. if the least significant bit is 1 send to the node with that bit zero, if the bit is 0 then do a receive and combine. During the reduce test the cost of the combine operation remains constant, while the communication cost of the call depends on the network configuration. The reduce operation compared with the broadcast operation has an increased cost because of the extra cost $t_o n$ of the combine operation:

$$t_{red}(n,p) = t_{oc} + t_{init} \cdot p + (t_s + t_w n + t_{op} n) \lceil \log_2 p \rceil \tag{6.17}$$

(where $t_{red}(n,p)$ is the time required for the reduce operation on p nodes, each step takes $t_s + t_w n$ for a single message transfer plus the time required for the combine operation $t_{op} n$).

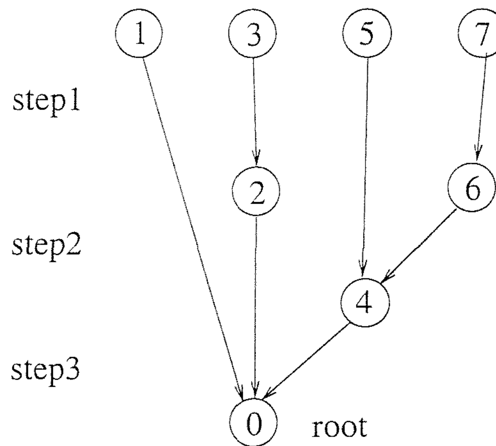

Figure 6.5: MPICH Reduce call communication pattern on an 8 node communicator

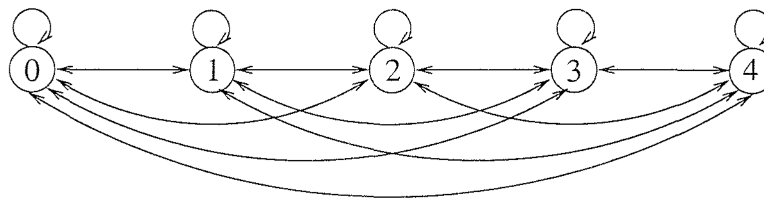

Figure 6.6: MPICH All-to-all call communication pattern on 5 node communicator

Data Movement-II Test

The complexity of this test is equivalent to COMMS3 benchmark of the low-level PARKBENCH suite. Its purpose is to measure the communication system performance under total saturation conditions and provide useful information of how the communication subsystem scales up with an increasing number of nodes. An all-to-all call, used for this test, is a data movement demanding routine because it requires each process (node) to send distinct data to every other process (node) and receive data from every other process accordingly. Figure 6.6 shows an example of the communication pattern involved with a 5 node communicator MPI *Alltoall* call.

6.6 SCOPE Kernel-level Tests

Traditionally kernel-level tests use algorithms or simplified fractions of real applications. Results of these tests are not sufficient to access completely the performance potential of a parallel machine on full scientific applications [114, 240]. However kernel-level performance tests can be biased in favour of particular parallel architecture features, for example NUMA or SM architectures. Information gained at this level of tests can provide a more realistic performance guidance for programmers and application developers.

The proposed SCOPE kernel-level benchmarks comprise algorithmic and operation tests. Kernel-level algorithmic tests include implementations of message-passing algorithms for matrix

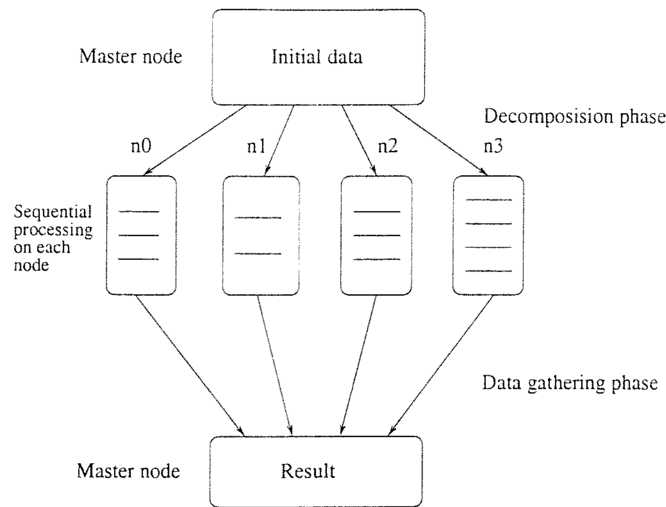

Figure 6.7: The data parallelism model with a domain decomposition phase

to matrix, sorting and multi-grid relaxation routines. Results of these tests can be used either to express the total elapsed time of the whole algorithm or partial performance on specific operation tests. Kernel-level operation tests provide information of the delivered performance of fundamental message passing operations such as broadcast, gather and scatter operations.

Algorithms used in kernel level tests do not use optimisations at the programming level, such as loop unrolling or blocking segmentation. The compiler optimisation used throughout the SCOPE tests is at level -O2 which is widely accepted for benchmarking. The following paragraphs describe briefly the implementation of the algorithms used as kernel-level benchmarks.

6.6.1 Kernel-level Message Passing Operation Tests

The single program multiple data (SPMD) model is an example of data parallelism used in MIMD and SIMD machines and clusters of workstations [185, 125]. In contrast to functional parallelism, data parallelism depends on the size of the problem because the entire data domain is partitioned among individual processes. This directory contains tests such as broadcast, scatter, gather and shift which are used in applications with data parallelism as Fig. 6.7 illustrates.

Although some of these calls have been examined previously in low-level tests there is a performance gap between low-level tests and real applications. Kernel-level message-passing operation tests are designed to examine the performance of these calls from a different level within the context of an application or an algorithm at the kernel-level. Results from kernel-level measurements are more realistic and closer to the effective performance delivered at the application level. Tests at this level measure the actual overhead of an operation at the programming level. In addition, some of these operations are often encountered as a combination of single peer-to-peer calls (e.g. shift operation or vector and stride scatter and gather operations) or even associated during application initialisation phases with preliminary communication data

exchange and datatype definitions, etc. Hence examination and benchmarking the performance of these tests will be very useful for application developers and system designers which can be combined to predict other application with data parallelism execution cost or scalability.

There are significant prospective differences between kernel-level tests and low-level tests. The construction of kernel-level tests does not always guarantee a pending receive call for each send call, something which undoubtedly improves performance in low-level such as ping-pong tests [97]. All kernel-level operation tests are stripped-down versions of kernel-level tests with the computational part omitted. The core of a kernel-level operation test does not have a loop within which the operation under test is executed for a number of times to overcome poor timer resolution. Instead the whole core of the test including initialisation phases, buffer allocation, is repeated together with the operation under test. The timer has to run only for the targeting routine or any other functions directly bound to that operation, e.g. buffer allocation, datatype definition, displacements, etc. The amount of data each call has to transfer to and from other nodes is relatively large, hence the disadvantage of poor timer resolution is not a primary issue. The use of register timers as a profiling method, presented in section 6.3.5, will provide more accurate results and shorten the length of tests.

The operations which are measured via the SCOPE kernel-level tests are:

- broadcast data
- scatter data
- gather data
- shift data among nodes

6.6.2 Kernel-level Broadcast Test

This test emulates an application which has to broadcast an array of size $N \ X \ N$ among a number of p processes. The structure of this test is based on the allocation of a matrix which will be initialised randomly on the root process. Then the root process has to broadcast the matrix into all other processes within its communicator. Accordingly each node in advance has to allocate a buffer area in order to run a bcast call. The structure of this test is similar to the low-level counterpart broadcast test. The matrix sizes for broadcasting varies N between 30 and 1080.

6.6.3 Kernel-level Scatter/Gather Operations

Scatter and gather are fundamental operations used in message-passing model to implement domain decomposition and composition phases. These two tests perform two almost-complementary operations. The first operation scatters a buffer in parts to all tasks within a group and the second one gathers together into the root node values from processes within that group. For kernel-level operation tests the vectorised versions of these routines are tested. The initialisation phase and explicit calculation of displacements are not timed during the tests.

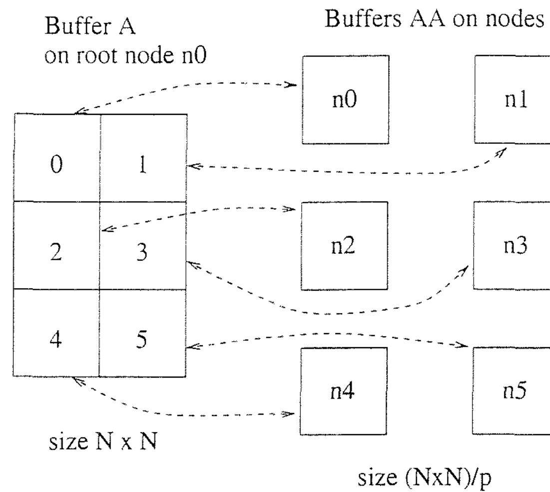

Figure 6.8: Gather/Scatter operations

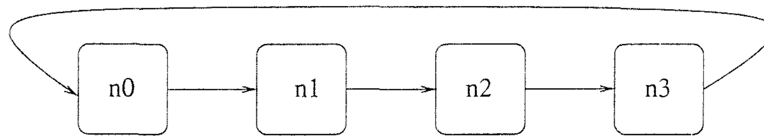

Figure 6.9: A shift right operation within all processors of a communicator

The scatter function has to decompose an array of size $N X N$ of the range 120–1680 and distribute the sub-arrays among the nodes of its group. The gather operation on the other hand has to transfer an array of size N of the range 120–1680 back into the root node.

6.6.4 Kernel-level Shift Operation Test

The need for a shift operation is frequently encountered in many parallel algorithms. The current shift operation test is a circular shift operation among p processors of the same processor array taken from the core of Cannon's matrix multiplication algorithm implementation. This test exchanges data between local nodes so it will provide a good indication about the scalability of a cluster for this particular form of local communication pattern. The actual shift operation starts with the identification of the processes that will participate in this call. For the successful implementation of the operation each node has to allocate an extra memory buffer for the receiving data. The exchange of data is completed within two peer-to-peer calls. Then locally each node has to swap its buffers at the end of the operation and return back the allocated extra buffer. Implementation of the call is tested for array size N ranging between 30 and 1080.

6.7 SCOPE Kernel-level Algorithmic Tests

This section of the benchmark suite includes a small set of kernel-level algorithmic tests which are included in a wide range of real parallel application algorithms. Kernel-level algorith-

mic tests will measure the overall performance of a cluster at a higher programming level. Kernel-level algorithms covered at this stage in SCOPE are two matrix-matrix multiplication algorithms, a sorting and a 2D relaxation algorithm. A particular attribute of these tests is the degree in which they can be analysed to provide performance details at an elementary level which can be applied to interpret more complicated algorithms later. Nonetheless algorithms in this module should be kept simple and must avoid becoming a benchmarking of the available software effort as happened in PerfectClub and other benchmark suites [120]. The following sections provide more detailed description about the proposed kernel-level algorithmic tests.

6.7.1 Matrix-matrix Benchmarks

Matrix multiplication is a fundamental component of many numerical and non numerical algorithms in various scientific applications. Matrix-matrix algorithms are highly parallelised by several algorithms and can assess the computational and communicational parts of a system. The naive algorithm of matrix multiplication has $O(N^3)$ complexity.

$$c_{i,j} = \sum_{k=0}^{m-1} a_{i,k} \cdot b_{k,j} \tag{6.18}$$

The sequential algorithm involves three nested loops and requires N^3 operations:

```
for i := 0 to N
   for j := 0 to N
      for k := 0 to N
         c(i,j) := c(i,j)+a(i,k)b(k,j)
      end k loop
   end j loop
end i loop
```

There are many ways to parallelise this algorithm, in this study we consider the Row/Column striped oriented algorithm and the Cannon's algorithm respectively. Both algorithms consider a two-dimensional decomposition of the original matrices A and B over p processors (nodes). Each process has to accomplish a task of $O(N^3/p)$ complexity. The difference among the two algorithms is the amount of partial memory required on each node and the number of interprocess communication steps required among nodes during the execution phase as well as the flexibility of the algorithms to make use of the available cluster resources. A modified version of the matrix multiplication algorithms that transposed the second matrix during the initial partitioning showed significant speed-up relative to the original code (2-4 times) for small numbers of nodes (results are presented in Appendix 203).

6.7.2 Row/Column Striped Algorithm Test

The Row/Column striped algorithm parallelises the outmost *for* loop of the naive sequential algorithm and requires a striped partitioning of the A matrix into *row* number of row blocks and a partitioning of matrix B into *col* number of column blocks. Then it involves N^2 dot products

which is the operation between a row of the A matrix and the column of the B matrix. After the dot product calculation each node has a sub-matrix of the C product matrix. Although the algorithm implementation is straightforward as soon as sub-matrices are distributed to nodes, the initialisation phase is complex because it requires the definition of several user datatypes and the use of different sub-groups of nodes or communicators in MPI terminology.

According to Foster [80], the execution time of a parallel program is the time that elapses from when the first processor starts executing on the problem to when the last processor completes execution. During execution, each processor is computing, communicating, or idling:

$$T_{tot} = T_{comp} + T_{comm} + T_{idle} \tag{6.19}$$

Adopting this formulae to our algorithm we have to add the extra time of new datatypes and communicators at the initialisation phase T_{init}, which may be expected to depend on the amount of datatype definition and the number of nodes p. In many cases the initialisation of new datatypes is done dynamically "on the fly" within the algorithmic phases, therefore the extra time overhead has to be credited to that operation.

Hence the equation 6.19 will become:

$$T_{tot} = T_{init} + T_{comp} + T_{comm} + T_{idle} \tag{6.20}$$

A more detailed analysis of this algorithm is presented in Appendix F.

6.7.3 Cannon's Algorithm Test

Cannon's algorithm follows a checkerboard partitioning N^2/p for each matrix and requires less memory on each node than the Row-Column striped algorithm. In order for a node to calculate a partial product it requires all blocks of its row and column to be systematically rotated among the processors, therefore the computation step requires $\sqrt{p} - 1$ steps with computation and communication rotation. The execution time of the algorithm is summarised in a similar way to Foster's analysis [80] as:

$$T_{tot} = T_{init} + T_{comp} + T_{comm} + T_{idle} \tag{6.21}$$

A more detailed analysis for both matrix multiplication algorithms is provided in Appendix F.

6.7.4 Sorting Routine Test

Sorting is a fundamental interesting problem in computing with numerous applications. Sequential comparison sorting algorithms have time complexity of $O(n \log n)$ (where n is the size of the array), but in general these are not easily paralleliseable and in addition do not scale very well. Tests of such algorithms can provide a good indication of a cluster performance as they include several all-to-all routines of random sizes. The SCOPE sorting algorithm test is an implementation of the "parallel sorting by regular sampling algorithm" (PSRS) introduced by Li et. al. [137] which has been effectively implemented on many MIMD architectures [185]. The

PSRS algorithm is a combination of characteristically important parallel operations among its nodes. The implementation of the algorithm is split into four stages. During the first phase the array is equally divided and distributed to all processes, each processor is assigned a continuous block of $\lceil n/p \rceil$ elements. Elements of these sub-arrays are sorted-out with sequential quick-sort algorithm locally. In phase two the root processor gathers and sorts samples from all locally sorted sub-arrays (nodes) and broadcasts *p-1* pivot values to every processor again. Each processor now has to partition the sub-sorted lists into sections according the pivot values. In the third phase processors exchange *p-1* partitions among them and rearrange again sub-arrays. In the final stage, each node merges its p partitions into a single list, the root processor is gathering (concatenate) all the lists in the final sorted list. The communication cost of the algorithm for an MIMD architecture implementation with p number of processors and an array size of n is: in the first phase p messages of $O(p)$ size, in phase two p pivots of size $O(p)$ while in phase three there are p processor sending *p-1* messages of size $O(n/p)$. The overall computational complexity of the algorithm is approximated to: $O((n/p)\log n + p^2 \log p + n/p \log p)$ which is asymptotic to $O(n/p \log n)$ when $n \geq p^3$ [137, 185]. See Appendix F for more details on the analysis of this algorithm.

6.7.5 Multi-grid Relaxation Routine Test

This routine is an example of a linear second-order partial differential equation (PDE) using a multi-grid iterative method to define an approximate solution. This is a rather simplified approximation of a two-dimensional Laplace equation on a rectangular domain using Gauss-Seidel-Relaxation:

$$\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} = 0 \tag{6.22}$$

where x and y represent coordinates in space, and f is the function that compute the values. This benchmark addresses the problem of processes exchanging and processing data locally.

The current multi-grid routine is based on a sample problem that acts on a two-dimensional grid of data values. The boundary values of the grid have fixed values (halo) while interior values are set to the average of all their neighbours:

$$f_{x,y} = \frac{f_{x-1,y} + f_{x,y-1} + f_{x+1,y} + f_{x,y+1}}{4} \tag{6.23}$$

The initial array of points is uniformly checkerboard distributed among processors into blocks of $(n/\sqrt{p}) \times (n/\sqrt{p})$ grid size. Each processor uses these values for boundary conditions and applying a normal sequential Gauss-Seidel averaging calculation on its own data. The next step of the algorithm is to check whether the convergence is "close enough" to a solution or to repeat another iteration. If a further iteration is required each processor exchanges its "boundary" values with its four neighbours (above, below, left, right) and repeats the above computational steps. The current test implementation on every iteration performs a global checking for the maximum change but does not abandon the loop unless a certain number of iterations is performed which guarantees a fixed workload each time the test runs. The

parallel implementation of the message-passing test is a mixture of Gauss-Jacobi and Gauss-Seidel methods known as a "chaotic" rubric [178]. A Gauss-Seidel iteration method is used for the data of each process, while data from neighbouring processes are "fixed old values" which resembles the Gauss-Jacobi method. Appendix F provides a detailed analysis of this relaxation algorithm.

6.8 Summary

This chapter has proposed and examined the Specific Cluster Operation and Performance Evaluation (SCOPE) benchmark suite. The main objective of the SCOPE benchmark suite is to contribute to the scientific benchmark methodology for a comprehensive examination of workstation cluster characteristics. The structure of the SCOPE benchmark suite is consistent with the hierarchical abstraction levels of well-known benchmark suites.

Single-node-level includes basic architectural benchmark tests to evaluate individual node characteristics. Low-level tests examine thoroughly performance at the communication level, there is a special emphasis on underlying network performance tests, e.g. TCP/IP sockets and on collective operations. Benchmarks at kernel-level provide performance evaluation closer to the user level. Kernel-level operation tests examine delivered performance of fundamental data parallelism operations such as shift, etc. Kernel-level algorithmic tests include implementations of message-passing algorithms, such as for matrix to matrix, sorting and multi-grid relaxation, used in a wide range of parallel applications and measure the overall performance of a cluster at a higher programming level.

At the current stage the SCOPE benchmark suite does not have any application-level tests because the implementation of such tests is beyond the scope of this thesis.

Chapter 7

Experimental Results and Analysis of SCOPE Benchmarks

This chapter demonstrates and analyses the SCOPE benchmark results obtained with the experimental implementation on a variety of workstation clusters. Computer architectures included in the clusters tested include SPARC workstations running Solaris, clusters of Pentium workstations running Linux or NT, a cluster of DEC Alpha workstations running NT and a cluster of SGI O2 workstations. Table 7.1 summarises the architectural characteristics and configuration issues for these clusters. Several benchmark tests also run on MPP systems (SP2, CS2) to provide reference points and assist in the analysis, evaluation and comparison of performance results in clusters of workstations although the benchmark suite does not target MPPs.

All the clusters presented in Table 7.1 can use the TCP/IP communication protocol suite and either the MPICH message-passing communication library or implementations based on it. Individual cluster characteristics such as a dedicated interconnection network, or dual network interfaces, were taken into account and tests of these characteristics are also considered wherever

Table 7.1: Cluster configurations used for testing with SCOPE benchmarks

Cluster	Node arch.	OS	Network config.	Dedicated	Nodes
Linux	Pentium PC	Linux/2.034	TCP/IP Ethernet	No	3
NT	Pentium PC	NT/4.0	TCP/IP Ethernet	No	8
Alpha	Alpha	NT/4.0	TCP/IP Fast Eth.	Yes	8
Solaris	SPARC-4	Solaris/2.6	TCP/IP Ethernet	No	12
Ultra-SP.	ULTRA	Solaris/2.6	TCP/IP Ethernet	No	12
Lyon/Eth.	PentiumP PC	Linux2.0.34	TCP/IP Ethernet	No	6
Lyon/Myr.	>>	>>	TCP/IP Myrinet	Yes	6
Lyon/BIP	>>	>>	BIP Myrinet	Yes	6
IRIX	O2	IRIX/6.3	TCP/IP Fast Eth.	No	32

Table 7.2: Individual processor SPECint95 SPECfp95 figures [204]

System	Processor	MHz	SPECint95	SPECfp95
SP2	Power2	66	3.23	9.33
SP2	Power2	133	6.17	17.6
CS2	SuperSPARC	40	>1	-
NT	DEC 21164	500	13.9	15.2
Linux	Pentium	200	8.09	6.75
O2	R10000	192	9.66	8.77
Ultra 1	UltraSPARC	143	5.87	8.38
SPARC-4	microSPARC II	110	1.59	1.99

possible. Measurements on "open" clusters (i.e. a non-dedicated intercommunication network) were obtained when there was no other user activity, if possible, on the network to avoid any interference and disturbance with the results and also affect the throughput of other user jobs [172]. Parameters that can influence tests and measurements are taken into account for each platform in order to analyse the results better, i.e. to assess the effect of other workload interference with some of our tests. The presence of background administrative workload "noise" is considered invariant, small and negligible for the purposes of the test results. The message size refers to payload. Each test is repeated several times in order to avoid any clock jitter, first-time and warm-up effects. The best time from each measurement test is presented in the results.

Before starting measurements on clusters it is useful to assess the SPEC benchmark performance levels for the individual nodes used in our workstation cluster platforms. As shown in Table 7.2 the equivalent SPEC performance marks [204] among these processors varies by almost an order of magnitude.

7.1 Tests on MPPs

This section presents the latency and bandwidth test results run on the SP2 and CS2 at the University of Southampton and on the SP2 at Argonne National Laboratory. Results of the low-level SCOPE benchmark tests on MPPs are used as a guideline to analyse and compare the performance of clusters. Higher-level SCOPE benchmark tests can also run on MPPs but the interpretation and analysis of the SCOPE test results on MPPs is beyond the scope of this thesis.

In both SP2 machines the native IBM MPI implementation was used while on the CS2 system the MPICH 1.0.12 version was used. The processors of the SP2 are Power2 Super Chip (P2SC) RS/6000 architecture which are superscalar pipelined chips capable of executing four floating point calculations per cycle. The SP2 communication architecture is based on a low-latency high-bandwidth two-level cross-bar switch (TB2 or TB3) with peak bandwidth for the TB3 DMA engine of 150 Mbyte/s [116]. Similarly the CS2 is an MPP machine with

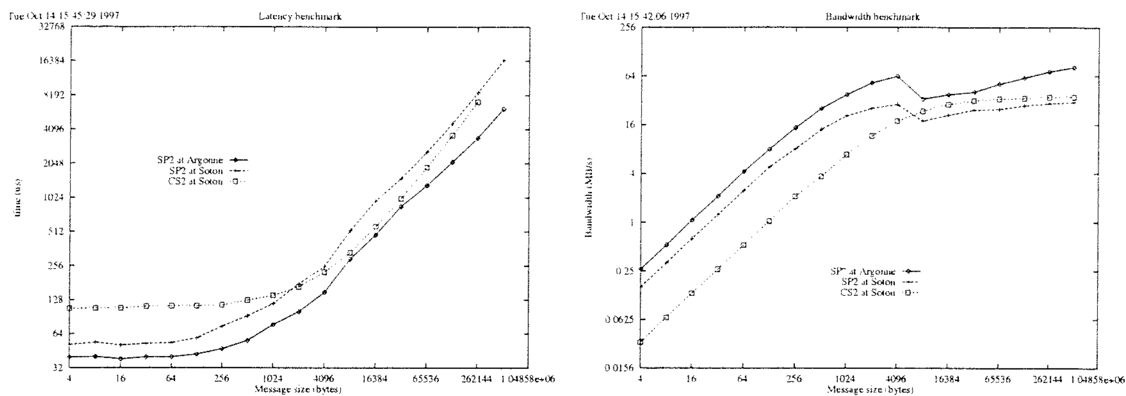

Figure 7.1: Latency and bandwidth of SP2 and CS2 Southampton and SP2 at Argonne.

superscalar SPARC processor nodes (microSPARC 110) while its communication cross-point switch is capable of providing 50 Mbyte/s link communication in each direction [218].

Figure 7.1 illustrates the SCOPE benchmark latency and bandwidth results on these MPP machines as a function of message size. The difference in performance between the two SP2 machines is due to the different type of high-performance switch (TB2/TB3). All three systems are batch systems and job allocation to nodes is done through a *scheduler*. The interconnection network is dedicated and graphs in Figure 7.1 illustrate smooth plots without fluctuations or anomalies. A closer examination of the SP2 graphs demonstrates a breakpoint for message sizes at 4 Kbyte, this is due to the different protocol policy used for sending/receiving small and large messages from the MPI implementation (which is specified by the MP_EAGER_LIMIT variable).

7.2 Low-level Communication Tests Results

This section presents the low-level network and low-level communication library test results. Low-level network tests targeting the underlying network API level performance directly. The TCP/IP latency and bandwidth performance has been measured in most of the target platforms. Minor modifications to these tests allow benchmarks to run on non-standard network protocol APIs such as the BIP protocol to meet platform portability requirements. Tests at the communication library level (i.e. the MPI level) did not require any modification across the range of the platforms tested.

Low-level communication benchmarks test for peer-to-peer and collective operations such as one-to-all broadcast, single-node accumulation (reduce) and barrier synchronisation among all processes within their communicator/group [134].

7.2.1 TCP/IP and Berkeley Sockets Interface Tests

Berkeley Sockets provide the main network UNIX API via which TCP and UDP services are made available at the application level. In API the initial objective was to present the network interface as a UNIX-like standard character device, in which sockets are file descriptors related to

the network device. In this way sockets are network communication endpoints, once connection is established (STREAM) can be used by standard *read*(), *write*(), *close*() OS functions.

The proposed SCOPE Berkeley socket-based benchmark has been designed around the classical client-server model and the ping-pong principles presented in section 3.2.2, with no computational steps. During the initialisation phase the establishment of the connection between the two nodes and exchange of control parameters takes place. When timers on both ends are setup, both nodes enter the main communication-intensive ping-pong loop.

7.2.2 The Linux Cluster

This cluster is composed of Pentium PC nodes (at 166 MHz and 200MHz) running Linux 2.0.x. The interconnection network is configurable, either as an *open* 10Mbit/s Ethernet subnet or a dedicated 100Mbit/s Ethernet segment. The MPI version used throughout the tests was MPICH 1.1. The ability of this cluster to reconfigure its network channel either as 10Mbit/s or 100Mbit/s and the fact that one of the nodes uses a dual processor, resulted in the running of three separate tests. One test is for the 10Mbit/s Ethernet channel (between two PentiumPro nodes), another test is for the 100 Mbit/s Ethernet channel (between a PentiumPro and a Pentium 166), and the third test uses the loopback interface of the dual processor node (Pentium 166).

Network Performance. The following Table and Fig 7.2 illustrate latency and bandwidth performance characteristics at the network level for the Linux cluster. The system could saturate the Ethernet channel just above 1 Mbyte/s only for message sizes larger than 64 Kbyte. The latency feature is moderate and rather high around 316 μs, the half bandwidth performance point is delayed as well at message sizes of 1 Kbyte or larger. Running the network test over the Fast Ethernet path we notice a significant improvement in latency which drops down to 90 μs but the effective bandwidth for larger messages is restricted below 5.5 Mbyte/s. Detectable breakpoints in the graphs occur between 1 and 2 Kbyte messages (Ethernet maximum packet of 1.5 Kbyte) and around 4 Kbyte because of the OS memory page length size. Applying the approximation model equation 6.10 and 6.11 for the Fast Ethernet channel yields a startup time of 195 μs and asymptotic bandwidth around 5.7 Mbyte/s.

Network performance of Linux cluster	min Lat.	max BW	$n_{1/2}$
Berkeley Sockets TCP/IP over Ethernet	316 μs	1.02 Mbyte/s	1 Kbyte
Berkeley Sockets TCP/IP over Fast Ethernet	90 μs	5.47 Mbyte/s	1.5 Kbyte

Communication library performance test. Latency and bandwidth test results at the communication library level (MPI) are illustrated in the following table and Fig. 7.3. Performance of these tests is degraded for both latency and bandwidth results compared to the raw network performance because of the MPI communication library overhead. For the Ethernet channel communication latency is increased to 638 μs while the bandwidth drops dramatically

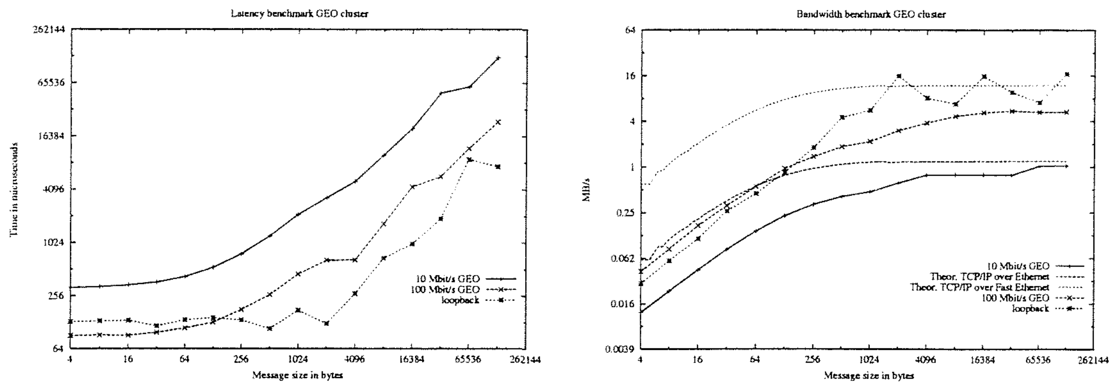

Figure 7.2: Network latency and bandwidth performance on the Linux cluster

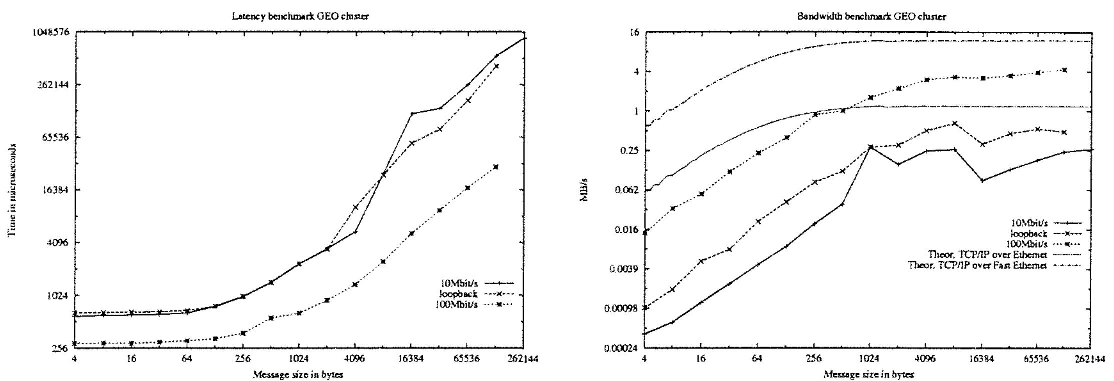

Figure 7.3: Latency and bandwidth of the Linux cluster

to 265 Kbyte/s with further unpredicted performance degradation in several points. Figure 7.3 illustrates a break point around 16 Kbyte, which is the default MPICH message protocol switching point, for all tested configurations. This is a feature of the MPI implementation which uses a different policy for *short* and *long* messages on MPI *send/recv* calls.

Performance results for the Fast Ethernet subnet, in comparison with the network-level performance test counterparts, are smoother and the MPI overhead does not degrade performance dramatically compared to the measured network level test performance. The overhead of the MPI library is around 200 μs, latency is 293 μs and bandwidth for large messages is 4.3 Mbyte/s. The half-bandwidth performance point $n_{1/2}$ is at about 1.5 Kbyte which indicates poor network performance for short-to-medium size messages. Further investigation of the OS and the TCP/IP implementation is required, see [127].

Configuration of a Linux cluster	min Lat.	max BW	$n_{1/2}$
MPI over TCP/IP and Ethernet	587 μs	265 Kbyte/s	1-1.5 Kbyte
MPI over TCP/IP and FastEthernet	293 μs	4.3 Mbyte/s	~1.5 Kbyte
MPI over TCP/IP using loopback	638 μs	624 Kbyte/s	1-1.5 Kbyte

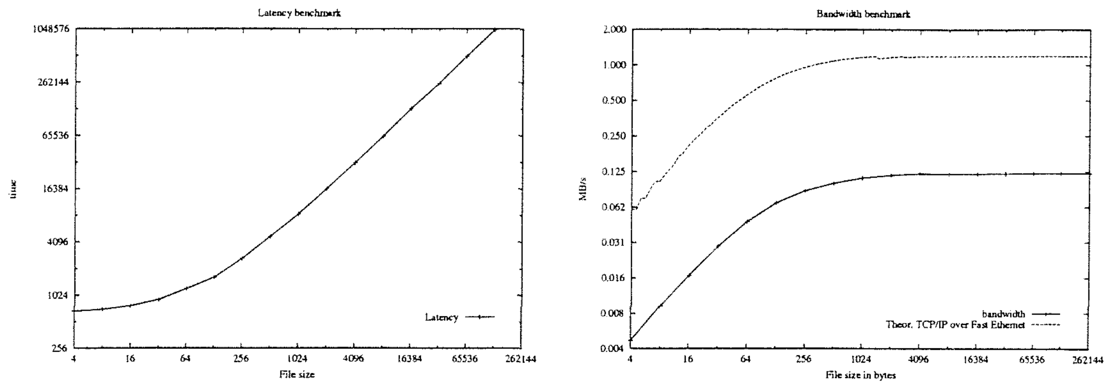

Figure 7.4: Latency and bandwidth of the NT cluster

7.2.3 The NT cluster

The NT cluster is composed of 8 Alpha based DEC workstations and uses a dedicated interconnection network with a 100Mbit/s Ethernet switch. According to Table 7.2 this cluster has the most powerful processors/nodes among the clusters available in our tests. Several programming environment differences with the rest of the systems tested, together with access restrictions prevented network-level performance tests been run on this cluster. On the other hand, the communication library tests were proved to be portable enough on this platform to run without difficulties. The MPI version used in these measurements was rather an early experimental version based on the MPICH one. The Abstract Device Interface (ADI) makes use of the TCP/IP protocol stack provided by NT. The dedicated interconnection network of the cluster ensures "smooth" undisturbed results.

Communication library performance test. As we can see from Figure 7.4 results for latency and bandwidth on the NT cluster are not impressive, the main reason for this is the premature experimental version of that MPI implementation. An unnecessarily large number of context switches seems to degrade performance significantly. The system was unable to use the communication channel efficiently even for large messages and the latency is high even for small messages. Further investigation of the TCP/IP implementation performance is necessary because the overall performance of this cluster compared to its hardware capabilities is especially poor.

Configuration of the NT cluster	min Lat.	max BW	$n_{1/2}$
MPI over TCP/IP	673 μs	120 Kbyte/s	100 byte

7.2.4 The SPARC cluster

The SPARC cluster uses an open, non-dedicated, 10Mbit/s Ethernet subnet running Solaris (SunOS 5.5.1) with a communication library based on MPICH 1.0.12. Cluster nodes are either SPARC-4 or UltraSPARC workstations (see Table 7.2). The following table shows the low-level latency and bandwidth tests results for this cluster.

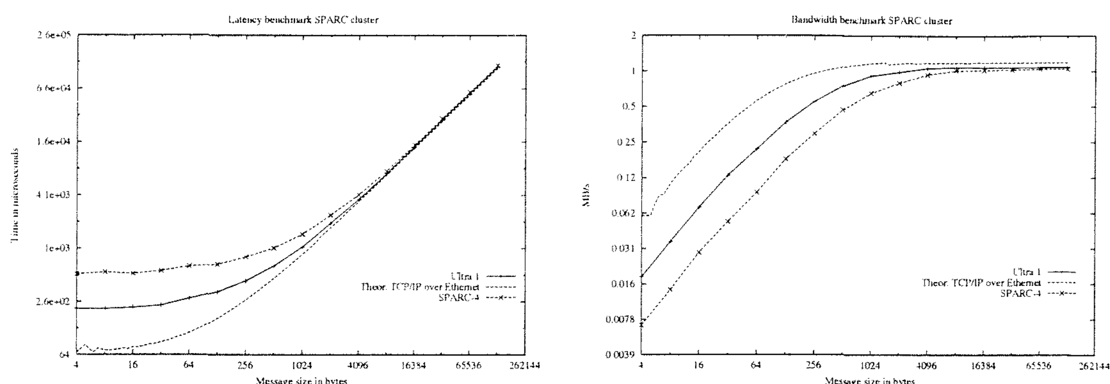

Figure 7.5: Latency and bandwidth on SPARC workstation clusters

Network Performance Tests. The network-level latency for a pair of Ultra1 nodes is 212 μs and 526 μs for a pair of SPARC-4 nodes. Bandwidth features for both systems are similar above 1 Mbyte/s. However, the half-bandwidth performance point provides some interesting results, for the Ultra1 which is faster at 256 bytes while for the SPARC-4 the half-performance point is delayed until a message size of 600 bytes. This is because Ultra1 nodes are computationally more powerful than SPARC-4 nodes thus start up time is considerable shorter and these nodes can therefore make more efficient use of the network channel when handling short messages. The approximate model equation 6.10 yields 160 μs start-up latency and 1.05 Mbyte/s asymptotic bandwidth.

Network performance of SPARC clusters	min Lat.	max BW	$n_{1/2}$
Berkeley Sockets TCP/IP over Ethernet Ultra1 workstation	212 μs	1.082 Mbyte/s	256 byte
$a + b * x$ approximation	160 μs	1.052 Mbyte/s	170 byte
Berkeley Sockets TCP/IP over Ethernet SPARC-4 workstation	526 μs	1.047 Mbyte/s	600 byte

Communication library performance test. Plots in Figure 7.6 illustrate communication-level latency and bandwidth-performance tests run on a pair of SPARC-4 and a pair of Ultra-SPARC nodes respectively. Once again here the communication library latency figure is 2 to 2.5 times longer for small messages as was expected due to MPI communication library overhead but bandwidth and the half-bandwidth performance point $n_{1/2}$ are not degraded significantly.

Configuration of a Sun cluster	min Lat.	max BW	$n_{1/2}$
ULTRA SPARC	660 μs	1.03 Mbyte/s	280 byte
SPARC-4	1.37 ms	1.01 Mbyte/s	750 byte
ULTRA SPARC (loopback)	1.5 ms	840 Kbyte/s	700 byte

Although theoretically this cluster has some of the least powerful nodes of our tested clusters (SPARC-4), it gives some of the best performance results among the 10 Mbit/s Ethernet configured clusters. Both SPARC-4 and ULTRA SPARC use the network channel very effi-

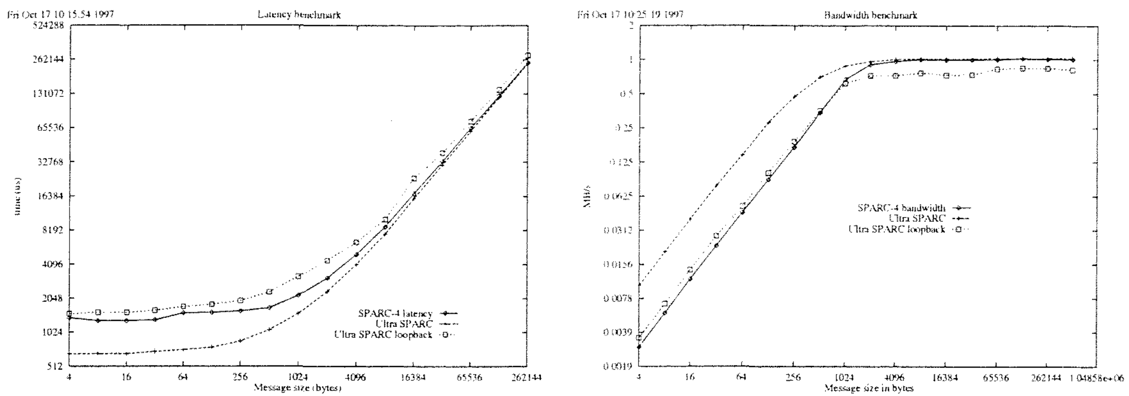

Figure 7.6: Latency and bandwidth of the Solaris cluster

ciently and communication bottleneck is therefore close to the Ethernet channel limits. The non-dedicated intercommunication network had almost no effect on the results. Keeping the test configuration essentially the same, (the same software and intercommunication network) we can switch between SPARC-4 and ULTRA SPARC nodes. As we expected the more powerful node improves latency and bandwidth performance figures especially for short-size messages.

7.2.5 The SGI Cluster

The SGI cluster is composed of Silicon Graphics O2 workstations interconnected with a switched Fast Ethernet network. The communication library used throughout our tests was the MPICH 1.1.1. Network level latency characteristics for this cluster are regarded rather high at 368 μs with the half-bandwidth performance point $n_{1/2}$ is at 1.5 Kbyte, while bandwidth for large messages approaches the theoretical absolute just above 12 Mbyte/s. The approximate latency and bandwidth (from equation 6.10) for this cluster are 480 μs and 12.4 Mbyte/s respectively. Figure 7.8 illustrates communication library level test results for this cluster. The MPI overhead is around 546 μs and the half performance point $n_{1/2}$ is affected very little. Large message bandwidth is degraded to 8.9 Mbyte/s. Results from the SGI cluster improve the overall latency performance but the half bandwidth performance point is delayed to about 1.5 Kbyte. This is due to the bandwidth improvement of the Fast Ethernet channel while the rest of the communication mechanism is not likely to have any improvement over a 10 Mbit/s Ethernet network.

Network performance of O2 cluster	min Lat.	max BW	$n_{1/2}$
Berkeley Sockets TCP/IP over Fast Ethernet	368 μs	12.06 Mbyte/s	1.5 Kbyte
$a + b * x$ approximation	480 μs	12.40 Mbyte/s	1.5 Kbyte

Communication performance of O2 cluster	min Lat.	max BW	$n_{1/2}$
MPI over TCP/IP over Fast Ethernet	546 μs	8.9 Mbyte/s	~1.5 Kbyte

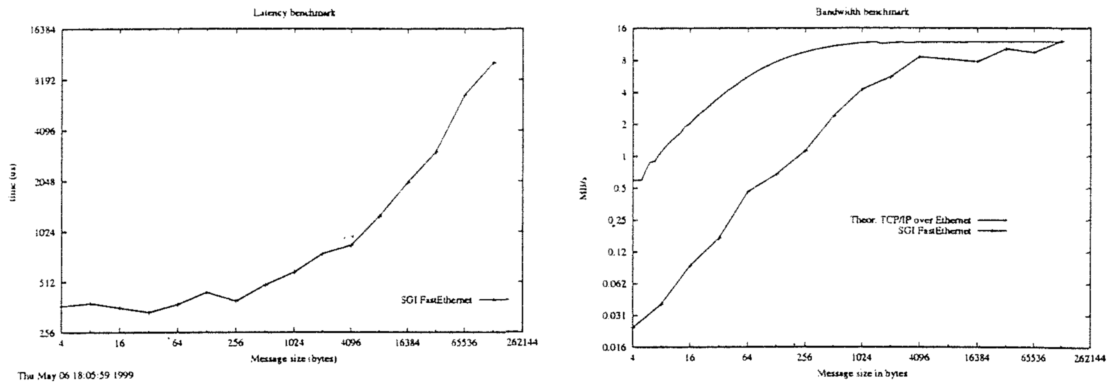

Figure 7.7: Latency and bandwidth on O2 cluster

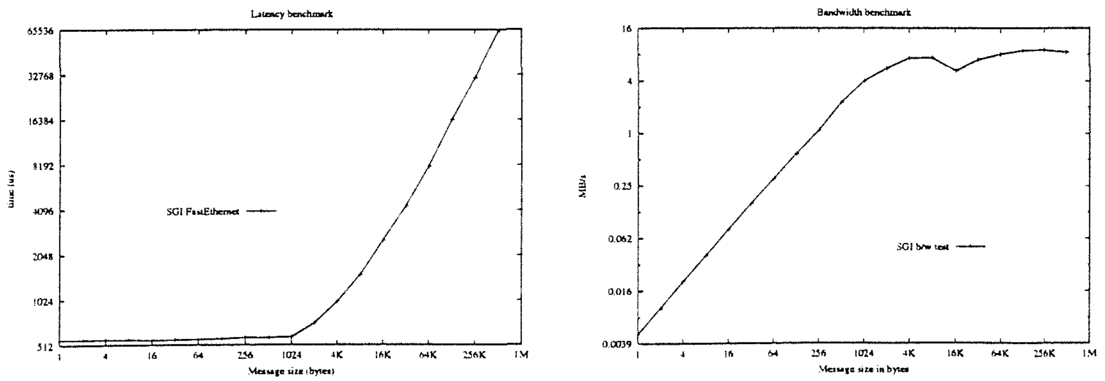

Figure 7.8: Communication level latency and bandwidth on O2 cluster

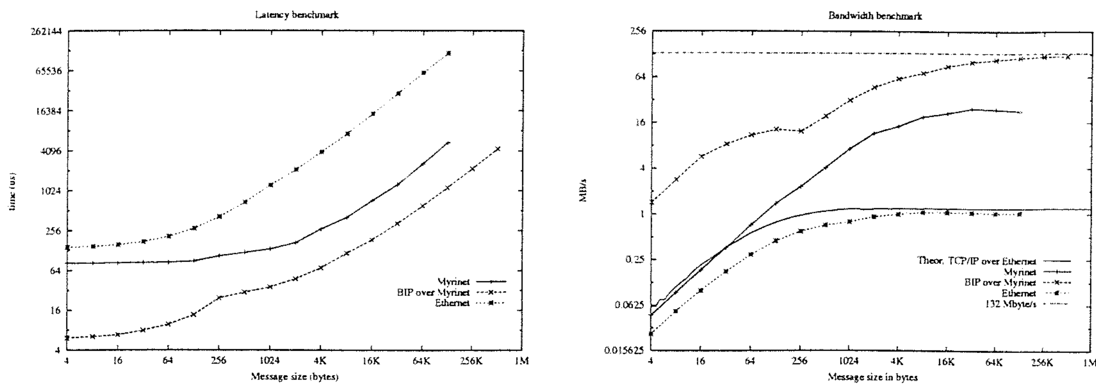

Figure 7.9: Latency and bandwidth on the BIP cluster for bare network protocols

7.2.6 The BIP Cluster

In previous sections we saw the impact in the performance of a general purpose communication protocols such as TCP/IP in various clusters of workstations. A mismatch between hardware and software evolution could essentially negate any advantages provided by high-performance hardware.

The BIP cluster is built around Pentium-Pro machines running Linux 2.0.1, with an interconnection based on a Myrinet network. This network interface has been designed to deliver to the application layer the maximum performance achievable by the hardware [183]. An attempt to compromise between new hardware features and software compatibility such as BIP could provide some very interesting results [183, 22]. The BIP cluster of workstations, presented in section 2.7, provides a very flexible communication APIs because each node has two NIC (Ethernet and Myrinet) and two flexible network protocol stacks TCP/IP or BIP (see Figure 2.11). Benchmarking tests have been run using an API configured either as typical TCP/IP stack over an Ethernet channel or TCP/IP over Myrinet or directly on top of the BIP network protocol over Myrinet. In the first case the TCP/IP protocol stack on top of the Ethernet network was selected.

Network performance of the BIP cluster	min Lat.	max BW	$n_{1/2}$
Berkeley Sockets TCP/IP over Ethernet	144 μs	1.04 Mbyte/s	150 byte
Berkeley Sockets TCP/IP over Myrinet	83 μs	24.6 Mbyte/s	2.5 Kbyte
BIP API over Myrinet	6 μs	121 Mbyte/s	4 Kbyte

Configuration of BIP cluster	min Lat.	max BW	$n_{1/2}$
MPI over TCP/IP and Ethernet	280	1 Mbyte/s	300 byte
MPI over TCP/IP and Myrinet	171	17.9 Mbyte/s	1.5Kbyte
MPI over BIP and Myrinet	11	114 Mbyte/s	8Kbyte

The performance of the TCP/IP Ethernet configuration gives latency for zero-size message length 290 μs and a bandwidth close to 1 Mbyte/s for messages larger than 1 Kbyte. Changing the physical network from Ethernet to Myrinet, via the same TCP/IP protocol stack (TCP/IP

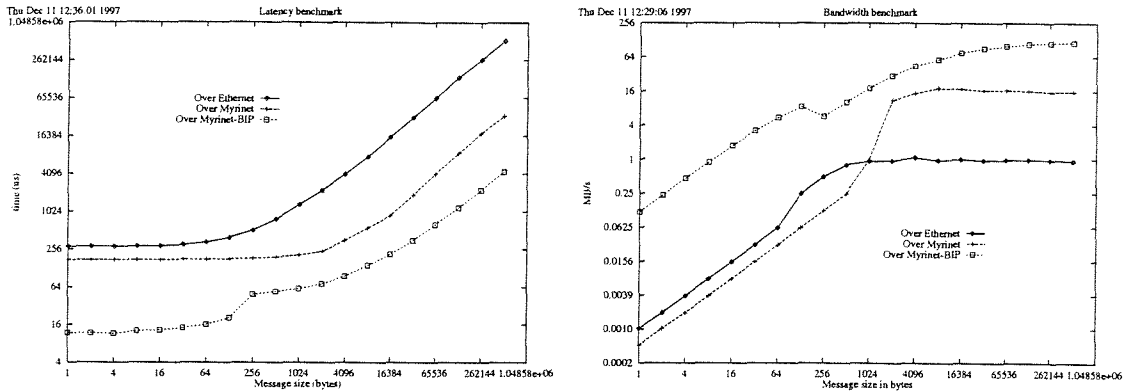

Figure 7.10: Latency and bandwidth on a Myrinet cluster with page alignment

over Myrinet) provides a significant performance improvement. Zero-size message latency is 171 μs and the bandwidth reaches 18 Mbyte/s, with an $n_{1/2}$ figure below 1.5 Kbyte. In the final API configuration (BIP over Myrinet) the application (benchmark) interacts directly with the network interface through the BIP stack. The performance improvement in this case is impressive, testing the network board performance to the network design limits. Zero length message latency is 11 μs and the bandwidth exceeds 114 Mbyte/s with half-bandwidth performance point $n_{1/2}$ at 8 Kbyte.

Figure 7.10 shows the latency and bandwidth graphs over those protocol stack configurations, while Figure 7.9 and Table 7.3 illustrates the peer-to-peer performance measurements of the underlying network protocols, without the use of communication libraries i.e. MPI. Analysing results from the Ethernet configuration we observe that latency drops down to 144μs but there is not any noticeable improvement in bandwidth because the current bottleneck is still in the low bandwidth of the Ethernet channel. Using BSD sockets over the Myrinet configuration we notice a further improvement in latency (84 μs) and bandwidth (23 Mbyte/s). Finally the BIP configuration as a user-space API gives the best results with latency of 6 μs and bandwidth above 120 Mbyte/s. Results from the last test show a noticeable discontinuity at message sizes of 256 bytes, on the BIP curve, that reveals the point at which different semantics, between short and long messages transmission modes, take place for that protocol (the PIO/DMA switch-over specified by *BIPSMALLSIZE* [182]). From the above latency and bandwidth graphs the impact of Ethernet and TCP/IP protocols on system performance is clear. Figures 7.10, 7.9 and Table 7.3 show the impact of the MPI communication library and the network protocols on the latency and the bandwidth. The Ethernet channel imposes a bandwidth barrier at 1 Mbyte/s while the TCP/IP protocol stack imposes a bandwidth barrier around 23 Mbyte/s. In terms of latency the time required by the OS to control the network interface (e.g. the Ethernet board) is apparent as is the time required to proceed through the network protocols according to equation 6.9.

The impact of the communication library (MPI) is related to the performance of the network protocols. Therefore, as Figures 7.10 and 7.9 illustrate, the cost of the communication library for a slow communication channel is very high (start time overhead $t_s \ll t_w n$ per byte cost), but as we move to faster communication channels this cost becomes higher and then rep-

Table 7.3: Ping-pong test results on various communication libraries

Configuration	min Latency	max BW	$n_{1/2}$
TCP/IP sockets vs. MPI over TCP/IP	144/280μs	1.06/1.0 Mbyte/s	0.3/0.3 Kbyte
a+b*x approximation	137/- μs	1.09/- Mbyte/s	
TCP/BIP sockets vs. MPI over TCP/BIP	84/171μs	23/17.9Mbyte/s	1.5/1.5 Kbyte
a+b*x approximation	74/- μs	24.7/- Mbyte/s	
BIP sockets vs. MPI over BIP	6/11 μs	121/114 Mbyte/s	3/8 Kbyte
a+b*x approximation	26/- μs	120.1/- Mbyte/s	

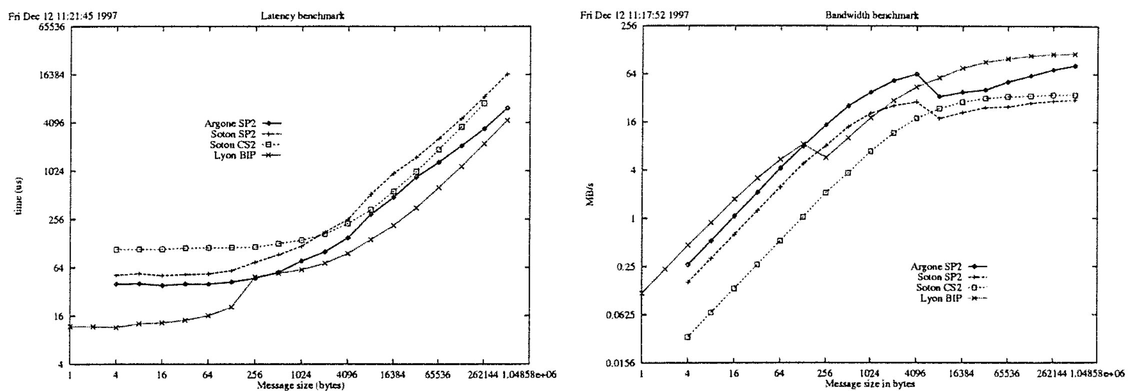

Figure 7.11: Comparing latency and bandwidth between a Myrinet cluster and MPPs

resents a significant fraction of the communication overhead e.g. TCP/IP/Myrinet ($t_s \sim t_w n$). Interfacing the MPI communication library directly onto the BIP without involving the OS can minimise that overhead considerably and take advantage of the hardware capabilities. The approximation modeling equation 6.10 provides results close to the measured ones as Table 7.3 shows.

7.2.7 Analysis of Peer-to-Peer Test Results

Traditional network protocols used with high-speed network technologies on NOWs often impose a communication bottleneck which limits performance. The 10 Mbit/s Ethernet interconnection in NOWs imposes a slow-speed barrier (1Mbyte/s) in network operations. Throughout our tests the Ethernet channel was saturated. Using a faster communication channel (such as a Myrinet network) we would move the bottleneck to the higher communication protocols such as TCP/IP. The communication bandwidth is now around 18 Mbyte/s for user applications but the network usage remains very low (13%). The use of a network switch increases the

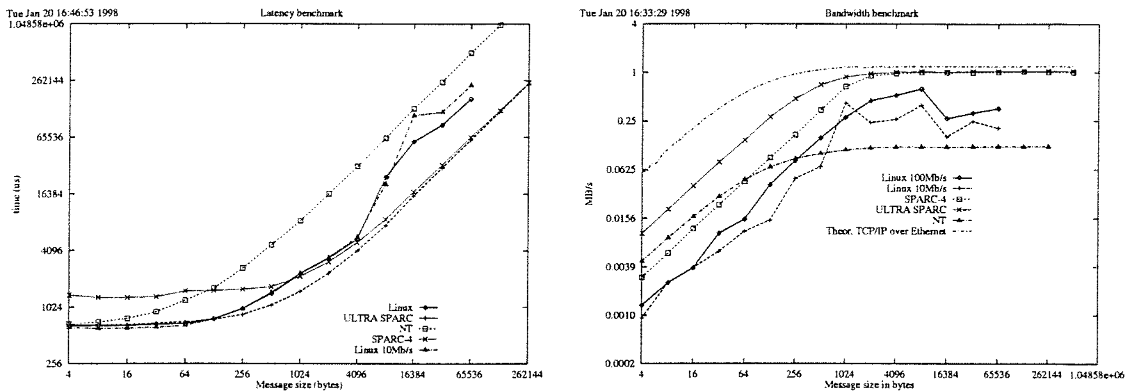

Figure 7.12: Comparing Latency and bandwidth of our clusters

aggregated bandwidth and improves performance by eliminating collisions and retransmissions. A faster implementation of TCP/IP has the potential to move the communication bottleneck higher but at the same time other OS restrictions are likely to limit performance.

An aggressive zero-copy user-space network protocol (such as BIP) can eliminate protocol bottlenecks, exploit more fully the network bandwidth (>83% at the application level) and in addition reduce latency for short messages as well. The very low end-to-end latency of 6 μs achieved by the BIP cluster is an important result for parallel applications which use small-size messages to coordinate program execution and for this size of message, latency overhead dominates the transmission time. As mentioned earlier for such small messages the communication cost cannot be hidden by any programming model or programming techniques. Other important features of the cluster are the high bandwidth of the network channel (121 Mbyte/s usable bandwidth) and the use of the Myrinet switch which reduces potential contention problems.

Low-level results from the BIP cluster show that the BIP interface exploits the network interface raw performance extremely well and delivers it to the application level. The MPI-BIP performance is directly comparable with MPP systems such as the SP2, T3D, and the CS2. As we can see from Figure 7.11 the BIP cluster has better latency features within the whole range of the measurement compared with the MPP systems of Fig 7.11. In bandwidth terms for short messages up to 256 bytes the BIP configuration outperforms all the other MPPs. Then for messages up to 4Kbyte, which is the breakpoint of the SP2 at Argonne, the SP2 has a better bandwidth, but then the BIP cluster performance is better again.

7.3 Low-level Collective Call Tests Results

This section will present results from collective call tests, which were discussed in section 6.5.4 of the previous chapter. An important feature of the MPI collective calls is that they are built on top of primitive peer-to-peer calls based usually on a tree-like algorithm. Barrier synchronisation, broadcast, reduce and all-to-all tests were run on clusters that have more than 8 nodes available unless the cluster had a specific characteristic e.g. the BIP cluster was tested in some tests with 6 nodes available only. In practice collective call testing proved more difficult than the peer-to-peer testing because of the higher requirements in both resources and time duration. For

Table 7.4: Latency and Bandwidth results

Configuration	Cluster	H/.W	min Lat.	max BW	$n_{1/2}$
MPI &TCP/IP	NT/Alpha	FastEth.	673 μs	120 KB/s	100
MPI & TCP/IP	Linux/P.Pro	Ethernet	587 μs	265 KB/s	1.5 K
MPI &TCP/IP	Linux/P.Pro	FastEth.	637 μs	652 KB/s	1.5 K
MPI & TCP/IP	SPARC-4	Ethernet	660 μs	1.03 MB/s	280
MPI & TCP/IP	ULTRA	Ethernet	1.37 ms	1.01 MB/s	750
MPI & TCP/IP	Linux/P.Pro	Ethernet	280 μs	1 MB/s	300
MPI & TCP/BIP	Linux/P.Pro	Myrinet	171 μs	17.9 MB/s	1.5 K
MPI & BIP	Linux/P.Pro	Myrinet	11 μs	114 MB/s	8 K

example it was difficult to book or pre-arrange clusters with a certain number of homogeneous performance nodes for a certain amount of time. Another practical problem we have to address was the large amount of information results for each of the test results.

7.3.1 Collective Call Tests on the SPARC Cluster

The SPARC-4 workstation cluster used for collective call tests is a typical cluster configuration with a shared bus network topology. The barrier synchronisation test measures the time required for up to 10 SPARC-4 nodes to synchronise. Figure 7.13 illustrates the results of this benchmark along with the fitting approximation of equation 6.15. These results point out a number of facts: the graph is not linear as could be expected for a shared bus network. The reason for this is the non-linear number of individual peer-to-peer calls required by the MPICH synchronisation algorithm (see Fig. 6.3 and equation 6.14). Another fact is that individual peer-to-peer calls do not include any payload and a small number of nodes are unlikely to cause congestion problems on a shared bus network channel. A larger number of nodes however might be expected to encounter congestion problems. According to equation 6.15 the approximated logarithmic step of that model is estimated at 1.34 ms for this cluster which is very similar to the single peer-to-peer latency time of 1.37 ms measured previously for this cluster. Consequently according to equation 6.15 a two node synchronisation latency is around 2 ms and involves only one "duplex" *Send/Recv* peer-to-peer call and so forth.

A broadcast collective operation is a data movement collective call with performance affected by both the number of nodes participating and the message size. The broadcast algorithm is based on a binary tree algorithm (described in section 6.5.4) and the actual number of messages which nodes have to exchange is *p-1* which cannot be overlapped in the shared bus technology network cluster. Figure 7.14 illustrates the results of the broadcast call on various number of nodes and different message sizes. In all graphs, time is proportional to the number of nodes or the message size. The initial broadcast latency (for very small messages) is 1.35 ms which is better than any barrier synchronisation call latency measured because the broadcast algorithm is considerably less complicated than the synchronisation one. Hence the broadcast

Figure 7.13: Barrier Synchronisation test for the SPARC cluster

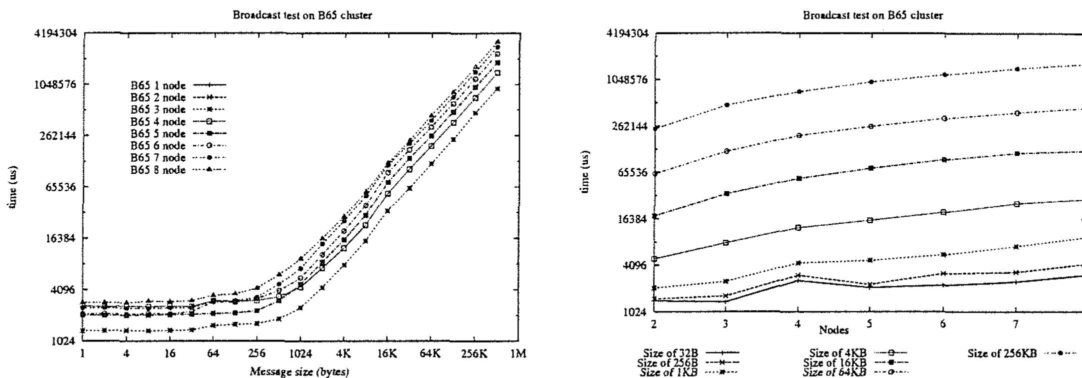

Figure 7.14: Broadcast test on a SPARC cluster

algorithm is expected to have less overhead than the barrier call. For larger messages the time required for transmission through the narrow channel bandwidth for each pair of nodes increases and becomes the dominant part of the call. The "effective bandwidth" of the broadcast, which is the aggregated message size sent to all nodes over the period of the call improves as the number of nodes increases for small to medium-size messages and for large-size message is approaching 1.09 Mbyte/s. This is because in both cases startup time is improved with warm cache effects or long message transmissions. The point at which the MPI implementation changes the send/receive protocol policy at 16 Kbyte is just distinguishable.

The result of the reduce operation is shown in Fig. 7.15. The overhead of the reduce call is very similar to the broadcast call, the essential difference being the extra overhead of the operation involved in the reduce call. The global computation of the logical OR operation used for the reduce operation benchmark is completed within a single clock cycle. The first plot of these results shows the cost of a single node reduction operation which is purely computational as there is no communication part involved. Here once more the latency of small messages (1.4 *ms*) is no better than the broadcast call but is still shorter than the synchronisation call.

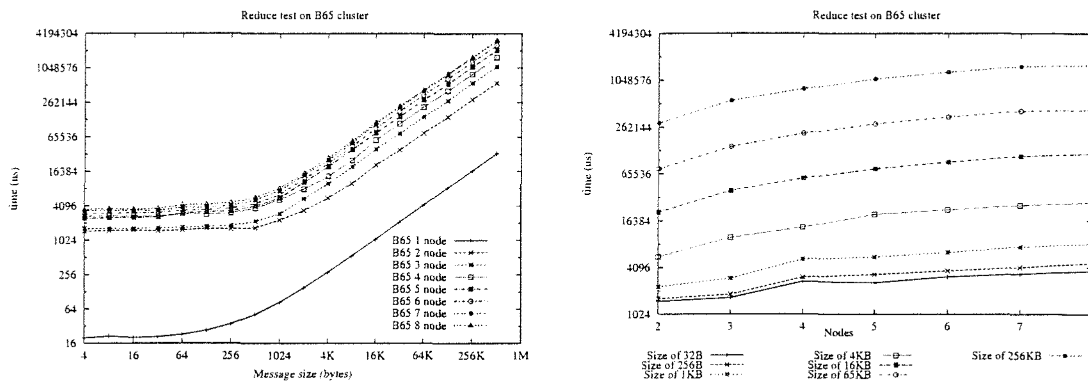

Figure 7.15: Reduce test on a SPARC cluster

Broadcast and reduce operation results are very similar because their algorithms have similar complexity. Figure 7.14b and 7.15b show characteristically how an increase in the number of nodes increases the broadcast and the reduce operation latency linearly on a shared bus network. The cost of the combined operation does not affect the results significantly because it represents a very small fraction of the call. A model approximation for broadcast and reduce call based on equation 6.16 and 6.17 yields the following equations for this cluster:

$$1764 + \lceil \log_2 p \rceil \cdot 0.91 \cdot n$$

and

$$2073 + \lceil \log_2 p \rceil \cdot 1.02 \cdot n$$

where 1764 and 2073 are the startup time cost in microseconds and 0.9095 and 1.0225 is the cost per byte respectively.

7.3.2 Collective Call Tests on the SGI Cluster

The SGI cluster represents a cluster of fast workstations with a Fast Ethernet switched interconnection network. This cluster has two main characteristics node availability, nodes are connected via a number of cascaded network switches, and an opaque allocation node scheduler which does not provide direct control over the selection of the nodes.

The barrier synchronisation test run on a set of 18 nodes. The initial synchronisation latency for two nodes is 0.65 *ms* and the cost for each extra step of the algorithm is approximated by equation 6.15 at 0.6 *ms* which is very close to the 0.55 *ms* latency measured for a single peer-to-peer call in section 7.2.5. Indeed the approximation model of equation 6.15 in practice was proved correct. Comparing the results with the synchronisation test on the SPARC cluster there are several conclusions. Fast nodes and a fast network channel such as those of the SGI cluster provide significant latency improvement. Initialisation overhead is reduced and the synchronisation call iteration steps are evenly distributed (around 0.65 ms) which can simplify the modeling approximation towards a logarithmic step function:

$$t_{bar}(p) = t_s \cdot \lceil \log_2 p \rceil$$

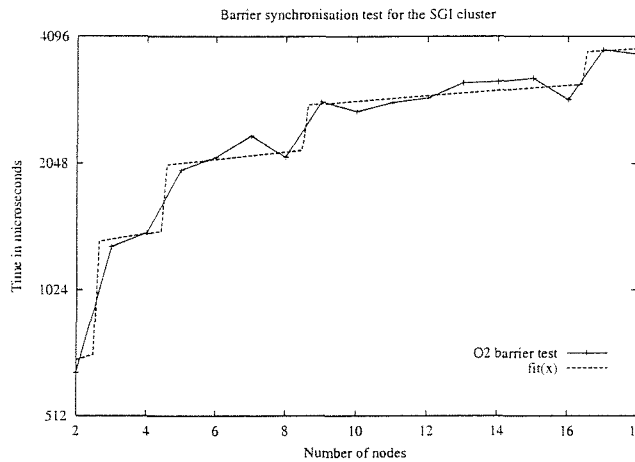

Figure 7.16: Barrier Synchronisation test for the SGI cluster

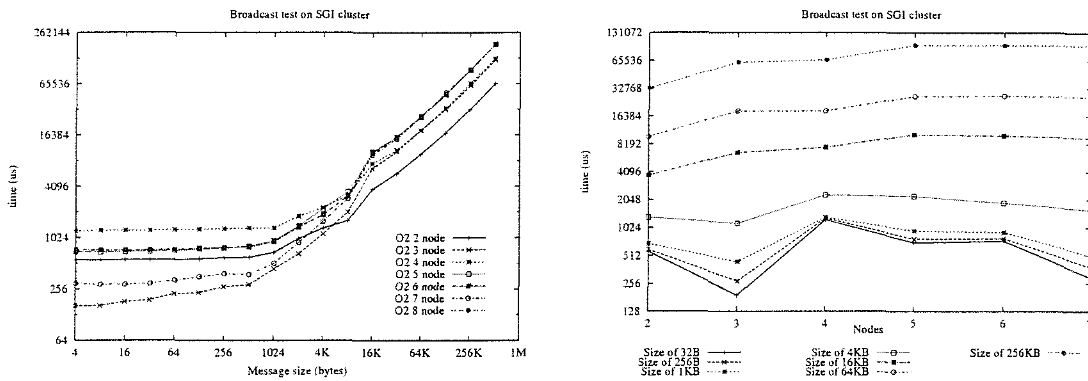

Figure 7.17: Broadcast test on a SGI cluster

Figures 7.17 and 7.18 show results of the broadcast and reduce operation test for the SGI cluster. The left hand side graphs show latency results as a function of the message size for different number of nodes. It is interested to notice that plots tend to group in three distinct areas representing different logarithmic iterations of the broadcast algorithm. The same result is illustrated on the right hand side plots where latency is shown as a function of the number of nodes for different message sizes. These logarithmic iterations are presented as distinct flat areas on the graph. Both broadcast and reduce operation tests have considerable reduced latency figures over the SPARC cluster results. As the communication part of the calls becomes smaller, due to improved node hardware, the cost of the reduce call slightly increases compared with the cost of a broadcast call especially for large messages. The effective bandwidth for broadcast call large messages is 8 Mbyte/s, for small-to medium-size messages and improves as the number of nodes is increasing (because of a caching effect).

Figure 7.19 illustrates the results of an all-to-all call on the SGI cluster. The number of peer-to-peer calls required for the implementation of this call is exponential to the number of nodes participating. Hence as the number of the message size or the number of nodes is

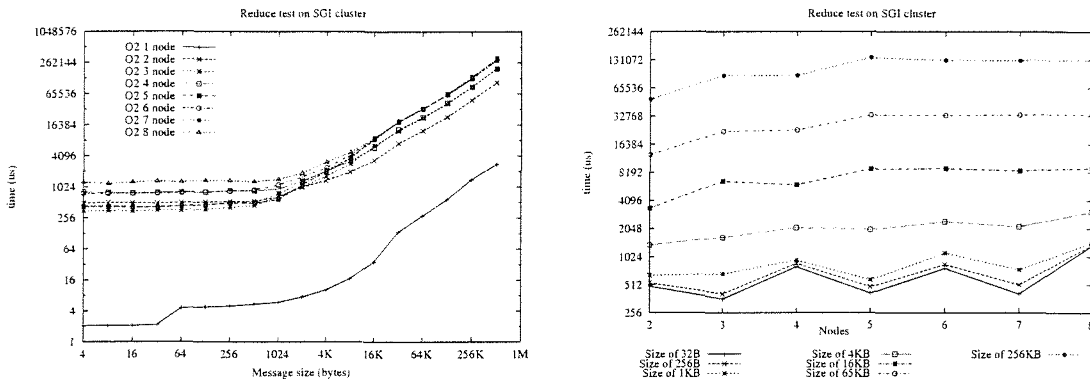

Figure 7.18: Reduce test on an SGI cluster

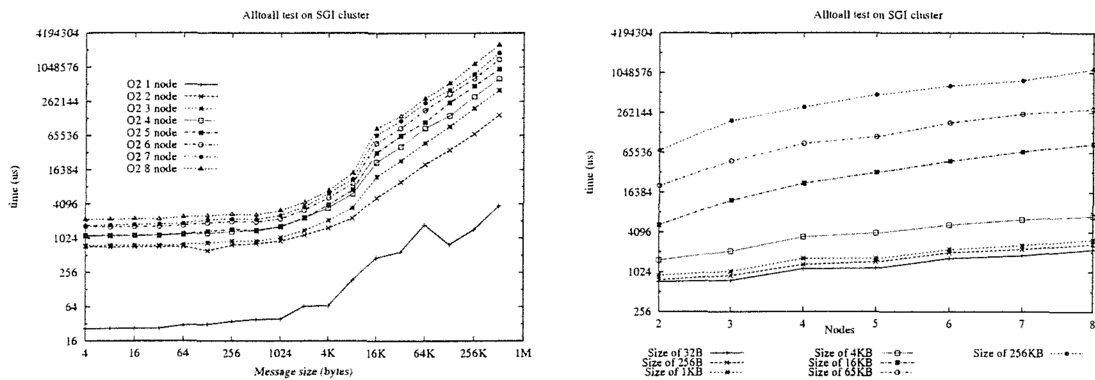

Figure 7.19: All-to-all test on an SGI cluster

increasing the latency of the call increases exponentially. An important characteristic of the MPI all-to-all call implementation is that each node has also to exchange a peer-to-peer call with itself, the left hand side figure shows the cost of this call when we run the test on a single node.

Throughout the tests on the SGI cluster it has been shown that the use of a switched interconnection network increases the overall efficiency of the cluster because the existence of multiple communication paths avoids potential congestion problems experienced on a single shared bus network previously. The bandwidth breakpoints at 16 Kbyte message size in Figures 7.17 and 7.19 are due to the MPI implementation transmission policy for long and short messages.

7.3.3 Collective Call Tests on the BIP Cluster

Collective call benchmarks run on the BIP cluster run on the same communication API configurations presented earlier for the peer-to-peer tests. The barrier test measures the time required for 2, 4 and 6 nodes to synchronise. The MPI implementation of the call is based on a butterfly mechanism, using a tree-like algorithm with *logp* steps (where *p* is the number of processors). Figure 7.20 illustrates the barrier synchronisation latency as a function of the number of nodes. From that figure we can see the discrete steps of the logarithmic implementation of the call.

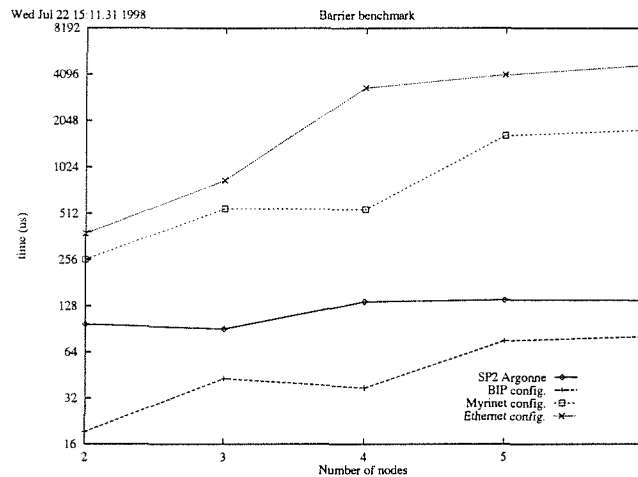

Figure 7.20: Barrier Synchronisation tests for the BIP cluster and MPPs

For the BIP configuration this step is estimated around 19 μs (equation model 6.15 fit approximation) while for the SP2 system similar tests show a start-up time close to 46 μs which again gives for the BIP cluster better synchronisation figures than the SP2 system.

Similar discrete steps are visible on with the TCP/IP over Myrinet configuration curve as well. The start-up time of this configuration is considerably longer (>250 μs). The existence of the Myrinet switch in both the BIP and the TCP/IP over Myrinet configurations allows concurrent use of multiple paths and therefore efficient implementation of the call. On a bus channel, (e.g. the TCP/IP over Ethernet configuration) latency start-up time is measured around 400 μs and becomes larger as the number of nodes increases. It is important to note that a barrier synchronisation call theoretically does not depend on the channel bandwidth, although network latency often has a significant impact on the performance of an application. Communication protocols that make significant use of OS calls (such as TCP/IP) are penaltised with high overheads leading to long latency times.

The Broadcast Operation Test. Each step takes $t_s + t_w m$ for a single message transfer, although in practice these steps are not very discrete for very small messages where noise and the start-up time t_s dominates the latency of the broadcast call.

For a bus-connected cluster (Ethernet configuration) the broadcast algorithm is implemented in a sequential way. Frame collisions and retransmissions can affect bus performance as the number of nodes ready to transmit and the size of the transmitting frame increases. Table 7.5 illustrates the results of the broadcast test over the 10 Mbit/s Ethernet channel. The bottleneck in the network layer is caused by the Ethernet channel saturating. Therefore, characteristics from higher-level network protocols are not seeing and the graphs are linear (the 16Kbyte breakpoint is clear).

Moving to a faster network protocol (Myrinet configuration) the bottleneck is focused now on the implementation of the TCP protocol. For broadcasting messages up to 2KB the start-up time t_s of the TCP protocol dominates the broadcast time, so the graph up to that message size is almost flat (200-350 μs). Another breakpoint is visible for message sizes close to 64KB,

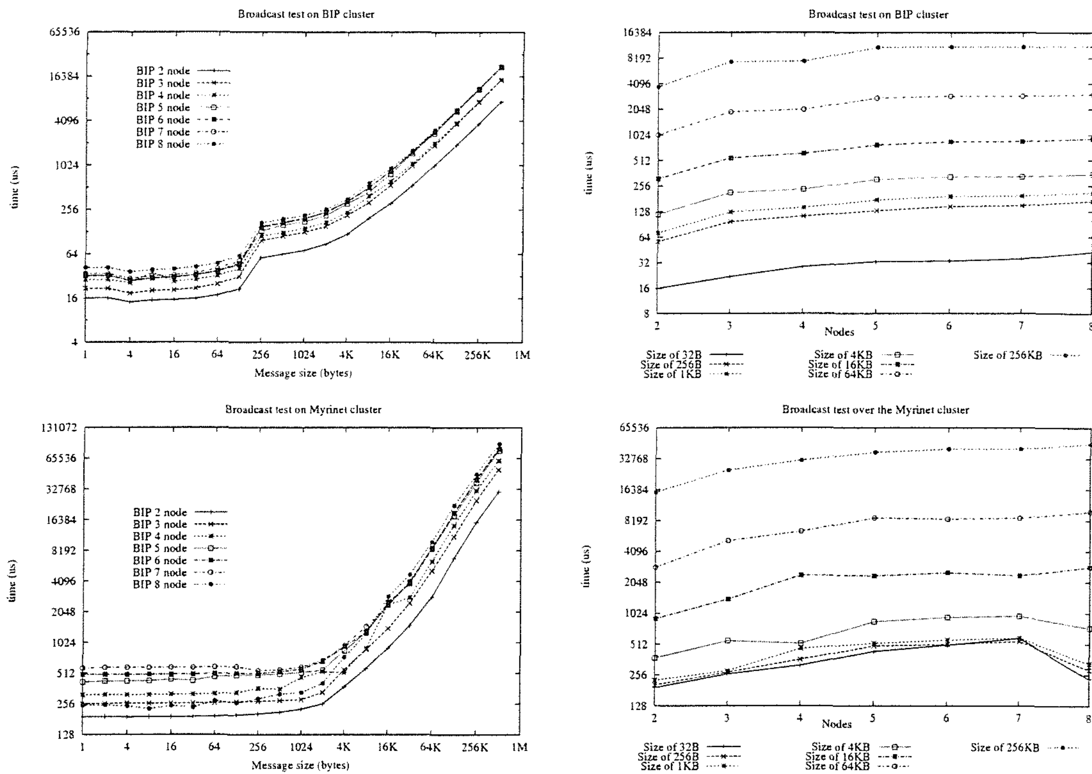

Figure 7.21: BIP cluster broadcast tests

which is the maximum size for an IP datagram.

The results of broadcasting, using the BIP configuration, are similar to the previous ping-pong ones. For messages close to 256 bytes there is a breakpoint because of the change of the transmission protocol (short/long messages). Finally the effective bandwidth stays invariant of the number of nodes and for large messages is close to 90% of send/receive peer-to-peer calls.

The last column of Table 7.5 illustrates the results of the same broadcast test on two SP2 machines. The first machine uses the native IBM MPI implementation while the second one uses the MPICH implementation. The binomial tree implementation of the broadcast call is obvious for the later graph. In comparison with the SP2, broadcasting in the BIP cluster is faster for short message (<256 bytes), then for message sizes up to 4KB the SP2 is faster, but after this point the BPI cluster is faster again.

The Reduce Operation Test. A reduce operation call as a global computation (or combine operation) collective call not only transfers data among the nodes, but nodes have to access locally the transferred data in order to calculate the partial results of the combine operation as well. For this reason the cost of a reduce operation is relatively higher, i.e. see Table 7.6 for the plot for a single node.

Reduce operation tests run successfully for a combine MPI_LOR operation in all the BIP cluster network protocol configurations. During the reduce operation test the cost of the combine operation, MPI_LOR, remains constant while the communication cost depends on the

Table 7.5: Broadcast operation test measurements (time in μs)

Size	Nodes	Ethernet	Myrinet	BIP	SP2
4	2	281	217	15	52
	4	690	314	24	72
	6	1217	334	25	100
256	2	480	227	51	59
	4	1365	402	97	90
	6	2493	371	123	124
4K	2	3.8 ms	386	96	150
	4	12.3ms	789	190	286
	6	·24 ms	911	292	360
256K	2	270 ms	19 ms	2226	3445
	4	837 ms	41 ms	4446	6857
	6	1.43 s	61 ms	8533	10.2ms

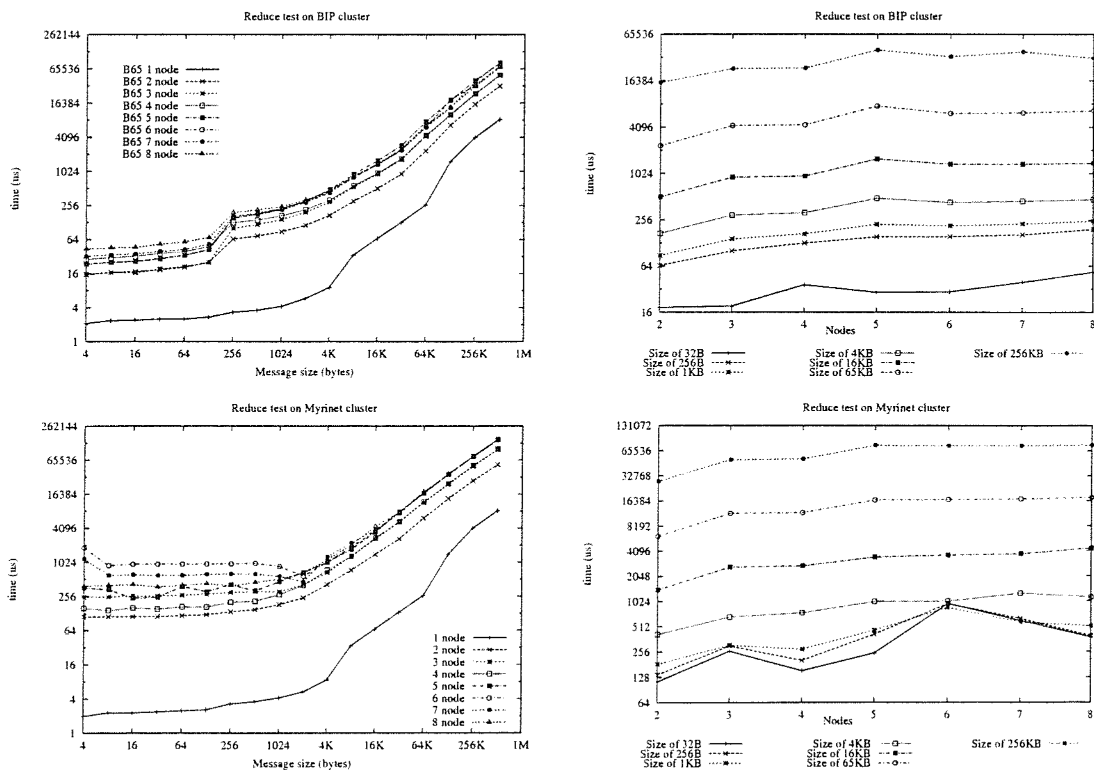

Figure 7.22: BIP cluster reduce tests

Table 7.6: Reduce operation test measurements (time in μs)

Size	Nodes	Ethernet	Myrinet	BIP	SP2
	2	227	232	16	82
4	4	736	435	20	104
	6	1561	1555	24	117
	2	432	248	61	93
256	4	1484	466	114	129
	6	3 ms	1413	142	150
	2	3807	439	16	256
4K	4	13 ms	876	299	476
	6	27 ms	1638	453	588
	2	274 ms	30 ms	13 ms	11 ms
256K	4	0.96 s	61 ms	32 ms	13 ms
	6	1.63 s	95 ms	49 ms	15 ms

protocol configuration. The reduce operation compared with the broadcast operation has an increased cost $t_{op}m$ because of the extra cost of the combine operation:

$$t_{red_p} = (t_s + t_w m + t_{op} m) \lceil log_2 p \rceil \tag{7.1}$$

(where t_{red_p} is the time required for the reduce operation on p nodes, each step takes $t_s + t_w m$ for a single message transfer plus the time required for the combine operation $t_{op}m$).

Table 7.6 illustrates the performance of the 10Mbit/s Ethernet bus, which apparently is very similar to the corresponding broadcast test graph. The reason for this is the high cost of the communication in the Ethernet bus. The cost of the combine operation remains a relatively small fraction of the communication cost and does not have a significant impact on performance i.e. $t_o \ll (t_s + t_w m)$. As the communication cost decreases the computation cost of the combine operation (which is constant) becomes a significant fraction of the reduce operation.

Comparing the results between the SP2 and the BIP cluster for the reduce test, we can see that for short-message-size reduce operations the BIP system is faster, but slower for longer messages (>8KB). The reason for this turned out to be a bug in the virtual-physical address space management of the BIP protocol which reset the address space.

7.3.4 Analysis of Collective Call Test Results

This section presents a brief analysis of the collective results presented in section 7.3 together with some general observations and comparisons. For all platforms the performance of collective calls is lower than their counterpart peer-to-peer calls. The startup time of collective calls is longer than for peer-to-peer calls, and in addition their implementation is based on algorithms with a logarithmic nature which can dominate performance when the payload is small, either on a shared bus or a switched network architecture. The barrier synchronisation test behaviour

is an example of such a call with zero payload. As the payload is increasing then the 10 Mbit/s and to a lesser extent the 100 Mbit/s Ethernet channel imposes a communication bottleneck which becomes noticeable on clusters with fast nodes. Network switches with multiple communication paths have the potential to alleviate congestion problems and thus improve the overall bandwidth performance.

Another paradox of collective call tests, on SPARC and SGI clusters, is the effective performance for small and medium size payload which increases slightly with the number or nodes. The reason for this is the cache effect on repeated iterations of the basic peer-to-peer call inside the core of the collective routine algorithm. For the SGI cluster the effective performance approaches 8 Mbyte/s (67% of the peer-to-peer call bandwidth) which does not deliver the full potential of the network channel. The BIP cluster based on a simplified communication protocol has the potential to deliver performance at higher levels. The effective bandwidth performance approaches 70 Mbyte/s (more than 90% of the peer-to-peer call bandwidth which was <80 Mbyte/s for that 8-node BIP cluster).

7.4 Kernel-level Tests

This section presents kernel-level benchmark results, most of which run on the SPARC, the SGI and the BIP clusters. A practical difference among low-level and kernel-level communication benchmarks is the ratio between computation/communication parts of the workload involved in tests. While for lower level benchmarks this ratio is in favour of the communication part, for kernel-level benchmarks the ratio is in favour of the computation part. Hence clusters with powerful workstations may be expected to provide improved results. In practice, application scaling to large number of nodes in workstation clusters can be severely restricted. This happens firstly by the sequential part of the algorithm according to Amdahl's law [4] and secondly by the corresponding increase in communication and synchronisation which increases the inefficiency [114, 191] (this cost includes initialisation overheads such as buffer allocation and synchronisation). Equation 7.2 gives the modified Amdahl's law with a simplified communication overhead cost α which increases with the number of nodes linearly. Figure 7.23 illustrates various speed-up curves according to equation 7.2 for an algorithm with a 10% of the original algorithm non-paralleliseable and for different values of communication overhead cost α:

$$S(n) = \frac{T_{seq}}{T_{ser} + T_{par}/n + \alpha n} \tag{7.2}$$

The first part of kernel-level benchmarks comprises of kernel-level message passing operations, used for domain decomposition in applications with data parallelism, such as scatter, gather, etc. These operations are examined within the context of more[1] realistic conditions compared to the idealised isolated core of the low-level tests. The second part of kernel-level tests presents results from matrix operations, sorting and relaxation algorithms which run on various network of workstation platforms. The size of problem/data used in kernel-level tests is larger than its counterpart used for low-level tests.

[1]In real applications performance of the calls is expected to decrease as the burden on each node will be usually larger.

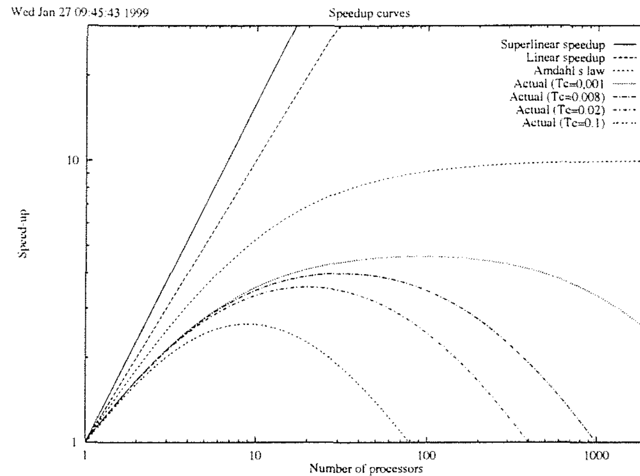

Figure 7.23: Speed-up curves, Amdahl's law and communication overhead

7.5 Kernel-level Operation Tests Results

Kernel-level operation tests can be divided into two groups, according to the total amount of data that has to be transmitted and as a function of the number of nodes. In the first case broadcast and shift operations increase linearly the amount of data need to be transmitted over the network. Performance of these calls depends on the network architecture, e.g. on a p node system the broadcast operation on a crossbar switch network can be implemented in $\log_2 p$ steps while a shift operation is implemented in one or two steps. A shared bus network on the other hand will need p sequential steps for the same data transmission.

The second group includes operations such as scatter and gather, data transmission requirements do not change as the number of nodes is increasing. In principle implementation of these operations is sequential and their performance does not depend on the underlying network architecture. Of course the underlying network speed in both of the above cases is expected to affect directly the performance characteristics for these operations.

7.5.1 The Broadcast Operation Test

This section presents the results of the kernel-level broadcast operation run on a 12-node SPARC cluster and a 6-node Pentium-Pro BIP cluster. The "root" process of the broadcast test has to distribute an array of size N X N over p processes within its communicator. The array sizes for broadcasting range N between 30 and 1080. Results below show the time for the broadcast operation which does not include initialisation phases or buffer allocation. Figure 7.24 illustrates results of the broadcast operation test measured as a function of the array size and as a function of the number of nodes.

As Fig. 7.24 illustrates, moving to a larger number of nodes while the array size is fixed the broadcast call results show a monotonically increased elapsed time. In comparison with the counterpart low-level broadcast test results presented earlier in section 7 the SPARC cluster results are neither clearly logarithmic nor linear as a function of the node number. The reason for

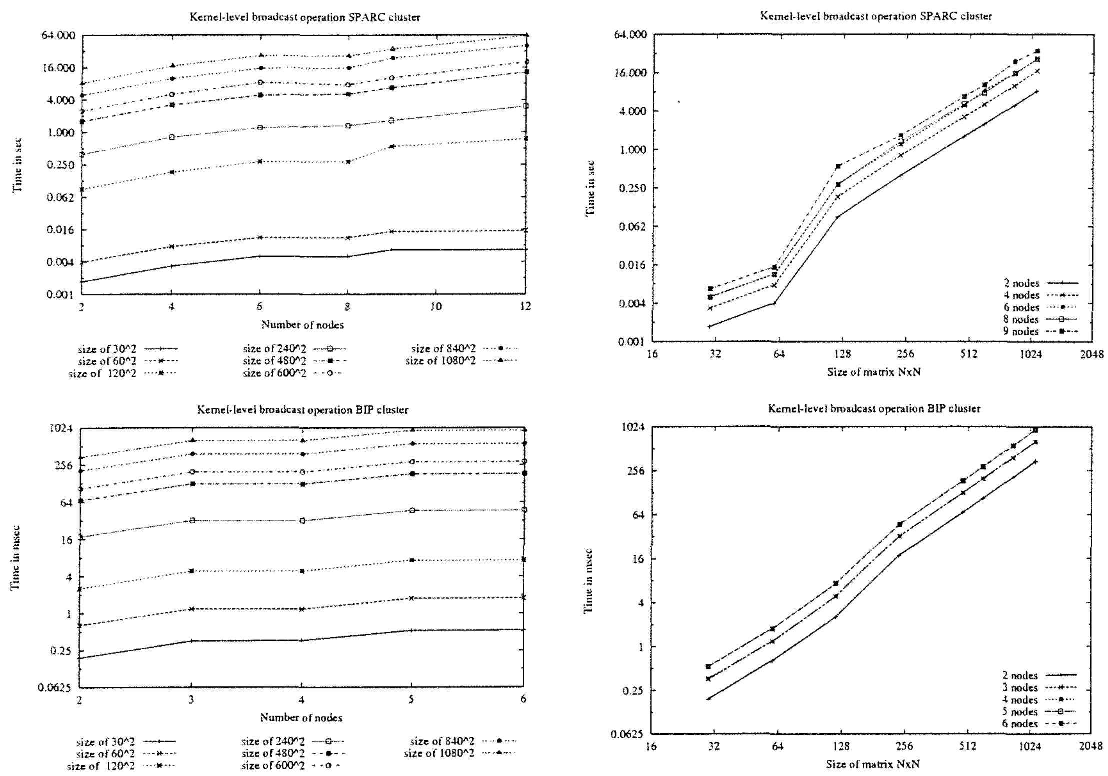

Figure 7.24: Kernel-level broadcast operation tests on the SPARC cluster (top) and the BIP cluster (bottom)

this is the linear (sequential at the best) non-deterministic nature of the Ethernet channel which overwhelms the logarithmic nature of the broadcast algorithm and represents a substantial fraction of the broadcasting call for large size arrays. On the other hand the BIP cluster is using a crossbar switch technology network which scales logarithmically according to the number of nodes (at least for the number of nodes we have tested). The bottom left graph of Fig. 7.24 shows clearly three "lines" only, the bottom line is for a 2-node test results, the middle line is the collapse of 3-node and 4-node plot results and the top line is the collapse of 5-node and 6-node plot results.

The SPARC cluster effective bandwidth is measured around 1.1 Mbyte/s for a small number of nodes but as the number of nodes increases the probability of collisions and congestion within the network channel is increasing also hence the effective bandwidth is reduced progressively to 0.8 Mbyte/s on a 12-node communicator. The size of the array also decreases slightly the effective bandwidth. The first two measurements for array size N of 30-60 are subject to noise and poor timer resolution.

The broadcast operation test on the BIP cluster gives some different results, with the effective bandwidth for array sizes of N larger than 120 around 26–29 Mbyte/s for all node configurations. This means that the network scales almost perfectly for the number of nodes we used. For an array size of $N=120$ or less effective bandwidth is almost double at 46 Mbyte/s, the reason for this being the mechanism used by the OS to allocate space and memory pages into buffers when the process needs them i.e. the Linux *demand-paging* memory management policy [190]. This is an effect which is hidden and eventually cached when the actual routine is repeated within a loop as the low-level tests do. Hence for large arrays the effective bandwidth is actually decreased by the system memory bandwidth limitation. In an absolute comparison the kernel-level broadcast operation on the BIP cluster is 26-30 times faster than the SPARC cluster. In the former one buffer allocation bottlenecks or limitations are masked by the relatively slow transmission medium rate. Results in the BIP broadcast test were verified by both timing methods, the conventional one using *MPI_ Wtime*() and the use of the register timers such as rdtsc().

7.5.2 The Scatter/Gather Operation Tests

This section examines the results of the scatter and gather operations presented previously in 6.6.3. The size of the arrays tested ranges N between N=120 and N=1680 with the node grid is ranging between 2–12 nodes (and 2–6 nodes for the BIP cluster). Figure 7.25 illustrates the results of the *MPI_Scatterv*() call run on the SPARC and BIP clusters. The left hand side plots of this figure show the elapsed time of the operation as a function of the number of nodes while plots on the right hand side of that figure show the same elapsed time as a function of the array size. The amount of data disseminating from the root node to the rest of the nodes for each array size is constant and sequential for each size of N, but independent of the number of nodes. Multiple path networks such as switched networks cannot benefit substantially from these operations.

The SPARC cluster results give an effective bandwidth around 1.1 Mbyte/s. Elapsed time

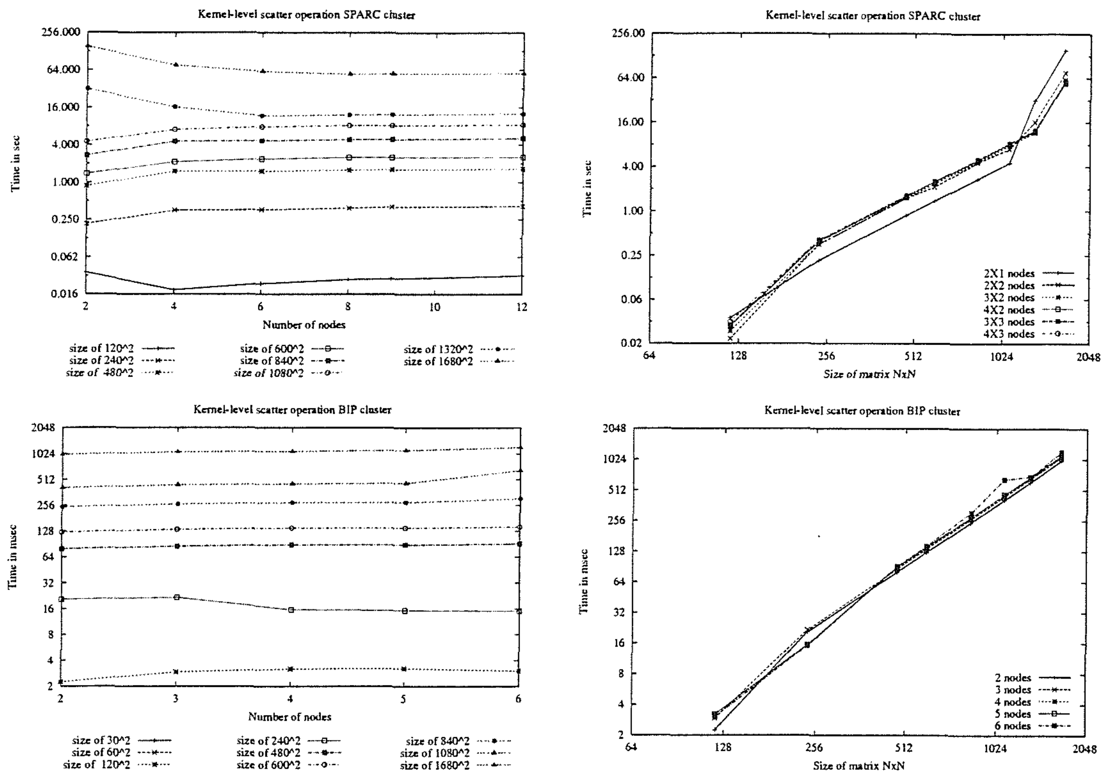

Figure 7.25: Kernel-level scatter operation tests on the SPARC cluster and the BIP cluster

steadily increases as the number of nodes becomes larger because of the extra overhead of the additional number of communication calls required and the possible congestion of the medium channel. Operating on a small number of nodes there is an anomaly in results for array sizes larger than 840 X 840 (a knee and a crossing overlap on the top left figure) which seems that elapsed time is decreasing as the number of nodes is getting larger. This is happening because the array sizes now approach the critical size of the system available memory and some memory swapping activity will be taking place.

Results from the BIP cluster scale very well with the number of nodes, mainly because the communication operation overhead is relatively low, thus results are practically independent of the node number. The effective scatter operation bandwidth of the BIP system for small size of arrays is 46–38 Mbyte/s but as the array size increases the effective bandwidth is stabilised down to 20 Mbyte/s. Once again here we suspect that the effective bandwidth performance drop is due to the system memory bandwidth limitations.

Results from the gather operation on the SPARC cluster are presented in Fig. 7.26. The performance of the gather operation is marginally slower than the scatter operation e.g. less than 5%. This means that receiving messages and writing to a buffer area is more costly than reading and sending data. Graphs of the gather operation results are very similar to the scatter operation observing also similar anomalies. An attempt to run the test over the BIP cluster failed due to a bug in the MPI implementation which did not allow different datatypes of the same data signature to be sent and received within the MPI_Gatherv() call.

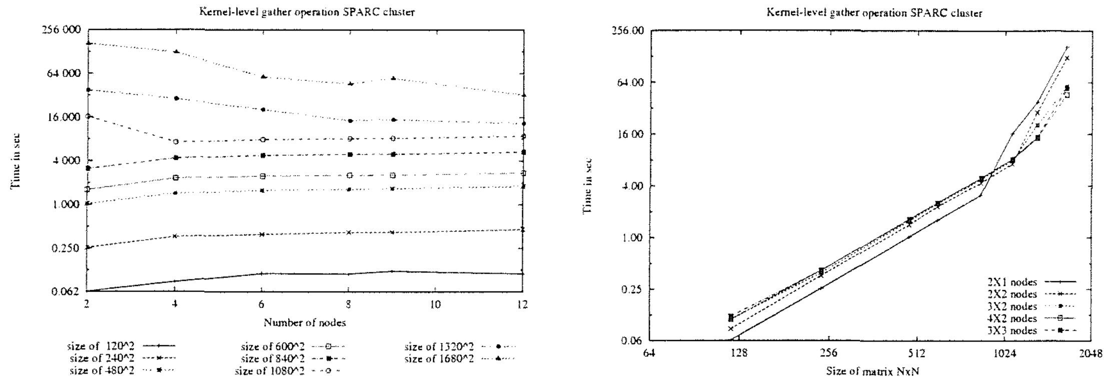

Figure 7.26: Kernel-level gather operation tests on the SPARC cluster

Performance difference between the scatter/gather functions and the broadcast call for the BIP cluster is rather related to different read data patterns. In gather/scatter operations data are accessed once usually with a stride which apparently is more costly than re-transmitting the same data as the broadcast call does [86].

7.5.3 The Shift Operation Test

The shift operation is often used as a compound operation in many message-passing parallel algorithms. MPI does not provide a single call for such an operation thus its implementation requires several single peer-to-peer calls. Unlike the kernel-level operations previous examined the shift operation is not a "strictly collective" call because performance is based on local calls among neighbouring processes. Scalability of this operation depends on the ability of the communication network to provide multiple overlapping communication paths. As the number of nodes increases the amount of transferred data is increasing also, i.e. $2(\sqrt{p}-1)N$. Implementation of this operation was tested for matrix size between 30 to 1080 square elements.

The shift operation benchmark was run on the SPARC and the BIP clusters for various number of nodes and array sizes as Fig. 7.27 illustrates. The SPARC cluster scales well only for a small number of nodes when the size of the shifted array is relatively small. As the number of nodes increases or the size of the array becomes larger then elapsed time increases rapidly without following a specific pattern. In contrary the BIP cluster scales well with the number of nodes and the size of the array. The cost of the shift operation, for the BIP cluster, depends on the array size but not the number of nodes. The bottom right figure shows this attribute very characteristically where all the plots of different nodes collapse into a single line. A small anomaly in the results was noticed on a 6-node-grid of the shift operation with large array sizes which is due to an uncontrolled background job on that sixth node during the test period. The effective bandwidth of the shift operation for the SPARC cluster is close to 1 Mbyte/sec, while on the BIP cluster once again the effective bandwidth for small messages is close to 42 Mbyte/sec but from medium to large size arrays it drops down to 30 Mbyte/sec.

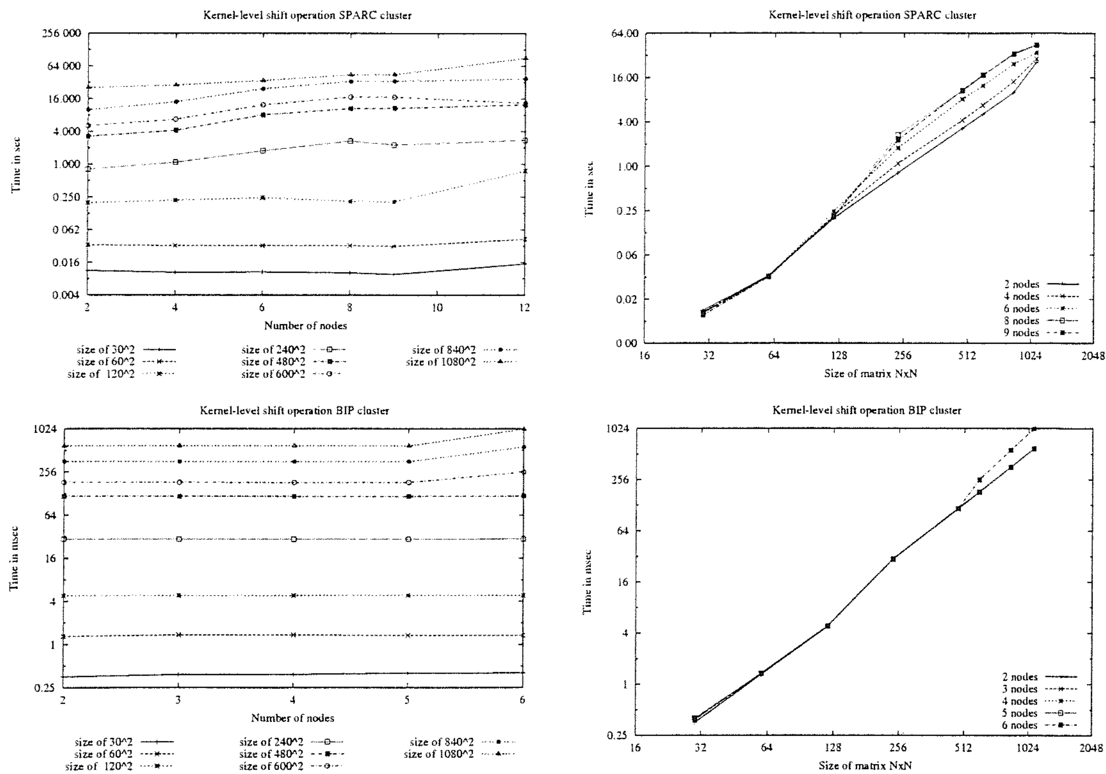

Figure 7.27: Kernel-level shift operation tests on the BIP cluster

7.5.4 Analysis of Kernel-level Operation Tests

Low-level tests targeting the absolute performance of a call, obtaining the raw performance by isolating the measured call with many iterations within the cache of the system. This level of performance is often unrealistic and hardly achievable from an application program irrespective of programming transformations and compiler optimisations. In kernel-level operation tests, routines and operations are examined from the context of communication within computation and from the point of view of the OS. Hence performance depends now on more realistic parameters such as communication protocol, OS handling, runtime programming techniques, algorithms. The two clusters of workstations used for kernel-level operation tests (the SPARC and the BIP cluster) represent two extreme ends of the current cluster intercommunication spectrum.

The 10 Mbit/s bus technology Ethernet network imposes a communication bottleneck that filters out any other activity of the node. Effective performance through the tests was limited below the 1 Mbyte/sec transmission barrier. Moving on to a Myrinet network with a user-space network protocol (such as the BIP) kernel-level operation tests performance becomes considerably better. The PCI communication barrier at 132 Mbyte/sec of the BIP cluster is not an issue here as other potential bottlenecks from the OS and hardware subsystems impose restrictions which limit performance to below 50 Mbyte/sec. The kernel-level operation performance of the BIP cluster ranges between 29% - 63% of the counterpart peer-to-peer send/receive communication low-level tests.

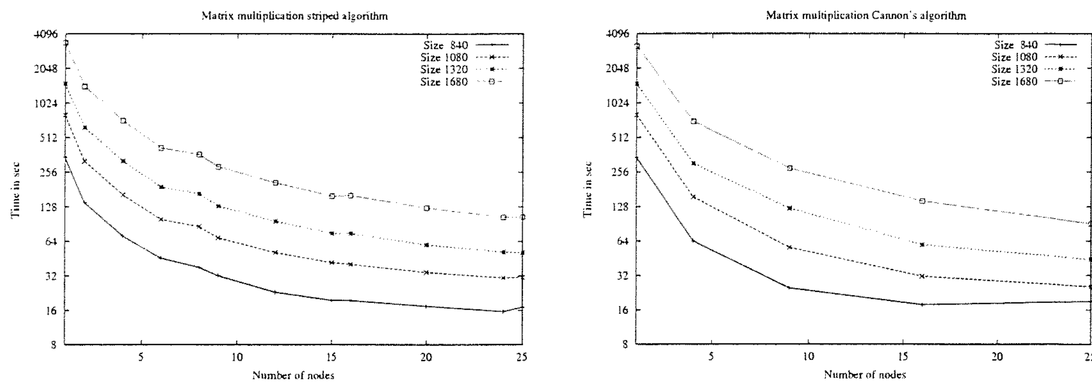

Figure 7.28: Matrix Row/Column Striped and Canon algorithm on the SGI cluster

Tests on the BIP cluster were run twice using different timing methods, the conventional one with *MPI_ Wtime*() calls and the register timer, e.g. the rdtsc(), via assembly calls (presented in section 6.3.5). Results did not show substantial differences because test measuring times were comparatively long (of the order of seconds).

7.6 Kernel-level Algorithmic Tests Results

This section presents kernel-level algorithmic benchmark results from a 12-node SPARC cluster and a 25-node SGI cluster configurations. The limited node number availability of the BIP cluster prevent the execution of the full scale of tests on this platform. Performance analysis and comparisons of kernel-level application tests will make use of information gained in previous low-level and kernel-level benchmarks. However performance on the tests is expected to be affected by the nature and the use of resources that each particular algorithm requires.

7.6.1 Matrix-matrix Benchmark Results

The matrix-matrix benchmarks consist of two different multiplication algorithms, the first algorithm is the Matrix Row/Column Striped and the other one is Cannon's algorithm. Both algorithms, described in previous chapters, have similar computational and communicational complexity on their nodes but their communication patterns and memory resource requirements are different.

Figure 7.28 illustrates the results of this benchmark for the SGI cluster. For a small number of nodes the two algorithms provide similar results, then as the number of nodes increases Cannon's algorithm starts giving better results. Finally for large number of nodes (and for smaller matrix sizes) the Row/Column Striped algorithm gives better results than Cannon's algorithm. There is a critical point at which further increase of the number of nodes generates a communication traffic overflow δt_{comm} that overwhelms any computational part improvement δt_{comp} of the algorithm and thus beyond that point there is no real benefit (see equation 7.2 and Fig. 7.23).

$$\delta t_{comp} > \delta t_{comm} \tag{7.3}$$

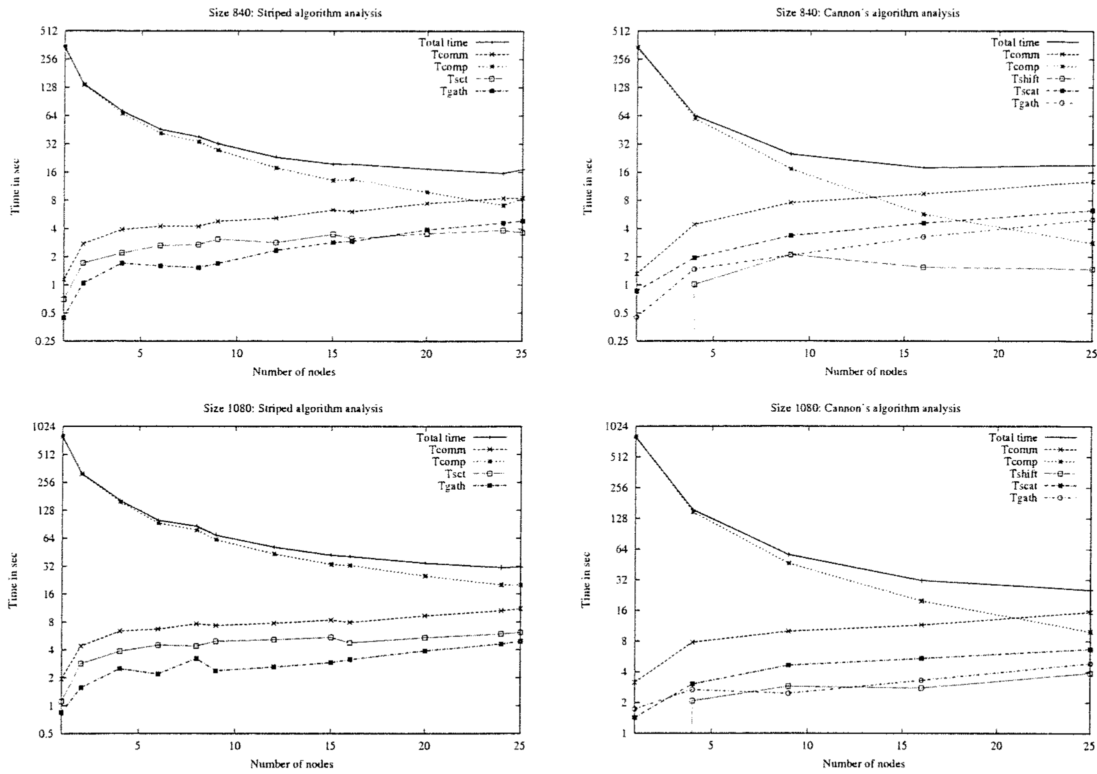

Figure 7.29: Communication versus computation part between matrix algorithms for the SGI cluster

This is more clear in Fig. 7.29 where the total cost of the algorithm is depicted along with its partial computational and communicational cost. For large number of nodes the Row/Column Striped algorithm uses fewer communication calls with larger messages than Cannon's algorithm. For a matrix size of 840 X 840 the critical point at which the number of nodes does not provide further improvement is beyond 16 nodes. On the other hand, for the Row/Column Striped algorithm that critical number of nodes is close to 24 nodes. Increasing the matrix size computation requirements increase exponentially and faster (complexity of $O(N^3/p)$) than communication requirements (complexity of $O(N + t_{ov} \cdot p)$) on a p number of nodes. Hence the critical saturation point is moving towards a higher number of nodes, for both algorithms in the tests for matrix sizes greater than 1000 X 1000 there was positive speedup for all the first 25 available nodes.

The SPARC cluster has a limited number of nodes (12 nodes which provides only 3 possible configurations for the Cannon's algorithm). Throughout the tests the Row/Column Striped algorithm implementation yields relatively better results for any number of nodes or matrix size than Cannon's algorithm does on this cluster. Apparently the bus network architecture of the SPARC cluster cannot tolerate the extra amount of communication calls i.e. shift operations, Cannon's algorithm requires in comparison to the SGI cluster with a switched network infrastructure. Figure 7.30 shows in detail the cost of the communication and the computation part of both algorithms for 600 X 600 and 1080 X 1080 matrix sizes respectively. In compar-

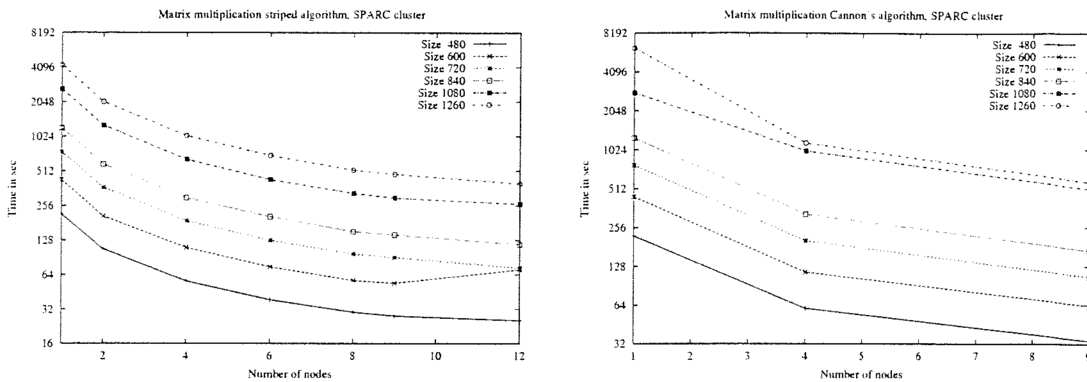

Figure 7.30: Matrix Row/Column Striped and Cannon's algorithm on the SPARC cluster.

ison with the SGI cluster results the slower computational and communication performance of these nodes it is noticeable as well their limited number. The computational part of the algorithm remains high even for small size matrices while the communication part remains high and unpredictable.

A modified version of the matrix multiplication algorithm was tested as well on these platforms. The modification attempts to increase the access locality of the inner-most loop of the computational part of the algorithm by transposing the second matrix B during its distribution among processes. This change in practice increases the communication cost of matrix B distribution for Ethernet and Fast Ethernet networked clusters only marginally but the computational part of the algorithm gains a substantial speedup. Results of those modified tests are presented in Appendix G. Figure 7.32 illustrates the relative speedup achieved amongst all the algorithms used for the SGI cluster for the matrix size of 1080 X 1080. The speedup for the SGI cluster is super-linear for the first 25 nodes. The computational part of the algorithm is clearly super-linear and in this case the communicational overhead is relatively small and does not overwhelm the computational speedup. In contrary the communication overhead of the SPARC cluster overwhelms quickly any computational part benefit and approaches at its best a linear curve for a 9-node configuration. Beyond that point speedup is increasing marginally without any clear benefit.

7.6.2 Sorting Algorithm Results

The PSRS algorithm test (presented earlier in section 6.7.4) run on the SGI and the SPARC cluster for sorting floating point vectors of size 1,2,4 and 8 million[2] cells. Figure 7.33 shows the results of the PSRS tests. Both the SGI and the SPARC system have similar behaviour on a different scaling, that is for a small number of nodes the qsort algorithm dominates elapsed time but as the number of nodes is increasing the communication part of the algorithm becomes the dominant part. The implementation of this algorithm requires a rather complicated set of communication structures which require many initialisation phases (e.g. vectorised calls for gather, scatter and all-to-all routines do not scale well with the number of nodes). This implies

[2] 1024*1024 elements

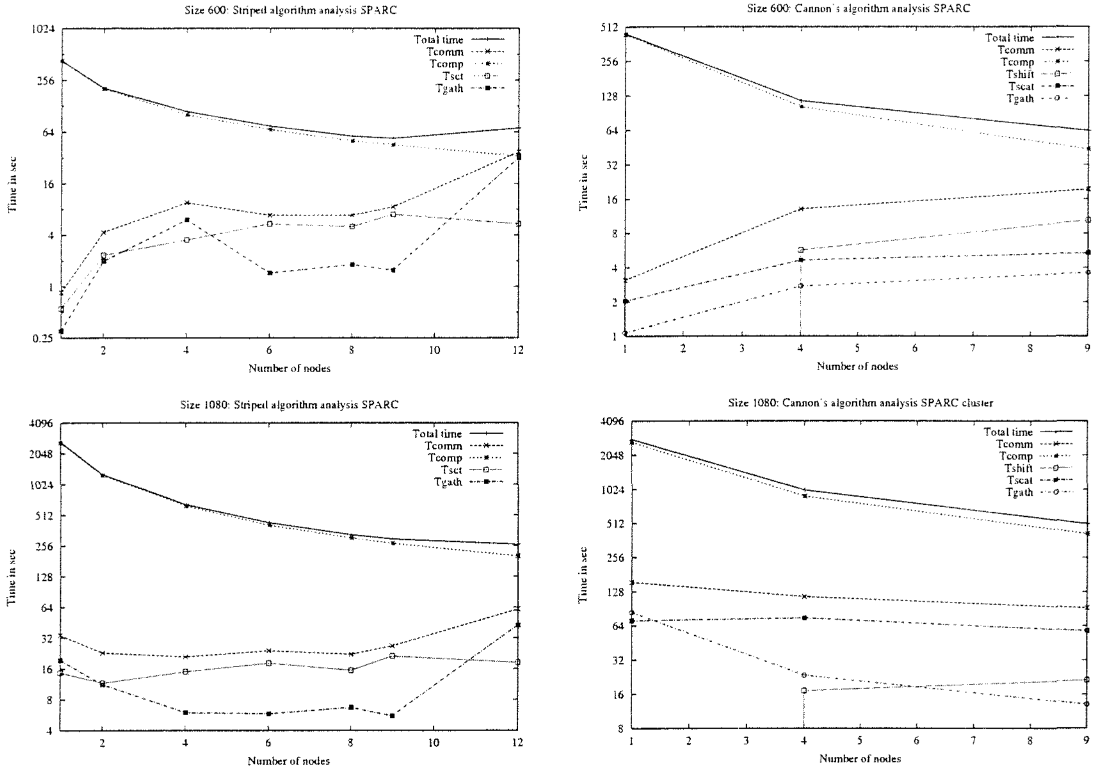

Figure 7.31: Communication versus computation part between matrix algorithms for the SPARC cluster

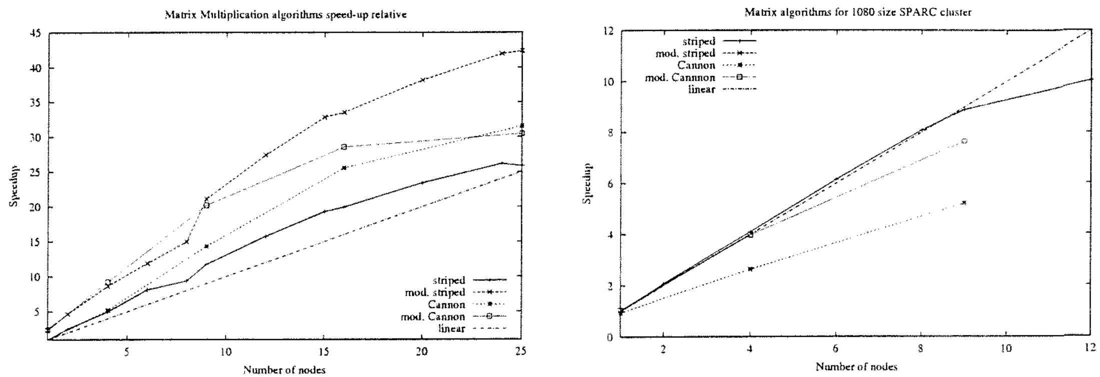

Figure 7.32: Relative speedup results of the multiplication algorithms

a high communication cost even for a small number of nodes (see Fig. 7.33 e, 7.33f).

The cost optimal complexity of the PSRS algorithm is $O((n/p)\log n)$ [185] where n is the size of the vector and p is the number of processors. Performance speedup for a small number of nodes is poor and saturates fast when scaling to larger number of nodes because of the excessive communication overhead. Despite the parallelisation algorithm, relative speedup is low <3.5 for the SGI cluster. An 8-node configuration for both clusters proved to provide the most effective configuration with a speedup between 1.8–3.3 for the SGI cluster, while speedup for the SPARC cluster ranges between 2–4.7 for large size vectors. Paradoxically the relative speedup of the SPARC cluster appears to be better than the SGI one simply because the computational performance of these nodes is relatively low.

7.6.3 Multi-grid Relaxation Test Results

The multi-grid relaxation test presented in section 6.7.5 is a mixture of Gauss-Jacobi and Gauss-Seidel iteration methods. Tests were run successfully on both the SGI and the SPARC network of workstations platforms for array sizes $N \ X \ N$ ranging between 720 – 1500 for a fixed number (1000) of iterations on a 16-node SGI and a 12-node SPARC cluster configurations respectively. Figure 7.34 illustrates the results of the relaxation tests run on these platforms.

Communication requirements for the domain decomposition phases are smaller and have a simpler complexity than the ones used for the matrix-matrix tests previously. The boundary condition data exchange cost between neighbouring nodes is also low because the amount of data each node has to exchange with its neighbouring nodes (halo of $4 \cdot n/\sqrt{p}$ elements) is limited and uniformly distributed within a four peer-to-peer calls cycle. This is the same operation which was studied previously with the shift operation test and it is proved scalable on a cluster with switched network such as the SGI cluster. Hence the aggregate cost of the repeating exchange operation remains relatively low and increases almost linearly with the halo size and the number of iterations. Speedup is almost linear but as the number of nodes increases the computational part becomes smaller and at the same time the communication cost gradually represents a significant fraction of the overall cost (e.g. the calculated speedup of the computation part for an array size of 1500 X 1500 running on 16 nodes is 14 and the overall speedup of the algorithm is 12.25).

7.6.4 Analysis of Kernel-level Algorithmic Tests

The scalability of the SCOPE kernel-level tests and other algorithms can be predicted in principle by analysing the algorithm complexity combined with information obtained at lower-level SCOPE tests. Such scalability prediction for data parallelism algorithms and applications is straight-forward by separating the computation and communication part of the algorithm.

Appendix F provides an example estimation about the cost of the SCOPE kernel-level algorithms. For the Matrix Row/Column Striped algorithm the communication cost is determined by the the sum of the three major communication operations for the sub-matrices distribution $T_{scatter}$, broadcast of sub-matrices along rows and columns communicators T_{bcast} and the gather operation at the end of the final stage of the algorithm T_{gather}. The combination of

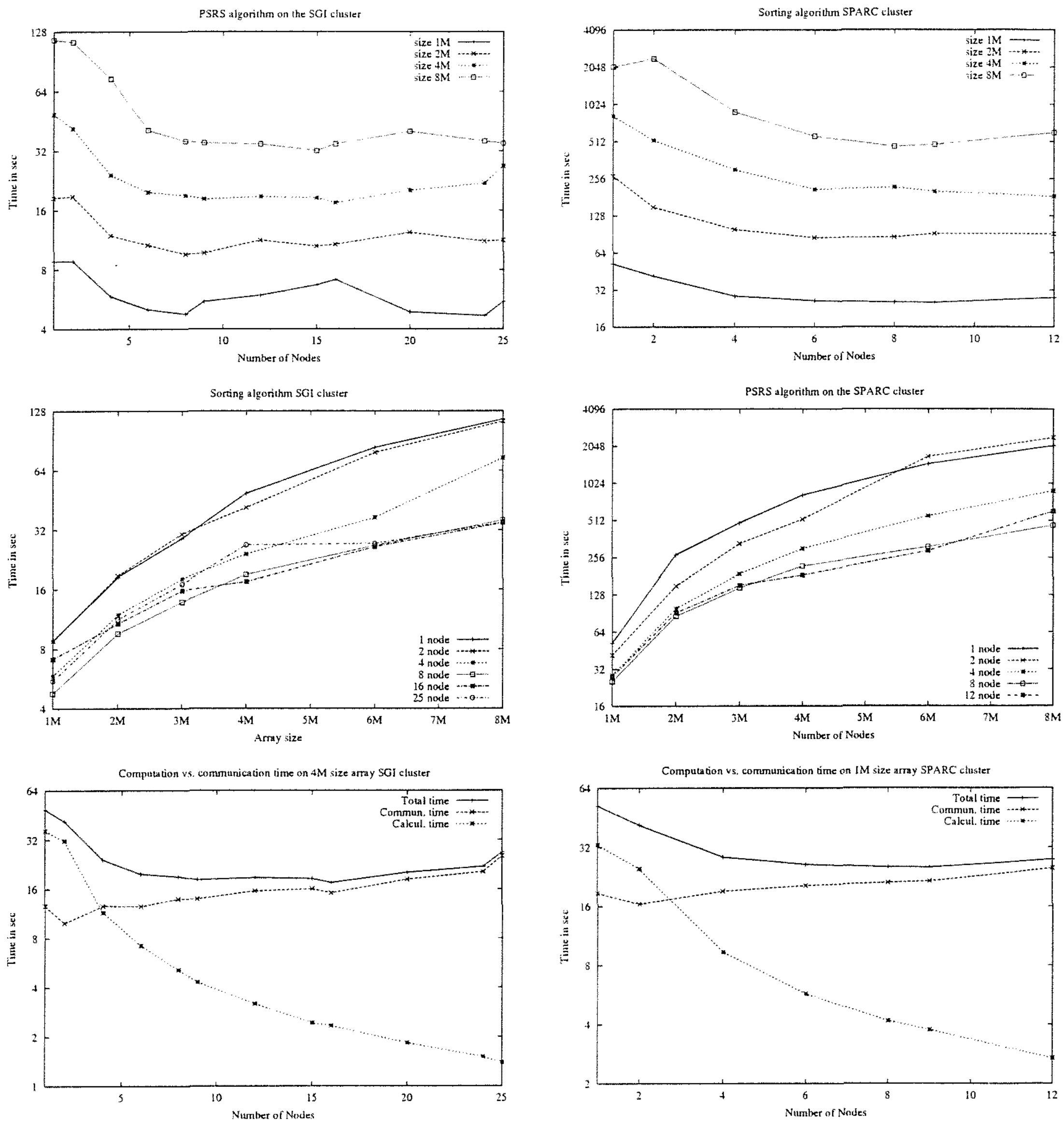

Figure 7.33: Sorting algorithm results

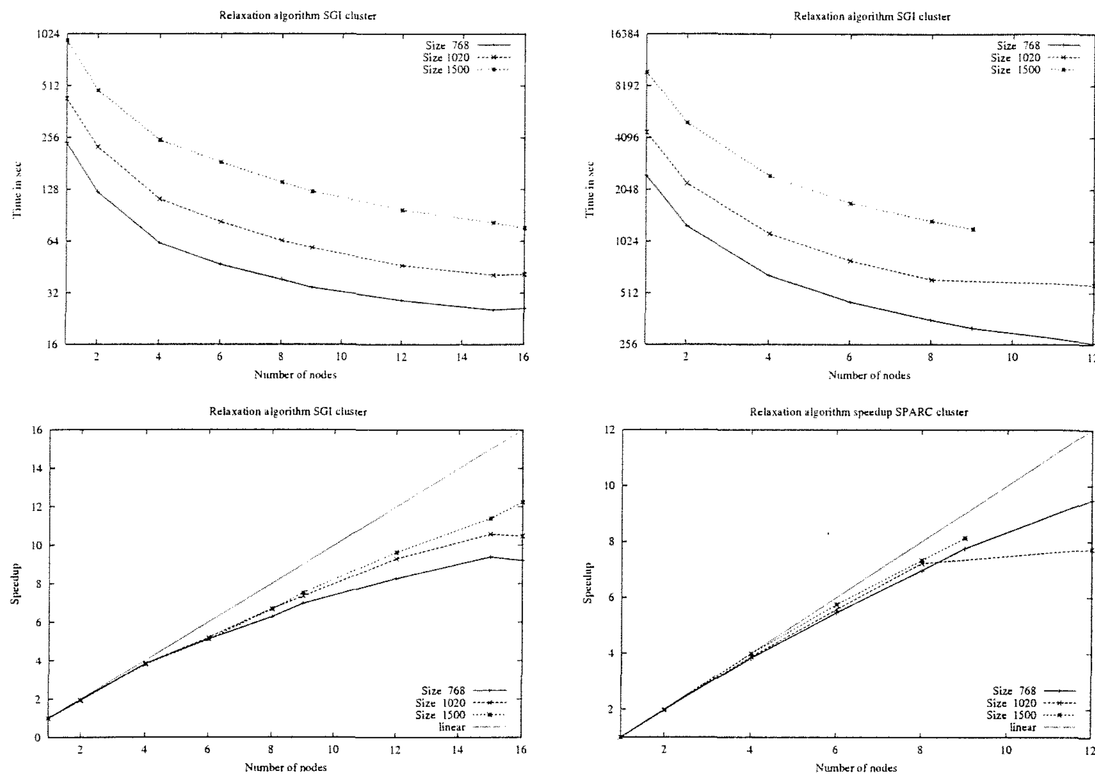

Figure 7.34: SGI cluster multi-grid relaxation test results

$T_{comm} \geq T_{scatter} + T_{bcast} + T_{gather}$ with equation 7.3 and 5.5 can provide information about the scalability of the algorithm. Results from kernel-level operation tests from section 7.5 and elapsed time of the algorithm running on a number of nodes are required too.

The objective of the SCOPE kernel-level algorithmic benchmarks is to measure performance of a workstation cluster closer to the programmer perspective. Cluster performance is examined within the context of a parallel task (computation and communication) amid other OS activities. Performance at this level is a combination of both the individual performances of the computation and the communication subsystems. Moreover kernel algorithmic tests measure the performance of the entire system and investigate how the system scales with a number of nodes and a large problem size. For example, the communication part of the algorithmic tests on the BIP cluster were on average 3-6 times faster than the SGI counterparts for the matrix multiplication algorithm, but the overall performance on a 4-node SGI cluster is overwhelming higher than the 4-node BIP system cluster because the SGI node computational power is significantly greater.

This is because performance at the higher levels depends on the nature of the algorithm and not only the raw performance of the hardware, e.g. communication hardware. Therefore for example the sorting algorithm scales rather poorly with the number of nodes. Problems with regular domain decomposition such as the Row/Column Striped multiplication algorithm gives a positive speed-up even on a 12-node shared bus network architecture cluster. For this kind of problem the computation part usually decreases (e.g. exponentially) with the number

Table 7.7: Comparisons with other benchmarks

Low-level network performance tests						
System	SCOPE		Other Benchmark			Hardware
	Lat	BW	Lat	BW	Test Name	specs
SPARC	212	1.082	–	1.13	netperf	1.25 Mbyte/s Ethernet
SGI	368	12.06	–	11.92	netperf	12.5 Mbyte/s Fast Eth.
BIP	6	121	5 μs	126	custom BIP	132 Mbyte/s Myrinet
Low-level message-passing tests						
SP2	50	31	52	32	PARKBENCH	40 Mbyte/s
CS2	107	35	113	38.6	PARKBENCH	50 Mbyte/s
Low-level synchronisation tests on two nodes						
SP2	91 μs		100 μs		PARKBENCH	–

of nodes but at the same time the communication part of the problem increases mainly due to the extra communication overhead rather than the actual payload (which is usually reduced for each call as the number of nodes increases). Thus algorithms with a higher computation to communication ratio will usually obtain an improved speedup figure. The positive speed-up of a system will stop at the time when the computation part improvement is balanced by the increasing communication part time. The problem size also affects that point because it mostly changes the computation part of the algorithm (usually exponential). This is also known as Gordon Bell's Law of Massive Parallelism based on application scaling[3] and Gustafson [100]. Hence as the problem size increases then speed-up improves as well with the number of nodes. In practice the optimal number of nodes for a specific problem of a certain size could be chosen well below that theoretical optimum when the economic cost of the extra nodes is taken into account.

7.7 SCOPE Overview

A comprehensive test and evaluation program for the SCOPE benchmark suite is a complex and difficult task which is beyond the scope of the thesis. This is because the various benchmarks have different workloads for each test. However, a brief examination and comparison of the results obtained with other well-known benchmarks can provide additional information about the reliability of the current SCOPE implementation.

The SCOPE low-level benchmarks detected correctly communication protocol changes on every platform, as well as other performance anomalies and bottlenecks such as memory bottlenecks. For example during the reduce test on the BIP cluster a performance anomaly was observed which was caused by a poor buffer allocation in the BIP implementation. The hier-

[3]According to Gordon Bell: "there exists a problem that can be made sufficient large such that any network of computers can run efficiently given enough memory, searching and work – but this problem may be unrelated to no other"

archical structure of the SCOPE benchmarks is evenly reflected on the results obtained from low-level to higher-level tests.

Sensible comparisons with other benchmark results is only feasible for a few of the SCOPE low-level tests. The underlying network level tests can be compared with other benchmarks such as netperf. In a similar way the communication library low-level tests can be compared with some of the Genesis or PARKBENCH tests such as COMMS-Pingpong1 or COMMS-Synch. In contrast, for kernel-level tests the workload among different benchmarks varies significantly and there is no common way of making comparisons. Table 7.7 illustrates results obtained from similar low-level benchmark tests run on common platforms, which gave almost identical results especially for some of the lower-level Genesis/PARKBENCH COMMS tests run on MPPs. This brief benchmark comparison confirm that the SCOPE benchmark suite implementation results are reliable and hence the benchmark suite has achieved its initial objectives.

7.8 Summary

This chapter has examined and analysed benchmark results obtained with the experimental SCOPE implementation on a variety of workstation cluster and MPP platforms. SCOPE tests on different abstraction layers were able to demonstrate accurately performance characteristics on every tested platform. Analysis and comparison of SCOPE tests has provided valuable inside information and understanding about potential performance bottlenecks on cluster sub-systems such as the communication network and the message-passing library.

Performance comparisons between SCOPE tests and other benchmarks on the same platforms, although not directly comparable, has shown that SCOPE results can provide a reliable performance guideline for workstation clusters.

Chapter 8

Future Work

The SCOPE benchmark suite presented in earlier chapters provides a tool for evaluation and research into the key performance issues of workstation clusters, rather than a development of a user-orientated software package. The work discussed in this chapter presents topics of user-orientated software development which are beyond the scope of this thesis and assume the availability of a well-defined user-friendly package which in turn creates an opportunity for novel research into the workstation cluster area. In addition this chapter discusses other research work on issues identified elsewhere in this thesis.

8.1 Standard Module and Baseline Tests

The SCOPE benchmark suite, as presented earlier in this thesis, is intended to address performance evaluation issues for workstation clusters and provide an overall performance characterisation for these parallel platforms. It is essential for the SCOPE benchmark suite to have an established baseline and standard modules of tests in order to become a useful and comprehensive performance evaluation tool for cluster administrators and application developers.

For these reasons the SCOPE benchmark suite will need to define a module of baseline tests and a minimum set of resource requirements for workstation clusters. The resource requirements are important because they will ensure that SCOPE baseline tests will run on cluster platforms which will provide sensible results and allow comparisons with other parallel systems. Typical minimum requirements will be, for example, the number of nodes, the underlying network configuration, the node memory size/hierarchy, the type/speed of the CPU, type of the OS and installed software tools. For example minimum hardware requirements could include features such as those included in Table 8.1.

The baseline module of tests will need to define which tests will be used together with their workload characterisation in a way that will provide a common basis for tests and workload configuration over a wide range of workstation platforms. The tests and results presented in Chapters 6 and 7 respectively, provide a useful starting point for the baseline module of tests. As the baseline tests will provide results only on the minimum required resources, users will be also interested in having tests running up to the limits of their current workstation cluster

Table 8.1: Minimum hardware system requirements for the SCOPE baseline tests

Cluster Features		
System size	number of nodes	> 4
Node size	RAM size	32 Mbyte
	Cache size	512 Kbyte
	CPU Performance	> 8.6 SPECint95 [1] > 6.5 SPECfp95 [1]
Message-passing system	Network bandwidth	10 Mbit/s or better
	Network latency	<250 microsecond

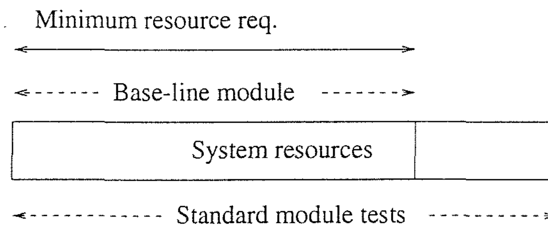

Figure 8.1: Baseline and standard module tests

These tests will be known as "standard module tests" and they represent an expansion of the baseline tests.

There is a subtle difference between baseline and standard module tests. Baseline test results will provide a common base for direct comparisons between clusters, while standard module tests will provide information about the maximum capabilities of each system. The establishment of both a baseline test module and a minimum configuration requirement for a workstation cluster will facilitate the automatic installation procedure of the benchmark software as well as simplifying the actual test procedure.

8.2 Results: Analysis and Presentation

The data output resulting from each benchmark execution is large and essentially meaningless without using appropriate analysis and presentation utilities. The SCOPE benchmark will need such utilities to analyse and present results in a concise and comprehensive way. Result acquisition and analysis for each cluster can be automated by the use of scripts and configuration tools. The definition and establishment of a baseline and a standard module of tests will assist to standardise the way results are analysed and presented.

The presentation of the results in Chapter 7 of this thesis used several custom scripts and tools in order to automate the data analysis procedure e.g. to enable the automatic creation of result reports. Further improvement, for example, will be the use of a visualisation tool or a 3-D

[1]SPEC rates are taken from a Pentium Pro 200MHz system equivalent to 200 Mflop/s

presentation of collective routine results [163]. Baseline test results will be compared directly with similar results taken from other clusters using a common benchmark result depository database or a public web site [132].

The non-deterministic nature of the underlying intercommunication networks used in workstation clusters together with the asynchronous mode of the message-passing model and other runtime system activities often causes unpredictable delays when sending or receiving messages. These delays frequently result in substantial measurement fluctuation which can provide additional information with statistical analysis.

A quantitative analysis of benchmark results is often useful to determine a particular behaviour of a cluster. Chapters 6 and 7 introduced simple analytic models to explain and understand system behaviour on various SCOPE tests. Most of the models discussed are relatively straightforward and in some cases do not accurately explain the complicated underlying hardware and software mechanisms. Further research in this area will improve the analysis of workstation clusters and will facilitate the reliable performance optimisation of complicated algorithms on clusters.

8.3 Advanced and Experimental Module Tests

Benchmarks in general should not disbar or discourage innovative hardware technologies or software techniques. The SCOPE benchmark should encourage and help cluster designers and programmers to experiment and use novel enhancements either in hardware configuration architecture of clusters or in experimental software techniques and algorithms tailored for clusters of workstations, e.g. development and evaluation of latency-tolerant algorithms.

The SCOPE benchmark will introduce an advanced model of test which will assist experimenting with new hardware and software features over possible improvements on workstation clusters. Tests in this module will run in a non-standard mode in which the workload can change. For this reason advanced module tests will be described as "paper and pencil" tests or modified existing tests according the needs of the experimental conditions. For example specific cluster optimisations can be applied to compiler parameters or algorithms i.e. dependence analysis and various algorithm transformations and optimisations.

Tests in advanced and experimental mode usually have to be carried out individually and are difficult to automate. In the later case the user needs to accept the responsibility for any benchmark changes as well for the correct interpretation of the results. Comparisons with standard mode test results should be cautious and usually carried out after a normalisation procedure has taken place.

8.4 Benchmark Expansion and Future Tests

The rapid evolution in both hardware and software technology will inevitable bring changes in workstation clusters as well. The SCOPE benchmark suite will need constantly to re-define and re-establish its benchmarks according to the current needs and trends at each time. The main

directions in which future tests of the SCOPE benchmark suite will focus on should include:

- Tests with non-blocking communication calls have to be added especially for those tests that include communication and computation parts which can be overlapped such as algorithmic and kernel-level tests.

- The new features introduced in MPI-2 such as single-sided function i.e. tests for put and get calls and dynamic process management (see Chapter 4). Progress in this area is not possible at present because of the inadequate MPI-2 implementations available for workstation clusters.

- Parallel I/O tests, Appendix D presents some preliminary work in this direction with tests on the proposed MPI-2 parallel I/O standard using the ROMIO implementation included in the latest MPICH distribution.

- Specific tests for clusters of symmetrical multiprocessors (SMP). COTS technology offers both the hardware and software infrastructure (e.g. 64-bit hardware multiprocessor architectures and real SMP mode support OS). The SCOPE benchmark suite will be extended to include tests such as multi-threading synchronisation/communication and other tests for shared memory mechanisms adapted by the message-passing model appropriate for clusters of SMP nodes e.g. put and get.

- Heterogeneity; there are two main types of heterogeneity that can be defined for clusters of workstations. The first type is a heterogeneous cluster which has modes with different hardware and software architectures (i.e. different data format and computational speed). The second type of heterogeneous cluster includes nodes that have identical software architecture i.e. the same OS and data format, but they differ in the hardware architecture i.e. computational speed[2].
  The first type of heterogeneity is more difficult to analyse and its performance estimation is more difficult. The second type of heterogeneity can be easily found as a gradual upgrading process of existing homogeneous clusters. In either case the most straightforward approach is to split cluster into sub-clusters with equivalent performance nodes and the SCOPE benchmark run individually for each of these sub-clusters.

8.5 SCOPE and Other Benchmarks

Interoperability aspects between the SCOPE benchmark suite and other benchmarks need to be integrated. Data results from the SCOPE tests could be converted to a format that can be used and processed with existing well-known benchmark data analysis tools (such as the Graphical Benchmark Information Service and the PARKBENCH Interactive Curve-fitting Tool). Kernel-level tests from known benchmarks such as NAS kernel tests can also be integrated within the SCOPE kernel-level test methodology.

[2]It mainly implies different processor speed and size of hardware resources e.g. compare for example computing platforms based on SPARC and UltraSPARC.

Table 8.3: Workstation (w/s) architectures and OS SCOPE distribution will support

Node Hardware Architecture	Operating system
PC-based w/s	NT
PC-based w/s	Linux
Sun SPARC w/s	Solaris
SGI w/s	IRIX
IBM w/s	AIX

This will increase the usability of the SCOPE benchmark and at the same time will enhance performance evaluation comparisons between workstation clusters and other parallel platforms. In a similar way tests from the SCOPE benchmarks suite could be used on MPP or SMP systems.

8.6 Other Issues

Other issues that the SCOPE benchmark suite needs to address are improved documentation and the usual software package distribution issues necessary to facilitate efficient installation on a wide platform of workstation architectures (Table 8.3).

The use of register timers as the primary timing mechanism can be expanded to all the platforms presented in Table 8.3. This will improve measurement accuracy and can also enable detailed functionality tests. Furthermore application programmers will be able to adapt the high-accuracy timing mechanism used in the SCOPE benchmark suite to instrument and monitor their own application code.

Batch scheduler management issues and other runtime system utilities for workstation clusters are not widely used or standardised. Currently the SCOPE benchmark tests use small script programs to launch tests on clusters. The introduction of a standardised job scheduler for workstation clusters will improve significantly the SCOPE benchmark test running procedure although it is not directly relevant to the SCOPE benchmark suit interests.

Chapter 9

Conclusions

9.1 The Requirement for HPC and Workstation Clusters

The requirement for HPC is constantly increasing as application demands for significant computing power are continuously increasing. A new generation of commercial applications such as e-commerce and e-business, along with "traditional" scientific Grand Challenge Applications, are generally increasing requirements for HPC. Parallelism is the key enabling technology which can deliver the required computing performance for these large and very large scale scientific and commercial applications. Although in principle the concept of aggregated computational power available by means of parallelism is straightforward, implementation in practice has proved to be a far more difficult task than originally envisaged. For this reason, most of the parallel systems built in the past were complicated (proprietary) and because they were relatively expensive they were used only in few large organisations for "traditional" scientific Grand Challenge Applications.

Workstation clusters using commodity components (sometimes referred as COTS) have the potential to provide, at low cost, an alternative parallel platform suitable for many HPC applications. Although in practice workstation clusters cannot replace completely MPPs or mainframe systems, they can provide an entry-level HPC solution with excellent scalability, availability, maintainability and performance/price characteristics for many large-scale applications.

The first part of this thesis investigates and establishes the current status of the workstation cluster concept. Chapters 2 and 3 have discussed and examined how over the past few years the key hardware and software components of the workstation cluster infrastructure such as node architecture, interconnection and OS functionality have improved their performance dramatically. In additional the inherent programming model of clusters, which is a multicomputer message-passing parallel model, has become well established with the advent of MPI and PVM.

Chapter 4 has examined the fundamental concepts of the message-passing model which is straightforward and can provide efficient SPMD and MPMD programming styles on distributed memory (DM) platforms. This is important because all these parallel platforms currently adopt the same computation model. Hence the potential advantage of this is that the techniques and

methodology developed for parallel systems as well as parallel applications can be directly adapted and used for clusters of workstations.

9.2 Evaluation of Workstation Clusters with SCOPE

Having established the workstation cluster concept in the first part of this thesis, the second part then investigates the provision of a novel performance evaluation tool that will assist to understand and analyse the performance behaviour of these systems.

In the past, workstation clusters were often wrongly classified either as distributed systems or as loosely-coupled MPPs. This is because workstation clusters borrow many components, techniques and research which were primarily designed for other platforms. In practice this combination of technologies provides several advantages for clusters but at the same time the evolution and performance of clusters is determined and limited by technologies designed for other systems. Clusters still suffer from inherent drawbacks such as long latencies, low bandwidth, lack of a "single system image" in terms of software programming environment and inadequate administration tools. Recent research in communication protocols presented in Chapter 9 has shown that improvements in the inherent limitations of clusters are feasible.

Research in this thesis has been focused around the concept of workstation cluster as a HPC platform. In particular a tailored benchmark suite for clusters called Specific Cluster Optimisation and Performance Evaluation (SCOPE) has been proposed and an initial implementation investigated in Chapter 6. The SCOPE benchmark suite, as proposed in this thesis, contributes to the scientific benchmark methodology for the comprehensive examination of workstation cluster performance characteristics. Among the objectives of this benchmark suite is the promotion of the workstation cluster concept by evaluating potential characteristics and performance. This will assist commodity workstation cluster designers to understand and analyse the performance behaviour of these systems better. Moreover, the SCOPE benchmark methodology is flexible and provides application developers with a useful tool to understand and program clusters more efficiently.

In order to achieve these objectives the SCOPE methodology in chapter 6 proposes a relatively small number of additional tests, in comparison to well-known parallel benchmark suites, which enable users to evaluate in greater detail performance measurements inside the multi-layered structure of workstation clusters. Low-level tests examine thoroughly the underlying network performance as well as the performance of primitive and commonly used message-passing communication library routines. Kernel-level and algorithmic-level benchmarks examine in detail the realistic performance of the system delivered at the application level. Workloads and tests at these levels consist mainly of common operations used in typical parallel algorithms such as domain decomposition problems. Hence performance comparisons at different stages and levels within the cluster structure become meaningful and provide a comprehensive picture of any cluster performance disparity between projected and delivered performance under different workload levels.

An initial implementation of the SCOPE benchmark suite was tested on a variety of workstation clusters with different internetworking technologies such as Ethernet, Fast Ethernet

and Myrinet networks. The results of these tests presented in chapter 7 demonstrate that the SCOPE benchmark suite is sufficient flexible to adapt and run useful tests on several different cluster configurations. The results also demonstrate potential performance characteristics on various cluster sub-systems and provided valuable information about the overall cluster performance behaviour and the run-time environment. Analytic performance models for low-level tests developed in Chapter 6 were also verified by the actual test measurements in Chapter 7. Many of the low-level benchmarks were also tested on parallel MPP systems. Results from these tests establish a clear and direct comparison performance guidelines between workstation clusters and MPP parallel platforms but at the same time the tests on MPPs also verify and validate the SCOPE test suite as well.

The potential infrastructure for workstation clusters is available to take full advantage of existing hardware and software to provide a viable inexpensive parallel systems. Technological advances in commodity computing components performance is expected to continue in the foreseeable future. Additionally, the increased need for HPC and the availability of this parallel platform will expand the usage of parallel programming, advancing further the overall field of high performance computing. The SCOPE performance evaluation tool proposed in this thesis has demonstrated the potential to identify and classify the performance evaluation of all workstation clusters. Moreover the SCOPE evaluation tool methodology can be expanded and provide support for the development of parallel applications and algorithms tailored to a specific parallel platform. The combination of these features denotes that the SCOPE benchmark has the potential to play a major role in developing clustered machines and applications together in a way that can exploit the full computational capacity of the underlying systems.

Bibliography

[1] Cliff Addison, James Allwright, Norman Binsted, Nigel Bishop, Bryan Carpenter, Peter Dalloz, David Gee, Vladimir Getov, Tony Hey, Roger Hockney, Max Lemke, John Merlin, Mark Pinches, Chris Scott, and Ivan Wolton. The Genesis Distributed-Memory Benchmarks. Part 1: Methodology and General Relativity Benchmark with Results for the SUPRENUM Computer. *Concurrency: Practice and Experience*, 5(1):1–22, February 1993.

[2] R. Alasdair, A. Bruce, J.G. Mills, and A.G. Smith. CHIMP/MPI User Guide. ftp.epcc.ed.ac.uk/pub/chimp/release/doc/user.ps.Z, 1994.

[3] G. S. Almasi and A. Gottlieb. *Highly Parallel Computing*. Benjamin/Cummings, second edition, 1994.

[4] G. Amdahl. The validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS conference proceedings, Spring Joint Computing Conference*, volume 30, pages 483–485, 1967.

[5] Akkihebba L. Ananda and Balasubramaniam Srinivasan. *Distributed Computing Systems: Concepts and Structures*. IEEE Computer Society Press Reprint Collection. IEEE Computer Society Press, Los Alamitos, California, 1991.

[6] T. E. Anderson, D. E. Culler, and D. A. Patterson. A case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[7] ANSA. *The Advanced Networks Systems Architecture (ANSA) Reference Manual*. Castle Hill, 1989.

[8] T. Aoyama, I. Tokizawa, and K. Sato. ATM VP-based broadband networks for multimedia services. *IEEE Communications*, 31(4):30–39, April 1993.

[9] S. Araki, A. Bilas, C. Dubnicki, J. Edler, K. Konishi, and J. Philbin. User-Space Communication: A Quantitative Study. In ACM, editor, *SC'98: High Performance Networking and Computing: Proceedings of the 1998 ACM/IEEE SC98 Conference: Orange County Convention Center, Orlando, Florida, USA, November 7–13, 1998*, New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1998. ACM Press and IEEE Computer Society Press.

[10] Semiconductror Industry Association. The national technology roadmap for semiconductors. http://www.sematech.org and http://itrs.net/ntrs/publntrs.nsf, 1998. International Technology Roadmap for Semiconductors 1998 Update.

[11] J. Bacon. *Concurrent Systems An Intergrated Approach to Operating Systems, Database, and Distributed Systems.* Addison Wesley, 1993.

[12] G. Bell. The Future of High Performance Computers in Science and Engineering. *Communications of the ACM, CACM*, 32(9):1091–1101, September 1989.

[13] G. Bernard, A. Duda, Y. Haddad, and G. Harrus. Primitives for distributed computing in a heterogeneous local area network environment. *IEEE Transaction on Software Engineering*, 15(12):1567–1578, December 1989.

[14] David Bernstein, Mauricio Breternitz, Jr., Ahmed M. Gheith, and Bilha Mendelson. Solutions and debugging for data consistency in multiprocessors with noncoherent caches. *International Journal of Parallel Programming*, 23(1):83–103, February 1995.

[15] B. Bershad, D. Ching, E. Lazowska, J. Sanislo, and M. Schwartz. A Remote Procedure Call Facility for Interconnecting Heterogeneous Computer Systems. *IEEE Transactions on Software Engineering*, SE-13(8):880–894, August 1987.

[16] B. N. Bershad, D. T. Ching, E. D. Lazowska, J. Sanislo, and M. Schwartz. A remote procedure call facility for interconnecting heterogeneous computer systems. *IEEE Transactions on Software Engineering*, SE-13(8):880–894, August 1987.

[17] Raoul A. F. Bhoedjang, Tim Rühl, and Henri E. Bal. User-Level Network Interface Protocols. *Computer*, 31(11):53–60, November 1998.

[18] A. Bilas and J. P. Singh. The effects of communication parameters on end performance of shared virtual memory clusters. In *Proc. of Supercomputing'97*, November 1997.

[19] Andrew Birrell and Bruce Nelson. Implementing Remote Procedure Calls. *ACM Trans. Computer Systems*, 2(1):39–59, February 1984.

[20] G. Blair, G Coulson, and N. Davies. Standards and platforms for open distributed processing. *IEE Electronics and Communication Engineering Journal*, 8(3):123–133, June 1996.

[21] J. M. Blum, T. M. Warschko, and W. F. Tichy. PSPVM: Implementing PVM on a High-Speed Interconnect for Workstation Clusters. *Lecture Notes in Computer Science*, 1156:235–235, 1996.

[22] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and Wen-King Su. Myrinet: A gigabit-per-second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[23] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and Wen-King Su. Myrinet: A gigabit-per-second Local Area Network. *IEEE Micro*, 15(1):29–36, Feb 1995.

[24] D. Borman, R. Braden, and V. Jacobson. RFC 1323: TCP extensions for high performance, May 1992. Obsoletes RFC1185.

[25] J. Y. Le Boudec. The asynchronous transfer mode: A tutorial. *Computer Networks and ISDN Systems*, 24:279–309, 1992.

[26] L.S. Brakmo and L.L. Peterson. Performance problems in BSD4.4 TCP. Technical report, University of Arizona, 1994.

[27] M. Broxton. Study of Performance and Optimization of MPI Over 100BaseT Switched Ethernet Network. MIT, Laboratory for Computer Science, Computation Structures Group Memo 412, August 1998.

[28] Jehoshua Bruck, Danny Dolev, Ching-Tien Ho, Marcel-Cătălin Roşu, and Ray Strong. Efficient Message Passing Interface (MPI) for Parallel Computing on Clusters of Workstations. *Journal of Parallel and Distributed Computing*, 40(1):19–34, January 1997.

[29] Greg Burns, Raja Daoud, and James Vaigl. LAM: An open cluster environment for MPI. pcmmpi, 1994.

[30] R. Butler and E. Lusk. Monitors, message, and clusters: The p4 parallel programming system. *Parallel Computing*, 20:547–564, 1994.

[31] F. Cappello, O. Richard, and D. Etiemble. Performance of the NAS benchmarks on a cluster of SMP PCs using a parallelization of the MPI programs with OpenMP. *Lecture Notes in Computer Science*, 1662:339–??, 1999.

[32] T. L. Casavant and J. G. Kuhl. A taxonomy of scheduling general-purpose distributed computing systems. *IEEE Trans. on Software Engineering*, 14(2):141–154, 1988.

[33] Maui High Performance Computing Center. SP Parallel Programming Workshop Message Passing Interface (MPI), January 1997.

[34] Sheue-Ling Chang, David Hung-Chang Du, Jenwei Hsieh, Rose P. Tsang, and Mengjou Lin. Enhanced PVM communications over a high-speed LAN. *IEEE parallel and distributed technology: systems and applications*, 3(3):20–32, Fall 1995.

[35] F. Cheng, P. Vaughan, D. Reese, and A. Skjellum. *The Unify System*. Engineering Research Center, Mississippi State University, June 1 1994. Wed, 6 Dec 1995 22:29:48 GMT.

[36] Andrew A. Chien, Mark D. Hill, and Shubhendu S. Mukherjee. Cover Feature: Design Challenges for High-Performance Network Interfaces. *Computer*, 31(11):42–45, November 1998.

[37] G. Chiola and G. Ciaccio. Gamma: a low-cost network of workstations based on active messages. In *Proceedings of PDP'97 5th EUROMICRO workshop on Parallel and Distributed Processing*, January 1997.

[38] G. Chiola and G. Ciaccio. Implementing a low cost, low latency parallel platform. *Parallel Computing*, 22(13):1703–1717, Feb 1997.

[39] G. Chiola and G. Ciaccio. Active Ports: A Performance-Oriented Operating System Support to Fast LAN Communications. In *EuroPar'98 Parallel Processing*, volume 1470 of *Lecture Notes in Computer Science*, pages 622–624. Springer Verlag, September 1998.

[40] G. Ciaccio. *A Communication System for Efficient Parallel Processing on Clusters of Personal Computers.* PhD thesis, Dipartimento di Iinformatica e Scienze dell'Iformazione Universita di Genova, feb 1999. DISI-TH-1999-02 http://www.disi.unige.it/ caccio.

[41] D. Clark, V. Jocabson, J. Romkey, and M. Salwen. An analysis of TCP processing overhead. *IEEE Communications Magazine*, 27(6):23–29, June 1989.

[42] David Clark. Focus: ASCI Pathforward to 30 Tflops and beyond. *IEEE Concurrency*, 6(2):13–15, April/June 1998.

[43] David Clark. Focus: Supercomputing: The next generation. *IEEE Computational Science & Engineering*, 5(4):79–81, October/December 1998.

[44] David Clark. Technology news: Heavy traffic drives networks to IP over Sonet. *Computer*, 31(12):17–20, December 1998.

[45] D. Cohen. Myrinet–on–VME Protocol Specification Draft Standard. Technical report, VITA Standards Organisation, January 1998. url:http://www.vita.com.

[46] Danny Cohen, Gregory Finn, Robert Felderman, and Annette DeSchon. ATOMIC: A low-cost, very-high-speed, local communication architecture. In P. Bruce Chen, C.Y. Roger; Berra, editor, *Proceedings of the 1993 International Conference on Parallel Processing. Volume 1: Architecture*, pages 39–46, Syracuse, NY, August 1993. CRC Press.

[47] D. E. Comer and D. L. Stevens. *Internetworking with TCP/IP, Volume III: Client-Server Programming wand Applications, BSD Socket Version.* Prentice Hall, Engelwood Cliffs, NJ, 1993.

[48] Compaq, Intel, Microsoft. *Virtual Interface Architecture Specification*, version 1.0 edition, December 16 1997.

[49] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems—Concepts and Design. 2nd Ed.*, chapter 17, pages 517–544. Addison-Wesley Publishers Ltd., 1994.

[50] C. Cranor, R. Gopalakrishnan, and P. Onufryk. Architectural Considerations for CPU and Network Interface Integration. *IEEE Micro*, 20(1):18–26, January/February 2000.

[51] D. E. Culler, A. Arpaci-Dusseau, R. Arpaci-Dusseau, B. Chun, S. Lumetta, A. Mainwaring, R. Martin, C. Yoshikawa, and F. Wong. Parallel Computing on the Berkeley NOW. In *Joint Symposium on Parallel Processing*, Kobe Japan, 1997. JSPP'97.

[52] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and van Eicken Thorsten. LogP: Towards a realistic model of parallel computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles Practice of Parallel Programming*, pages 1–12, May 1993.

[53] David E. Culler, Lok Tin Liu, Richard P. Martin, and Chad O. Yoshikawa. Assessing Fast Network Interfaces. *IEEE Micro*, 16(1):35–43, February 1996.

[54] Thomas Sterling Tom Cwik, Don Becker, John Salmon, Mike Warren, and Bill Nitzberg. An assessment of beowulf-class computing for nasa requirements: Initial findings from the first nasa workshop on beowulf-class clustered computing. In *Proceedings IEEE Aerospace*, 1998.

[55] Chris Dalton, Greg Watson, David Banks, Costas Calamvokis, Aled Edwards, and John Lumley. Afterburner. *IEEE Network*, 7:35–43, July 1993.

[56] Stefanos N. Damianakis. Efficient connection-oriented communication on high-performance networks (thesis). Technical Report TR-582-98, Princeton University, Computer Science Department, April 1998.

[57] M. A. R. Dantas. *Efficient Scheduling of Parallel Applications on Workstation Clusters*. PhD thesis, University of Southampton Electronic and Computer Science Dep., Sept 1997.

[58] M. A. R. Dantas. Evaluation of Process Migration for Parallel Heterogeneous Workstation Clusters. In *EuroPar'98 Parallel Processing*, volume 1470 of *Lecture Notes in Computer Science*, pages 397–400. Springer Verlag, September 1998.

[59] M. A. R. Dantas and E. J. Zaluska. Improving load balancing in an MPI environment with resource management. In Heather Mary Liddell, A. Colbrook, B. Hertzberger, and P. Sloot, editors, *High-performance computing and networking: international conference and exhibition, HPCN EUROPE 1966, Brussels, Belgium, April 15–19, 1996: proceedings*, volume 1067 of *Lecture notes in computer science*, pages 959–960. Springer-Verlag, 1996.

[60] M. A. R. Dantas and E. J. Zaluska. Efficient scheduling of mpi applications on networks of workstations. *Future Generation Computer Systems*, 4(13):489–499, June 1997.

[61] J. Dongarra, J. Martin, and J. Worlton. Computer Benchmarking: Paths and Pitfalls. *IEEE Spectrum*, 24(7):38–43, July 1987.

[62] J. J. Dongarra. The LINPACK benchmark: an explanation. In E. N. Houstis, T. S. Papatheodorou, and C. D. Polychronopoulos, editors, *Supercomputing. 1st International Conference Proceedings*, pages 456–474, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1988. Springer-Verlag.

[63] J. J. Dongarra and T. Dunigan. Message-passing performance of various computers. Technical Report UT-CS-95-299, Department of Computer Science, University of Tennessee, July 1995. Fri, 19 Sep 97 22:07:00 GMT.

[64] Jack Dongarra, Rolf Hempel, Anthony J. G. Hey, and David W. Walker. A proposal for a user-level, message passing interface in a distributed memory environment. Technical Report ORNL/TM-12231, Engineering Physics and Mathematics Division, Mathematical Sciences Section - Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, June 1993.

[65] Jack J. Dongarra. The Complete Linpack Report. Technical report, University of Tennesse, July 1994.

[66] P.W. Dowd, S.M. Srinidhi, F.A. Pellegrino, T.M. Carrozzi, D.L. Guglielmi, and R. Claus. Impact of transport protocols and message passing libraries on cluster-based computing performance. In *Proceedings of IEEE*, October 1995.

[67] Dave Dunning, Greg Regnier, Gary McAlpine, Don Cameron, Bill Shubert, Frank Berry, Anne Marie Merritt, Ed Gronke, and Chris Dodd. The Virtual Interface Architecture. *IEEE Micro*, 18(2):66–76, March/April 1998.

[68] P. Ein-Dor. Grosch's Law Revisited: CPU Power and the Cost of Computation. *Commun. ACM*, 28(2):142–151, 1985.

[69] Jakob Engblom. Why SpecInt95 should not be used to benchmark embedded systeum tools. In *Proceedings of the ACM Sigplan 1999 Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'99)*, volume 34.7 of *ACM Sigplan Notices*, pages 96–103, NY, May 5 1999. ACM Press.

[70] Packet Engines. Gigabit ethernet guide. Technical report, Packet Engines Co, 1997. Also appeared as http://www.packetengines.com/f-gigabiteducation.htm.

[71] Benjamin Falchuk and Ahmed Karmouch. Visual modeling for agent-based applications. *Computer*, 31(12):31–38, December 1998.

[72] D.G. Feitelson, P.F. Corbett, S Jonson Baylor, and Y. Hsu. Parallel I/O subsystems in massively parallel supercomputers. *IEEE Parallel and Distributed Technology*, 1995.

[73] J. T. Feo. An analysis of the computational and parallel complexity of the Livermore loops. *Parallel Computing*, 7(2):163–185, June 1988.

[74] Samuel A. Fineberg, Parkson Wong, Bill Nitzberg, and Chris Kuszmaul. PMPIO—a portable implementation of MPI-IO. In *Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computation*, pages 188–195. IEEE Computer Society Press, October 1996.

[75] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Computers*, C-21(9):948–960, September 1972.

[76] Michael J. Flynn, Patrick Hung, and Kevin W. Rudd. Deep-submicron microprocessor design issues. *IEEE Micro*, 19(4):11–22, July/August 1999.

[77] MPI Forum. MPI: A message-passing interface MPI forum. Technical Report CS/E 94-013, Department of Computer Science, Oregon Graduate Institute, March 1994.

[78] MPI Forum. MPI-2: Extentions to the message-passing interface. Technical Report NO, Department of Computer Science, Oregon Graduate Institute, November 1996.

[79] The MPI Forum. The MPI Message-passing Interface Standard. Technical report, Argonne National Lab, May 1995. Also available as http://www.mcs.anl.gov/mpi/standard.html.

[80] I Foster. *Designing and Building Parallel Programs.* Addison-Wesley, 1995.

[81] G. Fox, W. Furmanski, T. Haupt, E. Akarsu, and H. Ozdemir. HPcc as High Performance Commodity Computing on Top of Integrated Java, CORBA, COM and Web Standards. In *EuroPar'98 Parallel Processing*, volume 1470 of *Lecture Notes in Computer Science*, pages 55–74. Springer Verlag, September 1998.

[82] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine A Users' Guide and Tutorial for Networked Parallel Computing.* The MIT Press ftp ftp.netlib.org/pvm3/book/pvm-book.ps, Sept 1994.

[83] G. A. Geist and V. S. Sunderam. The PVM system: Supercomputer level concurrent computation on a heterogeneous network of workstations. In Quentin F. Stout and Michael Joseph Wolfe, editors, *The Sixth Distributed Memory Computing Conference proceedings April 28-May 1, 1991, Portland, Oregon*, pages 258–261, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1991. IEEE Computer Society Press.

[84] W. Gentzsch. Parallel benchmark results for shared memory systems. *Supercomputer*, 6(4):10–16, July 1989.

[85] L. Geppert. Technology 1998 analysis and forcast. *IEEE Spectrum*, 35(1):19–28, January 1998.

[86] V. Getov, E. Hernandez, and T. Hey. Message-passing performance of parallel computers. *Lecture Notes in Computer Science*, 1300:1009–1016, 1997.

[87] V. Getov, S. Hummel, and S. Mintchev. High-performance parallel programming in java: Exploiting native librarie. In *Proceedings of the 1988 ACM Workshop on Java for High-Performance Network Computing*, February 1998.

[88] M. Gourdreau, K. Lang, S. Rao, T. Suel, and T. Tsantilas. Portable and Efficient Parallel Computing Using the BSP Model. *IEEE Transactions on Computers*, 48(7):670–689, July 1999.

[89] W. Groop and E. Lusk. An abstract device definition to support the implementation of a high-level piont-to-point message-passing interface. Technical report, Algonne National Laboratory, 1995.

[90] W. Groop and E. Lusk. Creating an new MPICH device using the channel interface. Technical report, Algonne National Laboratory, 1995.

[91] W. Groop and E. Lusk. The second-generation ADI for the MPICH implementation of MPI. Technical report, Algonne National Laboratory, 1995.

[92] W. Groop and E. Lusk. The implementation of the second generation MPICH ADI. Technical report, Algonne National Laboratory, 1996.

[93] Bill Gropp and Barry Smith. Chameleon parallel programming tools user's manual. Mathematics and Computer Science Division, Argonne National Laboratory, 1992.

[94] W. Gropp and E. Lusk. Sowing MPICH: A case study in the dissemination of a portable environment for parallel scientific computing. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):103–114, Summer 1997.

[95] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[96] W. Gropp, E. Lusk, and A. Skjelum. *Using MPI Portable Parallel Programming with the Message-Passing Interface*. The MIT Press Books, 1994.

[97] William Gropp. Tutorial on MPI: The Message-passing Interface. Also available as http://mcs.anl.gov/tutorial.html.

[98] H. R. J. Grosch. High speed arithmetic: The digital computer as a research tool. *Journal of the Optical Society of America*, 43(4):306–310, April 1953.

[99] The PSCHED API Working Group. An API for parallel job/resource management version 0.1. Technical report, PSCHED, 1996.

[100] John L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, May 1988.

[101] F. Halsall. *Data Communications, Computer Networks and Open Systems*. Addison Wesley Publishers Ltd., forth edition, 1996.

[102] Steven W. Hammond, Richard D. Loft, and Philip D. Tannenbaum. Architecture and application: The performance of the NEC SX-4 on the NCAR benchmark suite. In ACM, editor, *Supercomputing '96 Conference Proceedings: November 17–22, Pittsburgh, PA*, New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1996. ACM Press and IEEE Computer Society Press.

[103] J. C. Hardwick. Porting a vector library: a comparison of MPI, Paris, CMMD and PVM. In IEEE, editor, *Proceedings of the 1994 Scalable Parallel Libraries Conference: October 12–14, 1994, Mississippi State University, Mississippi*, pages 68–77, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1995. IEEE Computer Society Press.

[104] G. Haring, P. Kacsuk, and G. Kotsis. Distributed and parallel systems: Environments and tools. *Parallel Computing*, 22(13):1699–1711, April 1997.

[105] K. A. Hawick, D. A. Grove, and F. A. Vaughan. Beowulf - A New Hope for Parallel Computing? In *Proc. of the 6th IDEA Workshop, Rutherglen*, January 1999. Also available as DHPC Technical Report DHPC-061.

[106] John Hennessy. The Future of Systems Research. *Computer*, 32(8):27–33, August 1999.

[107] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantative Approach.* Morgan Kaufman, San Mateo, California, 1990.

[108] A.J.G. Hey. The MPI standard a progress report. Technical report, University of Southampton, 1996.

[109] Nicholas J. Higham. Exploiting fast matrix multiplication within the level 3 BLAS. *ACM Transactions on Mathematical Software*, 16(4):352–368, December 1990.

[110] M. B. Hill. *Software Environments for Parallel Computing.* PhD thesis, University of Southampton, Electronic and Computer Science Department, April 1993.

[111] G. Hipper and D. Tavangarian. A new architecture for efficient parallel computing in workstation clusters: Conceptions and experiences. Technical report, Univesitat Rostock, 1996.

[112] R. Hockauf, W. Karl, M. Leberecht, M. Oberhuber, and M. Wagner. Exploiting Spatial and Temporal Locality of Accesses: A New Hardware-Based Monitoring Approach for DSM Systems. In *EuroPar'98 Parallel Processing*, volume 1470 of *Lecture Notes in Computer Science*, pages 206–215. Springer Verlag, September 1998.

[113] R. Hockney. *Portability and Performance of Parallel Processing*, chapter Performance Parameters and Results for the Genesis Parallel Benchmarks, pages 209–222. John Wiley & Sons Ltd, 1994. Editors: T. Hey and J. Ferrante.

[114] R. Hockney. *The Science of Computer Benchmarking.* SIAM, 1996.

[115] High Performance Computing and Communications Foundation of America's Information Future, 1996. Also available as http://www.ccic.gov/pubs/blue96/index.html.

[116] IBM. Sp2. Available as http://www.ibm.com.

[117] Myricom Inc. Myrinet: A brief, technical overview. Technical report, Myricom Inc, 1996. http://www.myri.com.

[118] Intel Co. *Using the RDTSC Instruction for Performance Monitoring*, 1999.

[119] D. James, D. Gustavson, and B. Fleischer. SerialExpress a High-Performance Workstation Interconnect. *Micro*, 2(5):54–65, May 1998.

[120] Rakesh Jha, Richard C. Metzger, Brian VanVoorst, Luiz S. Pires, Wing Au, Minesh Amin, David A. Castanon, and Vipin Kumar. The C3I parallel benchmark suite — introduction and preliminary results. In ACM, editor, *Supercomputing '96 Conference Proceedings: November 17–22, Pittsburgh, PA*, New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1996. ACM Press and IEEE Computer Society Press.

[121] Young Moo Kang, Robert B. Miller, and Roger Alan Pick. Comments on "Grosch's law re-revisited: CPU power and the cost of computation". *Communications of the ACM*, 29(8):779–781, August 1986.

[122] Vijay Karamcheti and Andrew A. Chien. Software overhead in messaging layers: Where does the time go? In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 51–60, San Jose, California, October 4–7, 1994. ACM SIGARCH, SIGOPS, SIGPLAN, and the IEEE Computer Society.

[123] Krishna Kavi, James Browne, and Anand Tripathi. Computer systems research: The pressure is on. *Computer*, 32(1):30–39, January 1999.

[124] G. M. King, D. M. Dias, and P. S. Yu. Cluster architectures and S/390 Parallel Sysplex scalability. *IBM Systems Journal*, 36(2):221–235, 1997.

[125] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing Design and Analysis of Algorithms.* The Benjamin/Cummings Publishing Company, Inc, 1994.

[126] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms.* Benjamin/Cummings, Redwood City, CA, 1994.

[127] K. Lai and M. Baker. A performance comparison of unix operating systems on the pentium. In *1996 USENIX Technical Conference*, January 1996.

[128] L. Lamport. A new solution of dijkstra's concurrent programming program. *Communications of the ACM*, 17(8), August 1974.

[129] L. Lamport. Time, clocks, and the ordering of events in a distributed system. In *Communications of the ACM.* ACM Press, July 1978.

[130] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEETC*, C-28(9):690–691, September 1979.

[131] Leslie Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, November 1977.

[132] Brian LaRose. The development and implementation of a performance database server. Master's thesis, University of Tennessee Computer Science Department, http://netlib2.cs.utk.edu/tennessee/ut-cs-93-195.ps, Aug 1993.

[133] M. Lauria. High performance mpi implementation on a network of workstations. Master's thesis, University of Illinois at Urbana-Champaign, 1996.

[134] Mario Lauria and Andrew Chien. MPI-FM: High performance MPI on workstation clusters. *Journal of Parallel and Distributed Computing*, 40(1):4–18, January 1997.

[135] Ted Lewis. Mainframes are Dead, Long Live Mainframes. *Computer*, 32(8):102–104, August 1999.

[136] Ted Lewis and Hesham El-Resini with In-Kyu Kim. *Introduction to parallel computing.* Prentice Hall, 1992.

[137] X. Li, P. Lu, J. Schaeffer, J. Shillington, P. S. Wong, and H. Shi. On the versatility of parallel sorting by regular sampling. *Parallel Computing*, 19:1079–1103, October 1993.

[138] M. Lin, J. Hsieh, D. H. C. Du, and J. A. MacDonald. Performance of high-speed network I/O subsystems for: Case study of a fibre channel network. In IEEE, editor, *Proceedings, Supercomputing '94: Washington, DC, November 14–18, 1994*, Supercomputing, pages 174–183, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1994. IEEE Computer Society Press.

[139] L. Liu, A. Mainwaring, and C. Yoshikawa. White paper on building TCP/IP active messages. November 94.

[140] H. Lu, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Message-Passing vs. Distributed Shared Memory on Networks of Workstations. In *Proc. of Supercomputing'95*, December 1995. Avialable online: <http://www.cs.rice.edu/~willy/pa pers/sc95.ps.gz>.

[141] Yong Luo. Shared memory vs. message passing: The COMOPS benchmark experiment. *The Journal of Supercomputing*, 13(3):283–301, May 1999.

[142] Maccabe. *Parallel Computing: Theory and Practice.* McGraw-Hill, 1994.

[143] B. E. Martin, C. H. Pedersen, and J. Bedford-Roberts. An object-based taxonomy for distributed computing systems. *Computer*, 24(8):17–27, August 1991.

[144] Richard P. Martin, Amin M. Vahdat, David E. Culler, and Thomas E. Anderson. Effects of communication latency, overhead, and bandwidth in a cluster architecture. In *Proceedings of the 24rd Annual International Symposium on Computer Architecture*, pages 85–97, Denver, Colorado, June 2–4, 1997. ACM SIGARCH and IEEE Computer Society TCCA.

[145] F. Mattern and S. Fuenfrocken. A non-blocking lightweight implementation of causal order message delivery. *Lecture Notes in Computer Science*, 938:197–205, 1995.

[146] Timothy G. Mattson. Programming environments for parallel and distributed computing: A comparison of P4, PVM, Linda, and TCGMSG. *International Journal of Supercomputer Applications and High Performance Computing*, 9(2):138–161, Summer 1995.

[147] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Technical Committee on Computer Architecture newsletter*, December 1995.

[148] Frank H. McMahon. The livermore kernels: a computer test of the numerical performance range. Technical Report UCRL-53745, LLNL, Livermore, CA, December 1986.

[149] Eugene S. Meieran. 21st century semiconductor manufacturing capabilities. *Intel Technology Journal*, 4th Quarter '98, 1998.

[150] P. Melas, M. Dantas, and E. Zaluska. Efficient Communication of MPI Applications on Networks of Workstations. In *Proceedings of the 15th IASTED International Conference*, pages 141–144. IASTED, February 1997.

[151] P. Melas and E. J. Zaluska. High Performance Protocols for Clusters of Commodity Workstations. In *EuroPar'98 Parallel Processing*, volume 1470 of *Lecture Notes in Computer Science*, pages 570–577. Springer Verlag, September 1998.

[152] P. Melas and E. J. Zaluska. Performance of Message-Passing Systems Using a Zero-Copy Communication Protocol. In *Parallel Architectures and Compilation Techniques*, volume 1, pages 264–271. IEEE Computer Society, October 1998.

[153] Phil Merkey. Beowulf Project at CESDIS NASA. http://www.beowulf.org, 1999.

[154] Joerg Meyer. Message passing interface for Microsoft Windows 3.1. Master's thesis, Department of Computer Science, University of Nebraska, December 1994.

[155] Sun Microsystems. *TCP/IP and Data Communications Guide*. Solaris 2.5, sunSoft edition, November 1995.

[156] S. Mintchev and V. Getov. Towards portable message passing in Java: Binding MPI. *Lecture Notes in Computer Science*, 1332:135–142, 1997.

[157] K. Morimoto, T. Matsumoto, and K. Hiraki. Performance evaluation of the MPI/MBCF with the NAS parallel benchmarks. In J. J. Dongarra, E. Luque, and Tomas Margalef, editors, *Recent advances in parallel virtual machine and message passing interface: 6th European PVM/MPI Users' Group Meeting, Barcelona, Spain, September 26–29, 1999: proceedings*, volume 1697 of *Lecture Notes in Computer Science*, pages 19–26, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1999. Springer-Verlag.

[158] D. Mosberger and L. Peterson. Careful protocols or how to use highly reliable networks. In *4th Workshop on Workstation Operating Systems (WWOS-IV)*, Napa, U.S., 1993.

[159] David Mosberger. Linux and the Alpha. *Linux Journal*, 42, October 1997.

[160] Shubhendu S. Mukherjee and Mark D. Hill. A survey of user-level network interfaces for system area networks. Technical Report CS-TR-97-1340, University of Wisconsin, Madison, February 1997.

[161] Shubhendu S. Mukherjee and Mark D. Hill. Research Feature: Making Network Interfaces less Peripheral. *Computer*, 31(10):70–76, October 1998.

[162] Sape J. Mullender. *Distributed Systems.* ACM Press, 1990.

[163] O. Naim, A. Hey, and E. Zaluska. Do-loop-surface: an abstract representation of parallel program performance. *Concurrency: Practice and Experience*, 8(3):205–234, April 1996.

[164] C. Catlett NASA. Parallel I/O: Getting ready for prime time. International Conference on Parallel Processing, August 1994.

[165] N. Nevin. The performance of LAM 6.0 and MPICH 1.0.12 on a workstation cluster. Technical Report OSC-TR-1996-4, Ohio Supercomputer Center, 1996.

[166] J. M. Nick, B. B. Moore, J.-Y. Chung, and N. S. Bowen. S/390 cluster technology: Parallel Sysplex. *IBM Systems Journal*, 36(2):172–??, 1997.

[167] Natawut Nupairoj and Lionel M. Ni. Benchmarking of multicast communication services. Technical Report MSU-CPS-ACS-103, Department of Computer Science, Michigan State University, 1995.

[168] P.S. Pacheco. *Parallel Programming with MPI.* Morgan Kaufmann Publishers Inc., 1997.

[169] S. Pakin, V. Karamcheti, and A. Chien. Fast messages (fm): Efficient, portable communication for workstation clusters and massively-parallel processors. Department of Computer Science University of Illinois To appear in IEEE Concurrency.

[170] S. Pakin, M. Lauria, and A. Chien. High performance messaging on workstations: Illinois fast messages (fm) for myrinet. In *In Supercomputing '95*, San Diego, California, 1995.

[171] Dhabaleswar K. Panda and Lionel M. Ni. Special Issue on Workstation Clusters and Network-Based Computing: Guest Editors' introduction. *Journal of Parallel and Distributed Computing*, 40(1):1–3, January 1997.

[172] C. Papadopoulos and G. M. Parulkar. Experimental evaluation of sunOS IPC and TPC/IP protocol implementation. In Michael G. Hluchyj, editor, *Proceedings of the 12th Annual Joint Conference of the IEEE Computer and Communications Societies on Networking: Foundations for the Future. Volume 2*, pages 628–637, Los Alamitos, CA, USA, March 1993. IEEE Computer Society Press.

[173] White Paper. Improving network performance with ethernet switching. Technical report, Hewlett-Packard, 1993.

[174] Parasoft Corporation. *Express User's Manual*, 1989.

[175] J. Pasquale, E. Anderson, K. Fall, and J. Kay. ihigh Performance I/O and Networking Software in Sequoia 2000. *Digital Technical Journal*, 7(3):84–94, 1995.

[176] David A. Patterson and John L. Hennessy. *Computer Architecture A Quantitative Approach.* Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, second edition, 1996.

[177] David A. Patterson and John L. Hennessy. *Computer Organization: The Hardware/ Software Interface.* Morgan Kaufmann Publishers, 2929 Campus Drive, Suite 260, San Mateo, CA 94403, USA, second edition, 1997.

[178] G. F. Pfister. *In Search of Clusters, 1/e.* Prentic Hall International Inc., second edition, Jan 1998.

[179] J. Postel. Internet Protocol. *Network Information Center RFC 791*, pages 1–45, September 1981.

[180] J. Postel. Transmission Control Protocol. *Network Information Center RFC 793*, pages 1–85, September 1981.

[181] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C.* Cambridge University Press, New York, NY, 1991.

[182] Loïc Prylli. Draft: BIP messages user manual for BIP 0.92. Technical report, Laboratoire de l'Informatique du Parallelisme Lyon, 1997.

[183] Loïc Prylli and Bernard Tourancheau. Bpi: A new protocol design for high performance networking on myrinet. Technical report, LIP-ENS Lyon, September 1997.

[184] Loïc Prylli and Bernard Tourancheau. New protocol design for high performance networking. Technical report, LIP-ENS Lyon, July 1997.

[185] M. Quinn. *Parallel Computing: Theory and Practice.* McGraw-Hill, 1994.

[186] PARKBENCH Committee: Report-1, assembled by Roger Hockney (chairman), and Michael Berry secretary). Public international benchmarks for parallel computers. Technical Report UT-CS-93-213, Department of Computer Sci/ence, University of Tennessee, November 1993.

[187] Steven H. Rodrigues, Thomas E. Anderson, and David E. Culler. High-performance local-area communication with fast sockets. In USENIX, editor, *1997 Annual Technical Conference, January 6–10, 1997. Anaheim, CA*, pages 257–274, Berkeley, CA, USA, January 1997. USENIX.

[188] W. Rosenberry, D. Kenney, and G. Fisher. *OSF Distributed Computing Environment: Understanding DCE.* O'Reilly and Associates, Inc, 103 Morris Street, Suite A, Sebastopol, Ca 95472, 1993.

[189] David E. Ruddock and Balakrishnan Dasarathy. Multithreading programs: Guidelines for DCE applications. *IEEE Software*, 13(1):80–90, January 1996.

[190] David A Rusling. *The Linux Kernel*. Linux Document Project, 0.8-3 edition, January 1999. Available as http://www.linuxdoc.org/LDP/tlk/tlk-title.html.

[191] Sartaj Sahni and Venkat Thanvantri. Performance metrics: Keeping the focus on runtime. *IEEE parallel and distributed technology: systems and applications*, 4(1):43–56, Spring 1996.

[192] J. Salmon, C. Stein, and T. Sterling. Scaling of Beowulf-class Distributed Systems. In ACM, editor, *SC'98: High Performance Networking and Computing: Proceedings of the 1998 ACM/IEEE SC98 Conference: Orange County Convention Center, Orlando, Florida, USA, November 7–13, 1998*, New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1998. ACM Press and IEEE Computer Society Press.

[193] R. Sandberg. The sun network filesystem: Design, implementation, and experience. In Akkihebbal L. Ananda and Balasubramaniam Srinivasan, editors, *Distributed Computing Systems: Concepts and Structures*, pages 300–316. IEEE Computer Society Press, Los Alamos, CA, 1992.

[194] Russel Sandberg. The design and implementation of the Sun network file system. In *Proceedings of the USENIX Summer Conference*, pages 119–130, Berkeley, CA, USA, June 1985. USENIX Association.

[195] Michael Santifaller. *TCP/IP and ONC/NFS*. Addison-Wesley, Reading, MA, USA, second edition, 1994.

[196] B. Saphir, P. Bozemen, R. Evard, and P. Beckman. Production Linux Clusters, the tribble project. In Supercomputing 99, editor, *SC 99 Tutorial*. Supercomputing, November 1999. Also available as http://www.nersc.gov/research/tribble.

[197] I. Schoinas and M. Hill. Address Translation Mechanisms in Network Interfaces. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture (HPCA)*, pages 219–230, February 1998.

[198] C. L. Seitz, N. J. Boden, J. Seizovic, and W. Su. The design of the caltech mosaic c multicomputer. In *Proceedings of the University of Washington Symposium of Integrated Systems*, pages 1–22. MIT Press, 1993.

[199] Charles L. Seitz. The caltech mosaic C: an experimental, fine-grain multicomputer. In *Proceedings of the 4th Annual Symposium on Parallel Algorithms and Architectures*, pages 1–2, San Diego, CA, USA, June 1992. ACM Press.

[200] Robert D. Silverman. Exposing the Mythical MIPS Year. *Computer*, 32(8):22–26, August 1999.

[201] D. Sima, T. Fountain, and P. Kacsuk. *Advanced Computer Architectures a Design Space Approach.* Addison-Weslay Longman Ltd, Essex, England, 1997.

[202] A. Skjellum, S. G. Smith, N. E. Doss, A. P. Leung, and M. Morari. The design and evolution of Zipcode. *Parallel Computing*, 20(4):565–596, April 1994.

[203] Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI: the complete reference.* MIT Press, Cambridge, MA, USA, 1996.

[204] SPEC. The Standard Performance Evaluation Corporation: Better Benchmarks, June 1999. Available as http://www.SPEC.org/spec.

[205] Carl Staelin, Larry McVoy, and BitMover Inc. mhz: Anatomy of a micro-benchmark. In *Proceedings of the USENIX 1998 Annual Technical Conference*, pages 155–166, Berkeley, USA, June 15–19 1998. USENIX Association.

[206] W. Stallings. *Operating Systems.* Prentice Hall International inc., second edition edition, 1995.

[207] T. Sterling, J. Salmon, D. Becker, and D. Savarese. *How to Build a Beowulf a guide to the implementation and application of PC clusters.* Number ISBN 0-262-69218-X in Scientific and Engineering Computation Series. The MIP Press, January 1999.

[208] T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer. BEOWULF : A parallel workstation for scientific computation. In *International Conference on Parallel Processing, Vol.1: Architecture*, pages 11–14, Boca Raton, USA, August 1995. CRC Press.

[209] W. R. Stevens. *TCP/IP Illustrated, Volume 1; The Protocols.* Addison Wesley, Reading, 1995.

[210] I. Stoica, F. Sultan, and D. Keyes. Evaluating the hyperbolic model on a variety of architectures. Technical report, Department of Computer Science Old Dominion University, 1996.

[211] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.

[212] V. Strumpen. Communication latency hiding – model and implementation in high-latency computer networks. Technical Report 1994TR-216, Swiss Federal Institute of Technology, Zurich, June, 1994.

[213] Volker Strumpen. The network machine. Technical Report 1995DI-th11227, Swiss Federal Institute of Technology, Zurich, July, 1995.

[214] Xian-He Sun and Jianping Zhu. Performance considerations of shared virtual memory machines. *IEEE Transactions on Parallel and Distributed Systems*, 6(11):1185–1194, November 1995. Also available as http://www.computer.org/tpds/td1995/l1185abs.htm.

[215] Sun Microsystems, Inc. RFC 1057: RPC: Remote procedure call protocol specification: Version 2, June 1988. Obsoletes RFC1050. Status: INFORMATIONAL.

[216] Sun Microsystems, Inc. RFC 1094: NFS: Network File System Protocol specification, March 1989.

[217] SunSoft. *SunOS 5.3 Network Interfaces Programmer's Guide*. Sun Microsystem, revision a edition, Nov 1993.

[218] Meiko Computing Surface. Computing sruface cs-2. Technical report, Meiko Scientific Corporation, 1993.

[219] Mark R. Swanson and Leigh B. Stoller. Low Latency Workstation Cluster Communications Using Sender-Based Protocols. Technical Report UUCS-96-001, University of Utah, Department of Computer Science, January 2, 1998.

[220] Ted Tabe and Quentin F. Stout. The use of the MPI communication library in the NAS parallel benchmarks. Technical Report CSE-TR-386-99, University of Michigan Department of Electrical Engineering and Computer Science, February 17, 1999.

[221] Andrew S. Tanenbaum. *Distributed operating systems*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1995.

[222] Andrew S. Tanenbaum. *Computer Networks*. Prentice-hall International, Inc., 3rd edition, 1996. ISBN: 0-13-394248-1
http://www.cs.vu.nl/ ast/ph/cn3.html.

[223] Andrew S. Tanenbaum and Robbert van Renesse. Distributed operating systems. *ACM Computing Surveys*, 17(4):419–470, December 1985.

[224] F.J. Valente. *An Integrated Parallel/Distributed Environment for Hight Performance Computing*. PhD thesis, University of Southampton, December 1995.

[225] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103, August 1990.

[226] A. J. van der Steen. The benchmark of the EuroBen group. *Parallel Computing*, 17(10-11):1211–1221, December 1991.

[227] Adrianus Jan van der Steen. *Benchmarking of High Performance Computers for Scientific and Technical Computation*. PhD thesis, ACCU, Utrecht, Netherlands, March 1997.

[228] Thorsten von Eicken. *Active Messages: an Efficient Communication Architecture for Multiprocessors*. PhD thesis, University of California at Berkeley, November 1993.

[229] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-net: a user-level network interface for parallel and distributed computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, volume 29, pages 303–316, 1995.

[230] Jack C. M. Wang, John M. Gary, and Hari K. Iyer. A technique to evaluate benchmarks: A case study using the Livermore loops. *The International Journal of Supercomputer Applications*, 4(4):40–55, Winter 1990.

[231] Thomas M. WARSCHKO, Joachim M. BLUM, and Walter F. TICHY. The paraPC/parastation project: efficient parallel computing by clustering workstations. Technical Report iratr-1996-13, Universität Karlsruhe, Institut für Programmstrukturen und Datenorganisation, 1996.

[232] Thomas M. WARSCHKO, Walter F. TICHY, and Christian G. HERTER. Efficient parallel computation on workstation clusters. Technical Report iratr-1995-21, Universität Karlsruhe, Institut für Programmstrukturen und Datenorganisation, 1995.

[233] R. P. Weicker. Dhrystone: A synthetic systems programming benchmark. *Communications of the ACM*, 27(10):1013–1030, October 1984.

[234] M. Welsh, A. Basu, and T. von Eicken. Low-latency communication over fast ethernet. In *Proc. EUROPAR '96*, August 1996.

[235] Matt Welsh, Anindya Basu, Xun Wilson Huang, and Thorsten von Eicken. Memory Management for User-Level Network Interfaces. *IEEE Micro*, 18(2):77–92, March/April 1998.

[236] Matt Welsh, Anindya Basu, and Thorsten von Eicken. ATM and Fast Ethernet network interface for user-level communication. In *Proceedings of the Third International Symposium on high Performance Computer Architecture*, pages 332–342, 1997.

[237] S Wilbur and B Bacarisse. Building distributed systems with remote procedure call. *IEE Software Engineering Journal*, 2(5):148–159, September 1987.

[238] M. Wilkes. CMOS Workstations and Servers — How Far Can Evolution and Innovation Take Us? In *Parallel Architectures and Compilation Techniques*, volume 1. IEEE Computer Society, October 1998.

[239] Wilson,K.G. Grand challenges to computational science. In *Proc. IEEE CS Intl. Conf. No. 6 on Data Engineering*, February 1990.

[240] Frederick C. Wong, Richard P. Martin, Remzi H. Arpaci-Dusseau, and David E. Culler. Architectural requirements and scalability of the NAS Parallel Benchmarks. In ACM, editor, *SC'99: Oregon Convention Center 777 NE Martin Luther King Jr. Boulevard, Portland, Oregon, November 11–18, 1999*, New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1999. ACM Press and IEEE Computer Society Press.

[241] P. R. Woodward. Prespectives on supercomputing: Three decades of change. *Computer*, pages 99–111, October 1996.

[242] Paul R. Woodward. Perspective on supercomputing: Three decades of change. *Computer*, 29(10):99–111, October 1996.

[243] K. Yocum, D. Anderson, J. Chase, S. Gadde, A. Gallatin, and A. Lebeck. Balancing DMA Latency and Bandwidth in a High-Speed Network Adapter. Technical Report CS–1997–20, Department of Computer Science, Duke University, Nov 1997.

Appendix A

The 802.3 MAC Sublayer

The calculation of the maximum theoretical bandwidth of the Ethernet channel requires an analysis of the data format field. Each frame of the 10 Mbit/s baseband standard starts with a 7 byte *preamble* field followed by a *start of frame* byte (see Figure A.1). Then the fields of *destination* and *source* address follow with 6 bytes each. Then the length of data field follows and the variable *data* field with 0-1500 bytes. The *pad* field has a variable length 0-46 bytes in order to guarantee the minimum frame length of 64 bytes. The last field of the frame is the *checksum* field of 4 bytes [222]. The minimum inter-packet gap required is 9.6 microseconds which corresponds to 12 bytes at 10 Mbit/s [209].

Thus the minimum Ethernet raw packet overhead from Figure A.1 and Table A.1 is 38 bytes with a payload up to 1500 bytes. For a 10Mbit/s channel the maximum theoretical throughput is:

$$\frac{1500}{1500+38} \times \frac{10^7 bit/s}{8bit/Byte} = 1.219 MByte/sec \tag{A.1}$$

For an Ethernet frame that carries a TCP/IP packet there is an extra overhead of 40 bytes due to TCP and IP headers (IP fragmentation is excluded [195, 209]), consequently the payload is reduced to 1460 bytes. The maximum theoretical throughput of a TCP/IP packet over a 10Mbit/s Ethernet without the corresponding ACK is:

$$\frac{1460}{1500+38} \times \frac{10^7 bit/s}{8bit/Byte} = 1.186 MByte/sec \tag{A.2}$$

If the acknowledgment packet (ACK), which the receiver has to send back to the sender, is included the throughput drops slightly down to 1.125 Mbyte/s. In practice this will be the

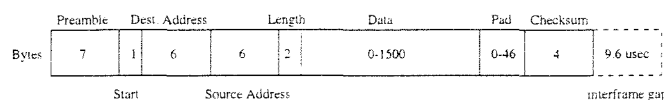

Figure A.1: The 802.3 frame format

Field	**Data** (bytes)	**ACK** (bytes)
Preamble field	8	8
Destination addr.	6	6
Source address	6	6
Length of data	2	2
IP header	20	20
TCP header	20	20
User data	0-1460	0
PAD field	0-6	6
CRC field	4	4
Inter-packet gap	12	12
	1538/84	84

Table A.1: Ethernet frame field sizes

Configuration	$n_{1/2}$
raw Ethernet	36
Ethernet/TCP/IP	70
Ethernet/TCP/IP, ACK	132
Experimental result (10Base)	>170
Experimental result (100Base)	

Table A.2: $n_{1/2}$ size for each protocol layer

worst case as the TCP/IP at the receiver end could acknowledge more than one frame within a single ACK reply.

$$y(x) = \begin{cases} \frac{5x}{336} & if\ x \leq 6 \\ \frac{5x}{312+4x} & if\ 7 \leq x \leq 1460 \end{cases} \tag{A.3}$$

Similarly the performance of a Fast-Ethernet network according to equations A.1 and A.2 is 12.19Mbyte/sec and 11.86Mbyte/sec respectively.

$$y(x) = \begin{cases} \frac{25x}{188} & if\ x \leq 6 \\ \frac{25x}{186+2x} & if\ 7 \leq x \leq 1460 \end{cases} \tag{A.4}$$

The Header of the IP and the TCP Layer Overheads Figure A.2 illustrates graphically the impact of the extra overhead on the throughput (performance equation A.3). A simple calculation of the $n_{1/2}$ (half throughput performance) gives a 36 byte message size for the raw Ethernet, a 70 byte message size for TCP/IP over an Ethernet frame, and a 132 byte message size for TCP/IP over an Ethernet frame including an ACK.

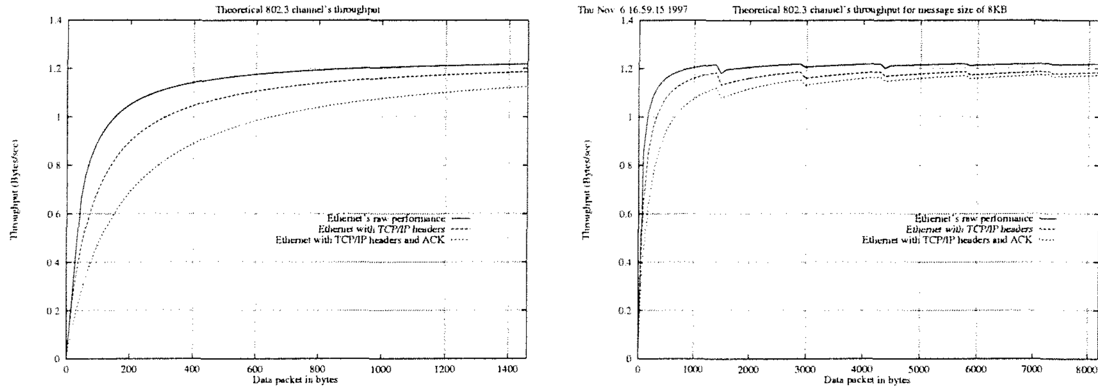

Figure A.2: Theoretical performance of an Ethernet packet

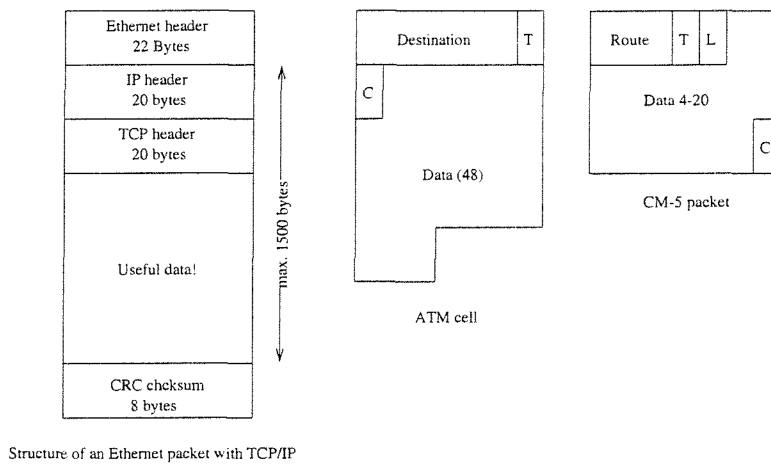

Figure A.3: The structure of an Ethernet packet with TCP/IP header overhead, an ATM cell and a proprietary network packet of CM-5

From plots of Fig A.2 we can see that for large messages the TCP/IP overhead and the ACK tend to merge and improve the throughput performance as the overhead and the ACK become a smaller fraction of the payload.

Appendix B

Processor Timing Mechanisms

B.1 Pentium Time Stamp Counter

The following code fragment has been written to measure the time take to perform a floating-point division:

```
/* read Pentium time stamp counter, 64 bits */
void rdtsc(val)
int val[];
{
   asm("rdtsc");              /* read time stamp to EAX */
   asm("movl 8(%ebp),%ecx"); /* move eax = val          */
   asm("movl %eax,0(%ecx)"); /* move low 32 bits       */
   asm("movl %edx,4(%ecx)"); /* move hi 32 bits        */
}
```

B.2 DEC Alpha timer

DEC Alpha processors provide time stamp counters similar to Intel Pentium. The process cycle counter (PCC) is an unsigned 32-bit integer that increments once per N CPU cycles, where N is an implementation specific integer in the range 1..16. The high-order 32 bits of the process cycles counter are an offset that when added to the low-order 32 bits gives the cycles count for this process. The process cycle counter is suitable for timing intervals on the order of nanoseconds and may be used for detailed performance characterization. Special-Purpose Instructions (rpcc d_reg) are used to read the content of this counter.

```
static inline __u32 rpcc(void) /* */
{
    __u32 result;

asm volatile ("rpcc %0" : "r="(result));
```

```
    return result;
}
```

B.3 Ultra-SPARC TICK Register

Ultra-Sparc I-III architecture provides several processor performance monitoring registers such as the Performance Control Register(PCR), Performance Instrumentation Counters(PICs) as well as a TICK register which is incremented once per machine cycle. (SPARC-V9)

```
rd %%tick, %0
```

B.4 RS6000 Tick Register

The RS6000 architecture provides 64-bit time-stamp registers.

```
long long Readtimer()
{
   register long hi;
   register long lo;

   asm volatile("mftb %1\n" "mftbu %0":
                           "=r(hi)", "=r"(lo):);
   return ((((long long)hi)<<32) | lo);
}
unsigned tstart, tend;
double duration;
asm("mftb %0": "=r" (tstart): ); /* perform the operation */
asm("mftb %0": "=r" (tend): );
duration= (tend-tstart)*(BUS_PERIOD*4.0)
```

B.5 CPU Speed

A simple timer and a machine cycle counter are sufficient to calculate the speed of a processor. The following code is an example which calculates the speed of a Pentium processor.

```
#include <stdio.h>
#include <time.h>

void main ()
{
   volatile unsigned long long int x, x1;
   int dt=5;
   time_t t;
```

```
    printf("This program runs only on Pentium CPU\n");
    printf("Please wait, this program takes about 6 sec
                                            to run...\n");
    time(&t);
    while( t==time(NULL) );
    t = time(NULL);

    __asm__ volatile("rdtsc" : "=A" (x));
    while( dt+t>time(NULL) );
    __asm__ volatile("rdtsc" : "=A" (x1));

    printf( "Your Pentium/PentiumPro's clock
                                rate is %g Mhz\n",
      ((double)(x1-x))/((double)1000000)/((double)dt));
}
```

Appendix C

Case Study: Matrix Multiplication

In scientific applications matrix-matrix multiplication is often the most power-demanding part of an application code. This implementation of the Strassen matrix multiplication algorithm [211, 181] does not claim to be the fastest one possible or to introduce a new parallel programming style, because the objective was restricted to use it as a test-bed for MPI performance evaluation on heterogeneous and homogeneous clusters. Its implementation can combine point-to-point communications and collective communications at the same time as well as a possibility for partial overlapping between communication and computation.

Strassen's Algorithm

A single multiplication of two $n \times n$ matrices requires n^3 multiplications and $n^2(n-1)$ additions resulting in $O(n^3)$ complexity [109].

$$C_{(i,j)} = \sum_{k=1}^{N} A_{(i,k)} \times B_{(k,j)} \tag{C.1}$$

Strassen's algorithm can be applied to any $n \times n$ matrix where $n = 2m$. The original matrices (A, B) can be partitioned into four $m \times m$ sub-matrices, so the algorithm proceeds as follows:

$$\begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \cdot \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix} = \begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} \tag{C.2}$$

then the calculation of the partial sub-matrices is:

$$P_0 \equiv (a_{00} + a_{11}) \times (b_{00} + b_{11}) \tag{C.3}$$

$$P_1 \equiv (a_{10} + a_{11}) \times b_{00} \tag{C.4}$$

$$P_2 \equiv a_{00} \times (b_{01} - b_{11}) \tag{C.5}$$

$$P_3 \equiv a_{11} \times (-b_{00} + b_{10}) \tag{C.6}$$

$$P_4 \equiv (a_{00} + a_{01}) \times b_{11} \quad (C.7)$$

$$P_5 \equiv (-a_{11} + a_{10}) \times (b_{00} + b_{01}) \quad (C.8)$$

$$P_6 \equiv (a_{10} - a_{11}) \times (b_{10} + b_{11}) \quad (C.9)$$

The final matrix C can be obtained as a single sum of Ps:

$$c_{00} = P_0 + P_3 - P_4 + P_6 \quad (C.10)$$

$$c_{01} = P_2 + P_4 \quad (C.11)$$

$$c_{10} = P_1 + P_3 \quad (C.12)$$

$$c_{11} = P_0 + P_2 - P_1 + P_5 \quad (C.13)$$

The algorithm uses 7 multiplications instead of 8 which the classical algorithm requires, and 14 extra summations. The number of multiplications of Strassen's algorithm is $n^{log_2 7}$. For large n, there is a net saving of time because more time is saved with the smaller number of multiplications than required for the extra additions. For example if n=1024 the complexity ratio between the classical algorithm and this algorithm is around *3.8* (for n=2048 the ratio is 4.4).

C.1 The Implementation

The MPI program runs on 7 processes and each process calculates a partial sub-matrix of the product. The size of the matrices is variable with matrix sizes up to 1024 X 1024 used for the tests. In the beginning, the two matrices which have to be multiplied are generated on processes 0 and 2. In the next stage sub-matrices A_{sub} and B_{sub} are distributed to other processes. In the first MPI-session processes use MPI *send* and *receive* functions to distribute the sub-matrices, as shown in Figure C.1.

The next step calculates the partial product for each process. The partial product of each process has now to be exchanged among 4 processes which finally calculate the 4 sub-matrices of the product matrix (the second MPI-session).

Note The sequential and the MPI implementation make use of the same matrix or sub-matrix multiplication function. This function has undergone minor optimisations for speed.

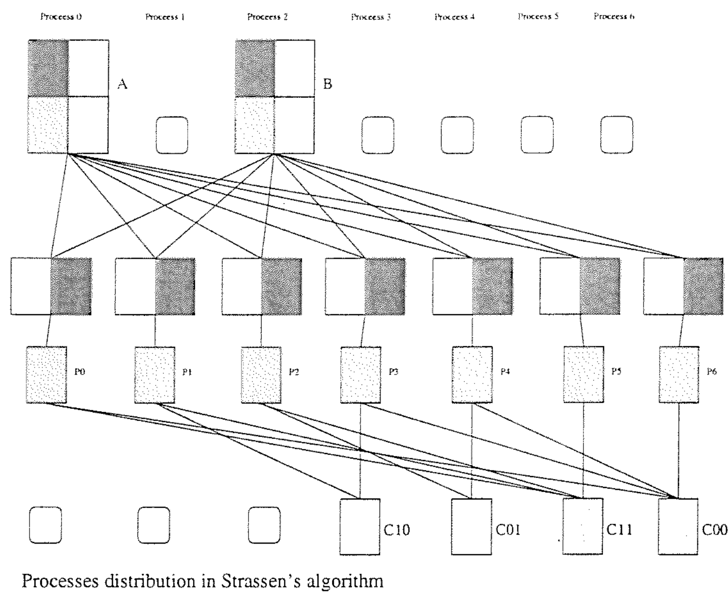

Figure C.1: Data flow among processes

C.2 The Environment

The application runs on both homogeneous and heterogeneous clusters (see Table C.1). During the tests there was no need to modify its code at all demonstrating the portability of the MPI implementations. The network used was an open one and the (uncontrolled) workload of other users on the nodes is taken as constant throughout the tests. The results were measured during off-peak hours to minimise the interference of other users.

The MPICH version used was:

```
MPI model implementation 1.00.12., ADI version 1.30-transport p4
Configured with -arch=solaris -device=ch_p4 -mpe -nof77
```

As can be seen from Table C.1 and Figure C.2 the network of workstations used is not homogeneous. Some clusters are located in different buildings and some clusters use either hubs or Ethernet switches as well. The Solaris and IRIX workstations clusters are based on standard Ethernet segments, while the departmental network is based on Ethernet hubs and Ethernet switches. In this last network configuration, careful consideration of the process distribution among the workstations could avoid potential network bottlenecks.

C.3 Results, Analysis and Comparisons

The sequential matrix multiplication program was run on different nodes to provide some measurements in order to compare and evaluate the parallel implementation. Because the cluster is heterogeneous results are affected mostly by the node architecture e.g. CPU architecture,

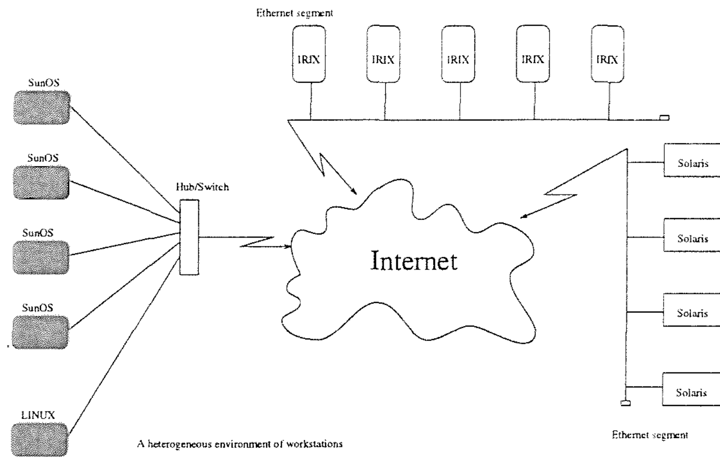

Figure C.2: The heterogeneous network architecture used

Table C.1: The heterogeneity of the cluster

WORKSTATION	CPU	MHz	RAM	SYSTEM	OS	Location
caesar	SPARC	25	56	Sun 300	SunOS	B 16
daedalus	Pentium	75	16	PC	Linux	B 16
ringwood	SPARC	20	32	Sun 4/60	SunOS	B 16
taranaki	SPARC	80	48	Sun SPARC2	SunOS	B 16
euclid	SPARC	25	56	Sun 300	SunOS	B 16
b25d-xx	R4000	100	48	SGI	IRIX	B 25
b25d-xx	SPARC	70	32	Sun SPARC5	SOLARIS	B 25

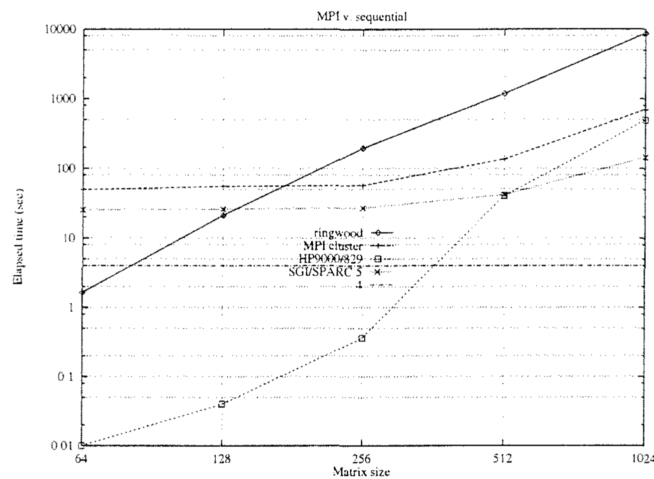

Figure C.3: Elapsed time between sequential and parallel code

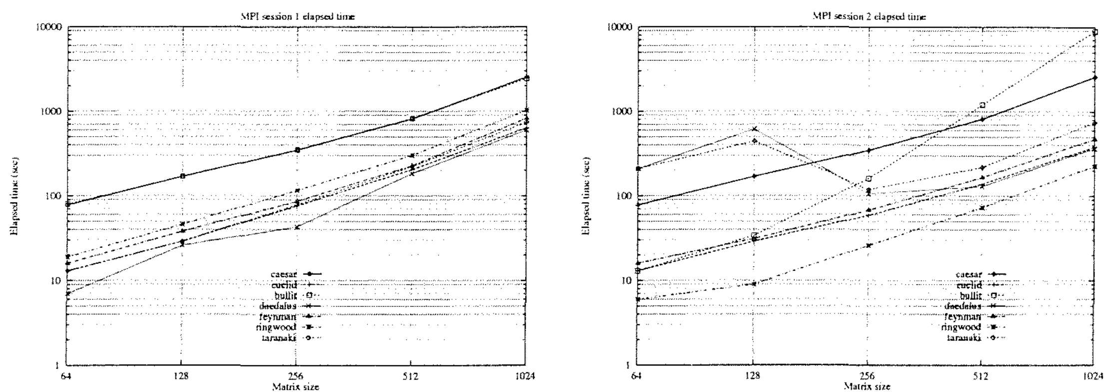

Figure C.4: The two MPI sessions

cache and RAM size[1], clock speed, etc. The sequential code was tested over many different hardware platforms with a wide range of results (from 465 sec to 4200 sec). The parallel MPI implementation does not have such wide variations (144 sec to 420 sec). Size scalability of the problem was tested as well. For example a test multiplication of two 2048 X 2048 size matrices run under 37 minutes for the parallel implementation while the sequential one failed on most of the available computers because of lack of resources. Slow workstations usually slow down the whole application, this means that faster workstations have to wait for the slow ones before they can carry on computation. The MPI start-up overhead is relatively large which means that there is no speed improvement for matrix sizes less than 512 X 512.

Figure C.4 shows the first MPI session (C.1) of the code, where processes "0" and "2" have created the two matrices and initial distribution of the sub-matrices to the other processes. In the second MPI-session of the program, processes start sending sub-matrices for the final calculation of the product matrix (equations. C.10-C.13).

[1]In all cases the RAM size was large enough to avoid any disk swapping.

Table C.2: Sub-matrices exchange among processes before optimisation

m m Proc. 0	Proc. 1	Proc. 2	Proc. 3	Proc. 4	Proc. 5	Proc. 6
S -> 6	S -> 3	S -> 4	S -> 6	S -> 6	S -> 0	R <– 0
R <– 1	S -> 0	S -> 0	R <– 1	R <– 2	–	R <– 3
R <– 2	–	–	–	–	–	R <– 4
R <– 5	–	–	–	–	–	–
9.78	3.79	89	3.68	4.74	2.26	7.53

C.4 Solutions and Suggestions

The main objective for this case study was to demonstrate that MPI is suitable and can be used successfully either in a homogeneous or heterogeneous cluster environment, without any modifications or special user privileges (as other similar message-passing packages require). An efficient utilisation in a heterogeneous environment has been demonstrated with actual results indicating that the existing heterogeneous cluster of workstations has a better performance than an expensive centralised computer system (e.g. compare the existing SGI and Solaris cluster and the HP9000 (1,5 GRAM, 4 nodes PA7020) in Figure C.3).

The underlying network has a significant effect on the performance of MPI in a cluster. Slow network interconnect and network congestion can severely affect node scalability and overall performance of the application. An SPMD program can cause a bottleneck when all the processes try to update or access the same resource simultaneously. Some overlap with the computation has to be introduced to avoid the extra cost of collisions and retransmissions. Potential bottlenecks in such environments can be caused by the underlying network and the node computational load. The role of the sustaining OS and the communication protocols should be considered as well in order to avoid superfluous underlying network layers overhead.

Rescheduling Communication A way to improve the MPI communication performance is to identify which MPI communication mode suits the specific application and the specific environment better, e.g. knowing the sequential nature of the Ethernet layer we can avoid congesting the Ethernet layer [150]. Initially the communication order among the modes for the second MPI section of the application is represented on Table C.2. Rearranging the communication order among processors we can minimise the amount of communication and consequently reduce the congestion problem on the Ethernet bus (e.g. by use of an Ethernet switch). As shown in Table C.3 there can be a significant reduction of the elapsed time in this phase.

An alternative to avoid network congestion can be the use of parallelism at the communication subsystem e.g. parallel communication architectures such as the Computer Network Architecture (CNA) [111, 208]. Switched networked clusters in a similar way can also alleviate most of the network congestion problems encountered during MPI session I and MPI session II of the algorithm by overlapping communication links among different pairs of nodes. Figures 8-9 illustrate the performance of those sessions on a switched Fast Ethernet cluster of fast

Table C.3: Optimised sub-matrices exchange

m m Proc. 0	Proc. 1	Proc. 2	Proc. 3	Proc. 4	Proc. 5	Proc. 6
S -> 5	S -> 3	–	R <- 1	S -> 6	R <- 0	R <- 4
–	–	S -> 5	S -> 6	–	R <- 2	R <- 3
S -> 6	S -> 5	S -> 4	–	R <- 2	R <- 1	R <- 0
2.5	2.5	2.37	2.2	2.28	3.21	2.86

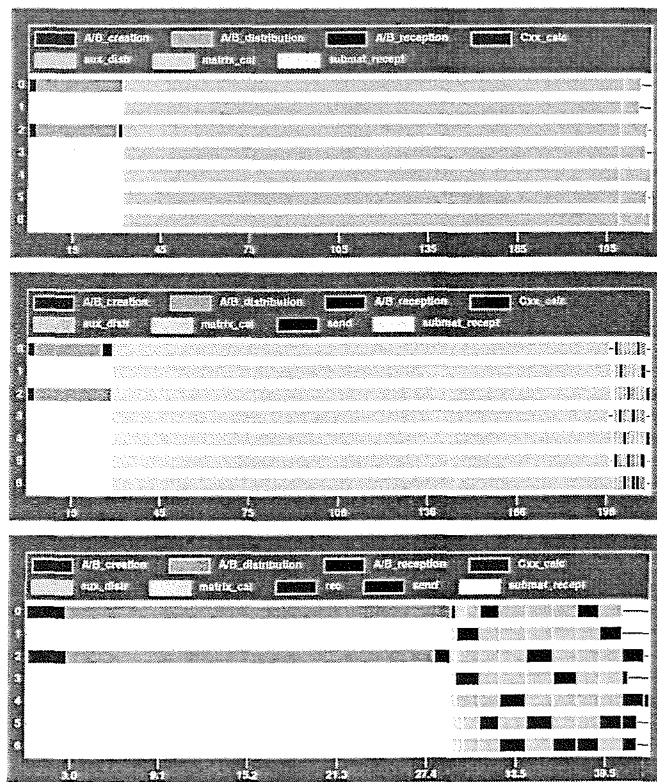

Figure C.5: Matrix multiplication: a total view upshot of the program (top), total view upshot of the optimised program (middle), detailed upshot of the optimised part of the program (bottom)

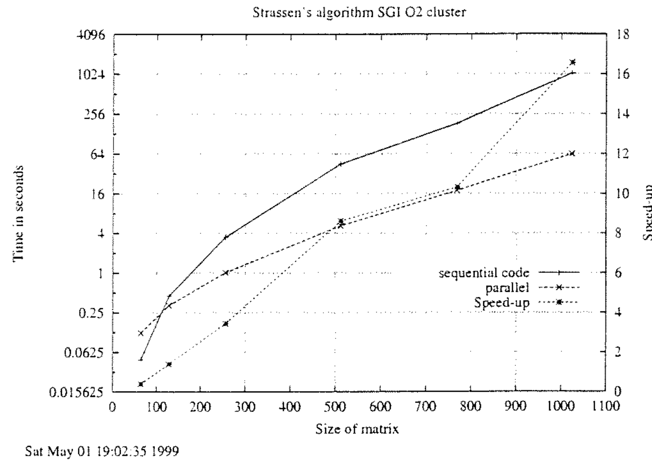

Figure C.6: Sequential algorithm versus. parallel Strassen's algorithm, and speed-up plots for the SGI O2 cluster

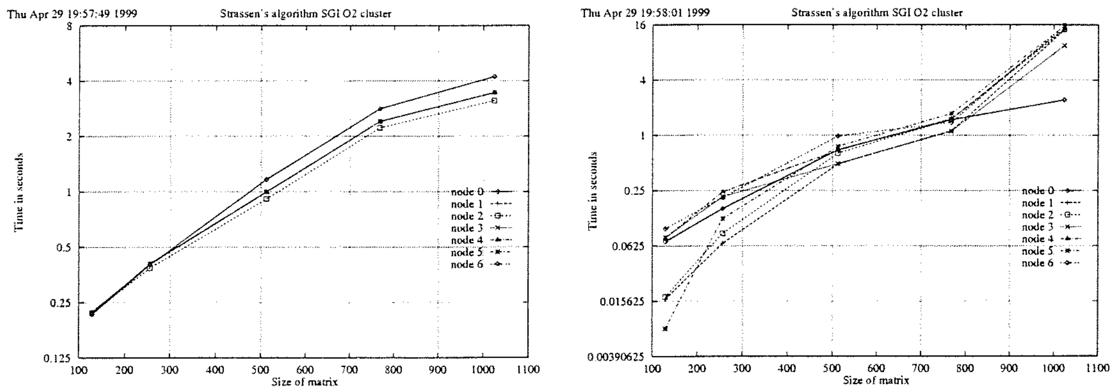

Figure C.7: MPI sessions I and II of the SGI O2 switched Fast Ethernet cluster

workstations (SGI O2). Although quantitative comparison with previous results is not possible, because both node and network performance of this cluster is significantly different, it is clear that the effect of multiple communication paths via the network switch provides a worthwhile improvement. Communication times are smaller and more evenly distributed among the processes.

Sometimes the non-deterministic network behaviour can cause problems especially for a heterogeneous environment spread among several LANs and buildings e.g. delays in transmission, performance degradation due to another network activities, or even worse workstations can go down at any time. A non-responding node has to be identified and isolated (or replaced) from the MPI environment. An S-MPI environment [60, 58] with a job scheduler and a task management could avoid many of these difficulties. Heterogeneous clusters require a different copy of MPICH installation for each different platform as well as different executables and MPI libraries. Any changes to the program source code require recompiling and updating of those

executables and other configuration files. The availability of distributed software tools for clusters could help considerably [60, 58]. The following list shows a possible directory structure for an MPI application to run in a heterogeneous environment of SGI and SUN machines:

```
~/mpi_prog/-+-program.c
            |-header.h
            |-Makefile
            |-IRIX-+-program.sgi
            |      |-program.sgi.o
            |      `-Makefile.sgi
            '-solaris-+-program.sun
                      |-program.sun.o
                      `-Makefile.sun
```

Appendix D

MPI-2 and Parallel I/O

Disk I/O is the slowest level of the memory hierarchy, excluding serial-access magnetic media e.g. tape drivers. Technological advances in storage devices have not improved the disk transfer I/O performance in the same way that disk capacity or CPU performance has improved over recent years. This disparity between CPU and I/O performance is a potential bottleneck, especially in HPC and parallel systems. More applications are now demanding enhanced I/O performance, i.e. database systems with large number of transactions, scientific applications with bulk data transfer, video applications, or real time interaction between computers and between computers and users [164].

Existing Distributed File Systems, such as NFS [193, 194] or AFS, are not adequate for HPC because they have been designed to run on Distributed systems and do not cope successfully with parallel applications or MPPs.

Internal parallel I/O subsystems were used successfully in MPPs such as Intel iPSC hypercubes, nCube Paragon XP/S CM-5 Meiko CS-2 SP1 SP2 Cray T3D, etc. Use of internal high performance switching networks is possible for parallel data transfer on multiple I/O nodes. Unfortunately for such operations there are no standards and hence portability was not preserved [72].

The semantics of a parallel file system are not the same as of a sequential one, however compatibility must frequently be preserved for many reasons. Concepts of parallel access modes, locally partitioning of subfiles, etc must be specified. The API's semantics have to change as well in order to adapt and exploit parallel I/O.

D.1 MPI I/O Concepts and Semantics

The approach of defining I/O access modes to express chosen common patterns of shared files such as collective data access is limited in its applicability. For this reason another radical way was chosen to access files in which data partitioning and data accessing among processes is expressed by derived datatypes. Selecting a datatype as the basis of partitioning a file among processes also provides additional advantages of flexibility and expressiveness.

An MPI file is an ordered collection of typed data items. Random or sequential access

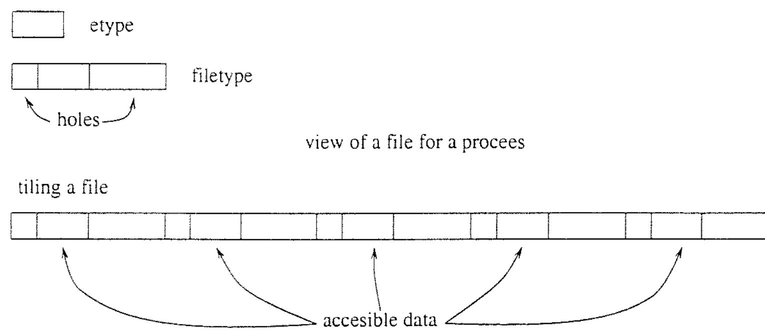

Figure D.1: An etype, a filetype, and a view of file for a process

to any integral set of these items is possible. Files are *created*, *opened*, *closed* and *deleted* collectively by a group of processes.

The **etype** or elementary datatype is the unit of data access and partitioning. *Etypes* can be any predefined or derived MPI datatype. Data access operations are performed in *etype* units, **offsets** are expressed as a count of *etypes*. *File pointers* point to the beginning of *etypes*. The absolute byte position from the beginning of a file is called *displacement*, it defines the location where a *view* begins, see Figure D.1.

Accordingly to the datatype, a **filetype** defines a template for accessing a file and is the basis for partitioning a file among processes. A *filetype* can be either a single etype or a derived MPI datatype constructed from multiple instances of the same *etype*. Any extend of a *hole* in a datatype must be a multiple of the *etype's* extend. Files are created by tiling of *filetypes* while file size is measured in bytes from the beginning of the file.

A **view** defines the current set of data visible and accessible from an open file as an ordered set of *etypes*. The *view* of a file is defined by three quantities: a *displacement*, an *etype* and a *filetype*. Each process has its own *view* of the file according to its *filetype*, each *view* is tiling from the *displacement*. The default *view* of a file is linear byte stream and can be changed by the user during **program** execution.

A position in the file relative to the current *view* is called **the offset**, it is expressed as a count of *etypes*. *Displacement* and *holes* of the *view filetype* are skipped. The end of the file is the *offset* of the first *etype* accessible in the current view starting after the last byte in the file (see Figure D.2) . File pointers are implicit local *offsets* to each process maintained by MPI. A *shared file pointer* is shared among all the group of processes that opened the file. Finally a *file handle* is an opaque object, which is used by all routines to operate on the file, it is created by MPI_FILE_OPEN and freed by MPI_FILE_CLOSE.

Sometimes it is useful if a user can provide information on the access patterns for a file and file streams which optimise I/O performance. MPI provides the FILE_INFO mechanism in which information can be passed to an *info* object.

There are a limited number of MPI I/O implementations that have been developed to date. An example is ROMIO Version 1.0.0, released in October 1997. It is a high-performance, portable implementation of MPI-IO that supports a wide range of hardware platforms and filing systems (NFS, PIOFS, UFS, etc). This implementation based its portability on an internal

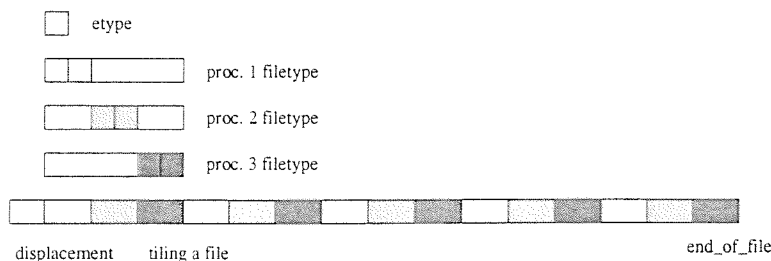

Figure D.2: Tiling a file with filetypes of three processes

abstract I/O device layer called ADIO. This version of ROMIO includes everything defined in the MPI-2 I/O chapter except file info, shared file pointer functions, split collective data access routines, support for file inter-operability, I/O error handling.

PMPIO is a preliminary implementation of MPI I/O from NAS. It supports MPICH implementations providing full support for arbitrary filetypes, collective I/O operations, and support for info objects cb_nodes and cb_buffer_size. This beta release does not include asynchronous I/O, shared file pointers or support for files stored on NFS file systems.

File Inter-operability

File inter-operability is the ability to read the information previously written to a data file, not just the bits of the file but the actual information the bits represent in a data file. File inter-operability is specified in the MPI_FILE_OPEN call. MPI supports the conversion of transferring the bits of a file into and out of the MPI environment between different machine representations, using three data representations:

native No data type conversions are performed during read or write operations.

internal The implementation will perform type conversions if necessary and thus supports heterogeneous environments.

external32 Each read or write operation converts all data from and to the "external32" representation which is defined by MPI-2 and based in big-endian IEEE format.

A problem may arise when handling data representations that are unknown for the implementation, therefore a user defined data representation is introduced that inserts a third party **filter** into the I/O stream to do data representation conversion.

In general, using the same data representation name when writing and reading a file does not guarantee that the representation is compatible between two different implementations. Instead *external32* representation guarantees compatibility.

D.2 MPI-I/O Data Partitioning

As we can see MPI I/O attempts to maintain similar semantics for accessing data in files as the MPI communication functions. Hence in MPI_FILE calls the format of data part argument has

the known order of *buf*, *count*, *datatype*. Restrictions of the type signature matching, number of continuation copies, overlapping regions etc, are similarly preserved in I/O as well.

MPI derived types are used to describe how data is laid out in the user buffer. The same aspect is extended to describe how the data is laid out in the file as well.

Thus we distinguish two derived datatypes in MPI-2:

filetype describing the layout in a file

buftype describing the layout in the user buffer.

Filetype and *buftype* are derived by a third MPI datatype referred to, as the *elementary* datatype or **etype**.

Offsets for accessing data are expressed as an integral number of *etype* items.

The *filetype* defines a data pattern that is replicated through the file to tile the file with data. MPI derived datatypes consist of fields of data that are located at specified offsets, (use of displacement and offsets can leave "holes" between the *primary* datatype fields, which do not contain any data[1]). A process can access the file data that matched items are in its access filetype (but not data that falls under holes, see Figure D.2).

Data which reside in holes can be accessed by other processes which use complementary filetypes. MPI-I/O provides filetype constructors to help the user to create complementary filetypes.

The use of filetypes allows a certain access pattern to be established MPI-I/O defines a displacement from the beginning of the file and the access pattern starts from that displacement (header information can be stored there).

D.3 MPI-I/O Data Access Functions

MPI-I/O defines three orthogonal mechanisms of data access:

positioning: explicit offset vs. implicit file pointer

synchronisation: blocking vs. nonblocking

coordination: independent vs. collective

MPI provides all combinations of these data access functions, including two types of file pointers, **individual** and **shared**.

D.4 Positioning

Unlike UNIX file systems, a parallel environment must decide whether a file pointer is shared by multiple processes or alternative each process is to maintain its own individual file pointer. Parallel programs do not generally exhibit locality to the reference within a file often move

[1] i.e. For a specific architecture, doubles should be strictly aligned at addresses that are multiples of 8, a datatype of{(double,0),(char,8)} would rounded to 16 with a hole of 7 bytes

between distinct non-contiguous regions of a file. Thus for each read or write operation a seek operation is almost always necessary. Multi-threaded or asynchronous I/O extend that need even further. Therefore MPI-I/O provides functions for positioning :

Explicit offset operation: the user specifies the offset (act as atomic seek-read/write operations)

Individualandshared file pointer operations use the implicit system maintained offsets for positioning.

The different positioning methods are *orthogonal*, in the sense that they can be mixed in the same program without affecting each other.

Explicit Offsets

MPI-I/O uses two "keys" to describe locations in a file: an MPI *datatype* and an *offset*, the first one is used as a template, and an *offset* which determines an initial position for transfers. Offsets are expressed as an integral number of *etype* items relative to the *filetype*. Any holes a filetype has are ignored and do not count as etype items for the offset.

File Pointers

When a file is opened in MPI-I/O the system creates a set of file pointers to keep track of the current file position. There is a global file pointer shared by all processes in the communicator group (processes should use the same filetype) and there are individual file pointers local to each process.

Each I/O operation leaves the file pointer pointing to the next data item after the last one that was accessed:

$$new\, file position = old position + \frac{size(datatype) \times count}{size(etype)} \tag{D.1}$$

where *count* is the number of elements of type *datatype* to be accessed and where size(datatype) gives the number of bytes of actual data that composes the MPI datatype *datatype*. For both performance and thread safety reasons MPI always updates the file pointer at the outset of an operation by the amount of data requested.

D.5 Synchronisation

The MÌ-2 standard supports explicit overlap of computation with I/O, through the use of nonblocking I/O functions.

- A blocking I/O will block until the I/O request is completed.
- A non-blocking I/O call initiates an I/O operation, but does not wait for it to complete. A separate request complete call (MPI_WAIT, MPI_TEST) is needed to complete the I/O request.

D.5.1 Coordination

Global data accesses have significant potential for automatic optimisation. Every non-collective data access routine has a collective counterpart. Independent calls do not imply any coordination among processes, and may be executed individually by any process within a communicator group.

Collective I/O requests are executed by all processes within a communicator group. A process can return from a collective call as soon as its participation in the collective operation is completed. Note that this return does not indicate that other processes have completed or even started the I/O operation.

End of File

Unlike Unix files, the end of file is not absolute and identical for all processes accessing the file. When a file grows, because of more data being written to it or the file being resized, the end of the file of all processes accessing the file may change, data now are accessible, but not yet written to the file, will be read as zeroes.

D.6 Collective Operations

Collective operations in order to access a file use the shared file pointer in the order determined by the ranks of the processes within the group. Calls return only after all the processes within the group have initiated their accesses. Implementation of collective calls can be used independently for each process hence it can be carried in parallel if possible.

D.6.1 Consistency Semantics

They define the outcome of multiple access to single file, all file accesses being are related to a specific file handle created from a collective open. MPI-2 provides three levels of consistency:

- Sequential among all accesses with a use of a single file handle
- Sequential among all accesses using file handles created from a single collective open
- Weak consistency among all accesses not handled with a use of synchronisation mechanism

The default semantics for overlapping accesses does not guarantee sequential consistency (non-atomic mode). In this mode all data in regions of the file which had overlapping accesses is undefined, unless weak consistency is enabled. Atomic access can be guaranteed for overlapping accesses by enabling atomic mode routines. Overlapping accesses are not by definition consistent.

Appendix E

Parallel I/O Tests

Technological advances in disk capacity and the improvement in CPU performance achieved over the last few years have not been matched by similar increases in disk bandwidth. This disparity inevitably will become a potential bottleneck for HPC applications. The use of parallel I/O provides a straightforward solution, although the implementation is still not easy. The proposed MPI-2 standard addresses this problem using existing file systems without major changes, thus preserving portability while still providing usable performance. Measurements of performance end evaluation of HPC I/O systems can be undertaken using the parallel I/O benchmark mechanisms reported in this paper.

This benchmark was originally based on the *bonnie* benchmark written by Tim Bray and subsequently extended at the University of Southampton. It consists of a suite of three benchmark programs: The *write* benchmark tests and measures the MPI_write performance, while the *read* benchmark tests and measures the performance of MPI_read, and the *rewrite* benchmark tests and measures the I/O performance of MPI.

E.1 Test Conditions

The tests were run on an Ethernet network cluster of 12 SUN SPARC-4 workstations running Solaris 2.5.1. During the experiments the network had no additional traffic, apart the overhead of the Operating System, and the NFS file system.

The tests were run for different file sizes and different block sizes. The size of file is restricted to 32 Mbyte maximum because of an internally imposed disk quota. Some of the tests create a file on the master's node temporary directory (*/tmp/test.file*).

The tests ware run for different sizes of file and different block sizes. The size of the file is restricted to 32 Mbyte maximum because there is a disk quota limit.

E.2 Writing to the file test

The objective of this program is to measure the *MPI_write()* function bandwidth. The test creates a test file and writes to it repeatedly a buffer of known block size. The use of blocking

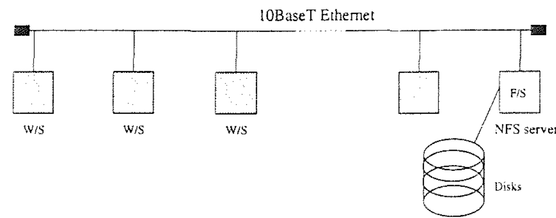

Figure E.1: The interconnection of the cluster

non-collective file operations ensures that all the outstanding requests associated with the file have completed before the process closes the file.

```
open a file
get wall time
for the number of sizeof(file)/sizeof(block)
  dirty the buffer
  calculate the offset
  write the buffer to the file
close the file
get wall time
```

E.3 Reading from the file test

This test is the opposite of the *write* program, it measures the *MPI_Read()* function bandwidth, by opening the already-written test file and reading it, in blocks of known size. The test is timed until it closes the file *MPI_Close()*.

```
open a file
get wall time
for the number of sizeof(file)/sizeof(block)
   calculate the offset
   read from the file a block
   buffer[random]++ (fool the compiler)
close the file
get wall time
```

E.4 Rewriting a file

This test reads a block from the test file, modifies the block and then writes it back to the file. The purpose of the test is to measure the effectiveness of the filesystem, the cache, and the data transfer rate. The main loop has additional commands in order to ensure that the compiler does not optimize the operation specified.

```
open a file
```

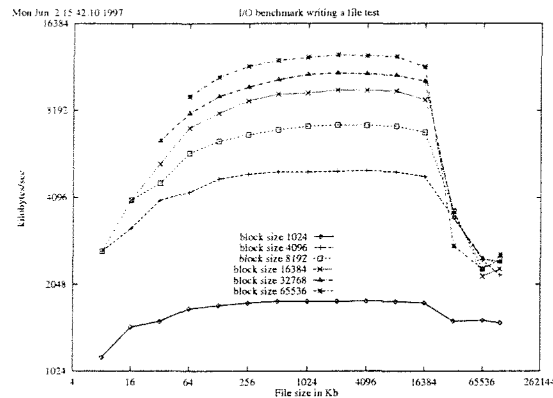

Figure E.2: Parallel I/O for one node, writing to a file

```
get wall time
set the offset
read the first block
while the file is not read
   dirty the buffer
   write the buffer back to the file
   calculate the nest offset
   read the next block from the file
close the file
get wall time
```

E.5 Experimental results, tests

Running tests in practice: There is a crucial test file size that is just larger than the maximum size of I/O cache the system can allocate. Usually this size is estimated to be around 10 Mbyte less than the system's RAM size. For sizes less than this crucial size the system appears to respond faster than its real I/O thus producing a misleading response.

E.5.1 Running the benchmark for one node

The parallel I/O benchmark was run on one node for a single process. The local attached file system was used (*/tmp*) for the test file.

The results are consistent with the *bonnie* benchmark, although there is a slight degradation of the throughput due to the extra overhead of the MPI calls. Increasing the buffer size provide a considerable speed-up in all I/O operations.

E.5.2 Running parallel I/O benchmark for two nodes

This test runs on two nodes with two processes. The shared file system is NFS with a theoretical maximum throughput of less than 1.28 Mbyte/sec. The impact of the interconnection network,

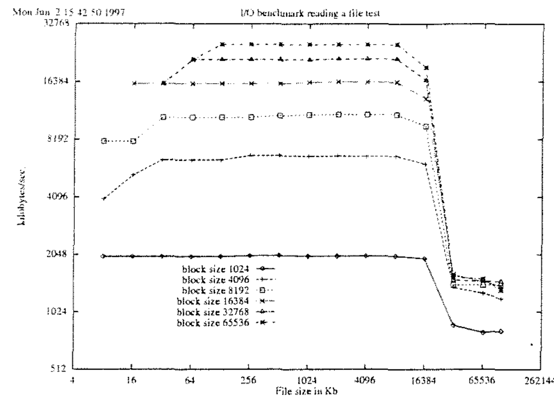

Figure E.3: Parallel I/O node reading a file for one

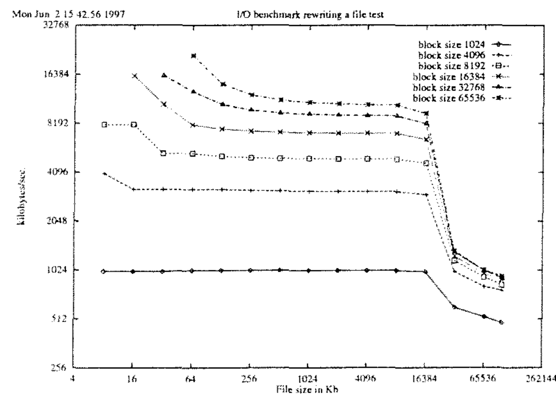

Figure E.4: Parallel I/O for one node, rewriting a file (throughput)

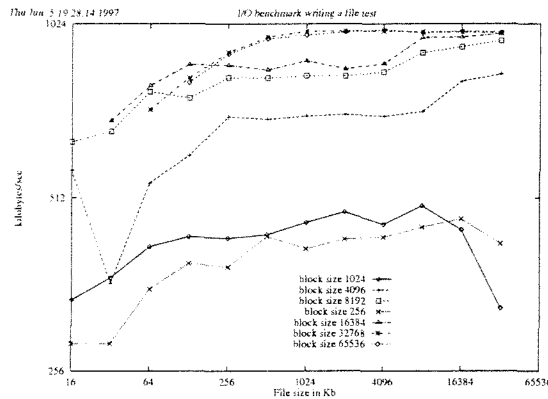

Figure E.5: Writing a file test on two nodes

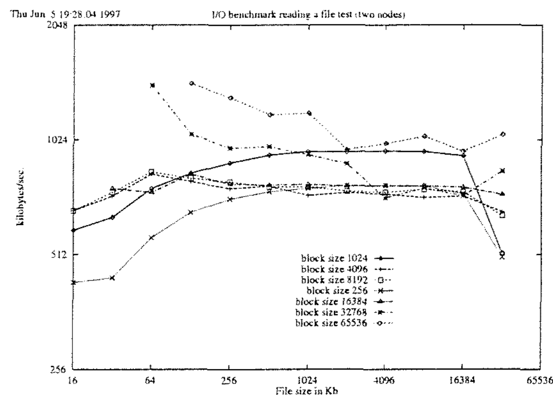

Figure E.6: Reading a file test on two nodes

Ethernet, affects the disk buffer cache and is responsible for the unpredictable behavior of the system as well.

E.5.3 Running parallel I/O benchmark for four nodes

The communication overhead, of the test nodes introduces congestion into the communication channel, affecting the shared I/O subsystem and decreasing performance slightly.

E.6 Summary

Internode connections and the I/O subsystem attachment are major factors in determining the performance characteristics of a parallel system. The parallel I/O benchmark described in this note provides useful information on the I/O performance that can be expected for applications software. Future work will include measurements on alternative parallel systems (e.g. SMP and MPP) and also incorporate additional tests.

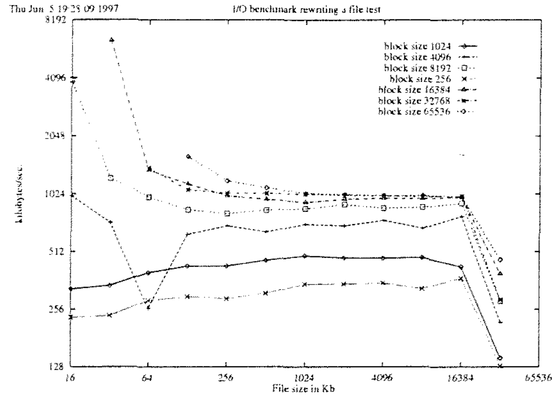

Figure E.7: Rewriting a file on two nodes

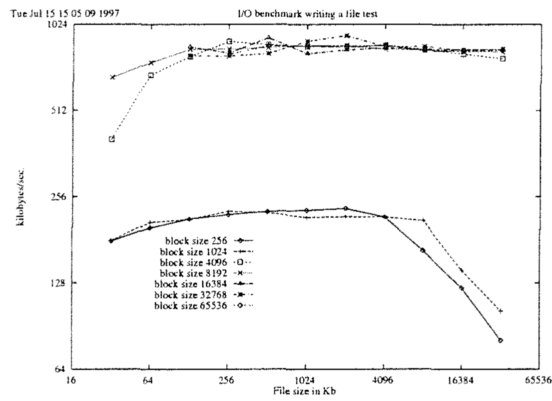

Figure E.8: Writing to a file over 4 nodes

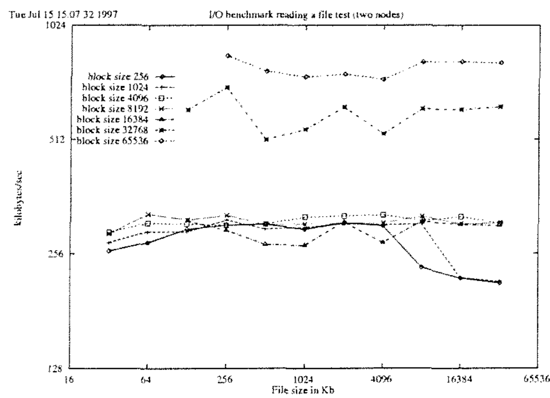

Figure E.9: Reading from a file over 4 nodes

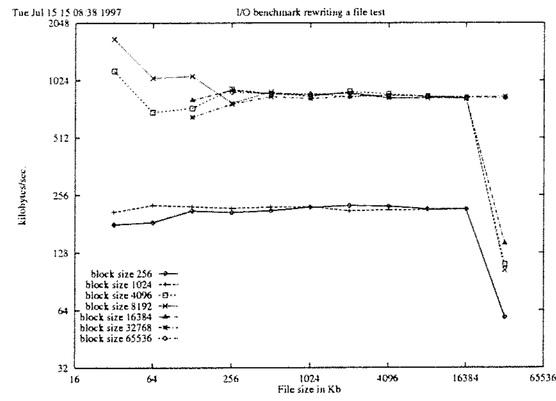

Figure E.10: Rewriting to a file over 4 nodes

Appendix F

Kernel-level Algorithmic Tests

F.1 Row/Column Striped Algorithm

The implementation of the Row-column-Oriented algorithm involves the following phases:

Initialisation phase: During this phase matrices A and B of size $N \times N$ are initialised and filled with random numbers. A virtual two-dimension topology of nodes is defined and groups of communicators for the number of rows and columns are initialised together with user-specified block vectors needed in later stages of the algorithm.

Phase one: A block-striped partitioning of A and B matrices takes place. Matrix A is partitioned horizontally and distributed over the first logical column of processors, then each processor of that column broadcasts its sub-matrix along its own logical row. In a similar way matrix B is partitioned vertically and distributed over the first logical row of processors, followed by a broadcast operation along the columns of that row.

$$T_{scatter} = col(t_s + t_w \frac{N^2}{col}) + row(t_s + t_w \frac{N^2}{row}) = (col + row)(t_s + t_w N^2)$$

where t_s is the startup time and t_w the cost per byte transmission. In a similar way the cost of the broadcast call will be:

$$T_{bcast} = \lceil \log_2 col \rceil \, (t_s + t_w \frac{N^2}{col}) + \lceil \log_2 row \rceil \, (t_s + t_w \frac{N^2}{row})$$

Phase two: The computation cost for each process with an $N/p \times N/p$ matrix-size problem is given by:

$$T_{comp} = t_c \cdot \frac{N^3}{p}$$

where t_c is the averaged cost for each iteration of the multiplication operation.

Phase three: During the last phase processes have to use an *MPI_ Gather*() call to gather the product result matrix to a master node. The cost of this operation will be:

$$T_{gather} = p(t_s + t_w \frac{N^2}{p}) = pt_s + t_w N^2$$

Accordingly the total communication time will be the sum of the three major communication operations: scatter for the sub-matrix distribution, broadcast of sub-matrices along rows and columns and finally the local computation takes place to gather all the product sub-matrices to the master node:

$$T_{comm} = T_{scatter} + T_{bcast} + T_{gather}$$

For the shake of simplicity when the processor grid is $\sqrt{p} \times \sqrt{p}$ the total execution time becomes:

$$T_{tot} = T_{init} + t_c \frac{N^3}{p} + 2(\sqrt{p} + \lceil \log_2 \sqrt{p} \rceil)(t_s + t_w \frac{N^2}{\sqrt{p}}) + p(t_s + t_w \frac{N^2}{p})$$

$$T_{tot} = T_{init} + t_c \frac{N^3}{p} + t_s(2(\sqrt{p} + \lceil \log_2 \sqrt{p} \rceil) + p) + t_w(\frac{2N^2}{\sqrt{p}}(\sqrt{p} + \lceil \log_2 \sqrt{p} \rceil + \frac{1}{\sqrt{p}}))$$

F.2 Cannon's Algorithm

The implementation of algorithm has the following phases:

Initialisation phase: This phase is very similar to the previous algorithm.

Phase one: The checkerboard partitioning of matrices A and B among p processors into blocks of $(N/\sqrt{p}) \times (N/\sqrt{p})$ size takes place here as well as the initial alignment of blocks at the same time. The two MPI calls for this operation are for the *MPI_Scatter()* routine:

$$T_{scatter} = 2 \times (p \cdot (t_s + t_w \frac{N^2}{p}))$$

Phase two: In this phase there is a loop of $\sqrt{p} - 1$ iterations, in each iteration there is a computation part of an accumulated sub-matrix multiplication as well as a shift operation among matrices of the same row/column. The computation time is given accordingly:

$$T_{comp} = t_c \cdot \frac{N^3}{p}$$

During the shift operation sub-matrices of the four neighboring processors are exchanged so the total cost of the "shift" operation is given by:

$$T_{shift} = 4 \times (\sqrt{p} \cdot (t_s + t_w \frac{N^2}{p}))$$

Phase three: At the end of phase two processes have sub-matrices of the product result matrix. A gather operation again is used in this phase to congregate the product matrix on a master node.

$$T_{gather} = p(t_s + t_w \frac{N^2}{p})$$

The total communication cost is the sum of the partial times required for the initial matrix decomposition (the scatter operation), the shifting and gathering of the final sub-matrices:

$$T_{comm} = T_{scatter} + (\sqrt{p} - 1)T_{shift} + T_{gather}$$

At the root node the idling time is regarded as zero therefore the overall time is given by:

$$T_{tot} = t_c \frac{N^3}{p} + (7p - 4\sqrt{p}) \cdot (t_s + t_w \frac{N^2}{p})$$

$$T_{tot} = t_c \frac{N^3}{p} + t_s(7p - 4\sqrt{p}) + t_w \frac{N^2}{p}(7p - 4\sqrt{p})$$

The two different algorithms of matrix multiplication are implemented and compared using MPI and run over a cluster of SGI (O2) workstations.

F.3 Sorting Routine

The implementation of the PSRS algorithm involves the following steps throughout its phases:

$$T_{tot} = T_{init} + T_{comp} + T_{comm} + T_{idle}$$

Initialisation phase: A list of n elements is created and filled with pseudo-random integers by the master process. The values are evenly distributed in the range 0 through $2^{32} - 1$. Initialisation of various vectors, user-defined datatypes and space allocation takes place as well during this phase for use later in the algorithm.

Phase one: The root node distributes data (n/p) evenly among nodes via an *MPI_Scatter()* call. Then each node is running a *qsort()* algorithm to sort its local list of data $O(n/p \log n)$.

$$T_{scatter} = t_s + t_w \frac{N}{p}$$

$$T_{qsort} = O(n/p \log n)$$

Phase two: An MPI_Gather() operation selects and sends p samples from each process to the master process. The master process then has to sort the sample list and to broadcast *p-1* values to all nodes (*MPI_Bcast()*).

$$T_{gather} = p(t_s + t_w p)$$

$$T_{bcast} = t_s + t_w(p - 1)$$

$$T_{qsort} = O(p^2 \log p)$$

Phase three: All processes have to partition their sub-lists according to the broadcasted pivot values. The exchange of *p-1* list partitions amongst all nodes is dynamic and based on an *MPI_Alltoallv()* call, the implementation of this step is more complex and requires the definition of various displacements and extra memory space allocations on the fly. The main *MPI_Alltoallv()* call is expecting to exchange at its worst case n element messages in total among its nodes simultaneously.

$$T_{all} = p^2(t_s + t_w)$$

$$T_{allv} = p^2(t_s + t_w \frac{N}{p})$$

Phase four: Each node uses a *qsort*() function to sort-out the re-arranged lists and an MPI_-Gatherv() collective function to restore the complete sorted list on the master node.

$$T_{gather} = p(t_s + t_w)$$

$$T_{gatherv} = p(t_s + t_w \frac{N}{p})$$

F.4 Multigrid Relaxation Routine

The implementation of the multigrid routine follows a conservative approach including the following phases:

Initialisation phase: In this phase various declaration of user-defined MPI datatypes take place as well as a virtual processor grid $(n/\sqrt{p}) \times (n/\sqrt{p})$ to map the problem.

Phase one: Checkerboard partitioning of the two-dimensional grid amongst the processor grid is performed. An MPI_Scatter() call with the appropriate vectors is used:

$$T_{scatter} = p \cdot (t_s + t_w \frac{N^2}{p})$$

Phase two: This phase incorporates the main loop of the algorithm. Each iteration scans each point of the local grid and calculates its new value and the maximum deviation.

$$T_{comp} = t_{cal} \cdot \frac{N^2}{p}$$

After that a combination of *MPI_Reduce*() and *MPI_Bcast*() operation informs all processes about the maximum deviation change. If that change is not small enough neighbouring processes exchange boundary values and repeat the iteration all over again. For benchmarking purposes the number of iterations is fixed in number controlled by a COUNTER variable. The cost of the reduce and the broadcast operation is relatively small as they operate on a single value only and can be simplified to:

$$T_{red+bcast} = p(t_{red} + t_{bc})$$

During the exchange of neighbouring process boundary values a combination of Send-Receive MPI calls can be used, the cost of these exchanges can be approximated by:

$$T_{exch} = 4 \times (t_s + t_w \frac{N}{p})$$

Phase three: At the end of phase two processes already have sub-matrices of the approximated solution array. A gather operation is used again in this phase to congregate the final multigrid solution on a master node.

$$T_{gather} = p(t_s + t_w \frac{N^2}{p})$$

The total communication cost is the sum of the partial times required for the initial matrix decomposition (the scatter operation), the shifting and gathering of the final sub-matrices:

$$T_{comm} = T_{scatter} + T_{red+bcast} + T_{exch} + T_{gather}$$

Appendix G

Modified Algorithmic-level Test Results

This appendix illustrates test results from the matrix-to-matrix algorithmic test which has been modified.

The modification transposes the second matrix which takes place during the decomposition phase of the algorithm using a modified memory access pattern. The result is a complete transparent matrix transpose in terms of both time and resources (e.g. extra memory). The complexity of the new algorithm main loop is still the same $O(N^3)$ and the real benefit derives from the improved access locality.

$$c_{i,j} = \sum_{k=0}^{m-1} a_{i,k} \cdot b_{j,k} \tag{G.1}$$

This modification can be equally applied to both matrix-to-matrix algorithms used, the Row/Column striped algorithm and Cannon's algorithm. The following figures show the results from tests run on the SGI cluster. Performance improvements over the original algorithm for the SGI cluster varies according to the matrix size between 100%-50%. These results demonstrate the importance of a tailored algorithm for a specific platform.

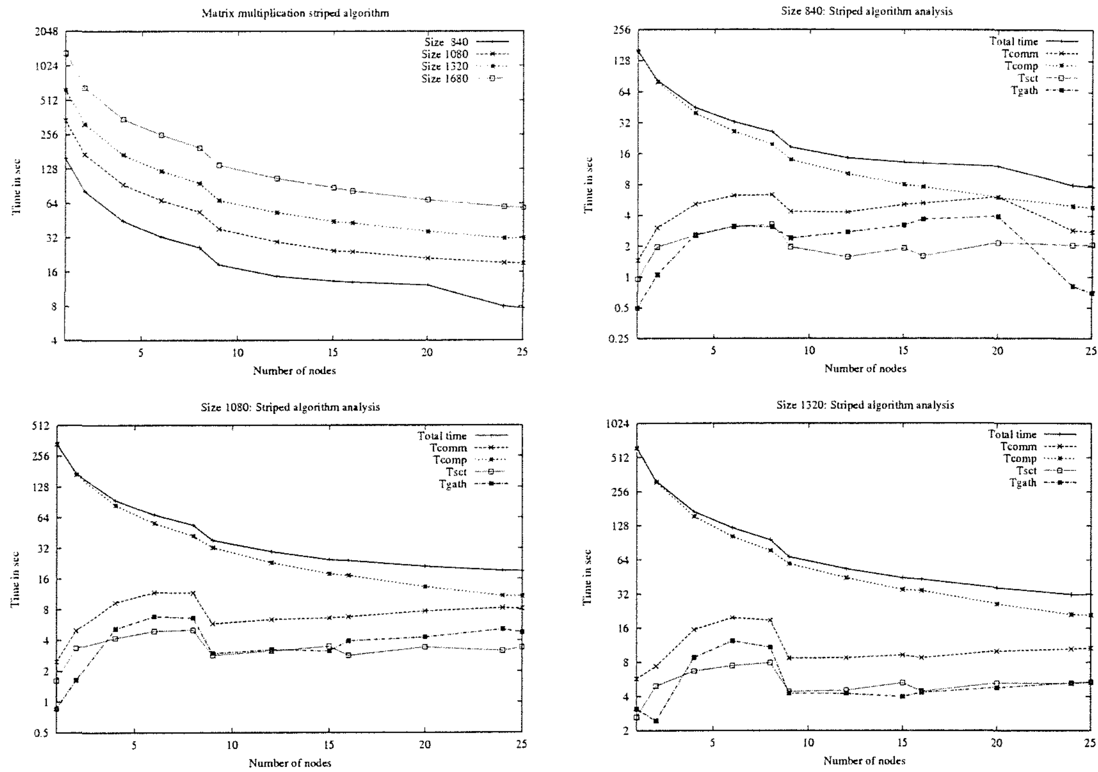

Figure G.1: Modified striped algorithm (first part)

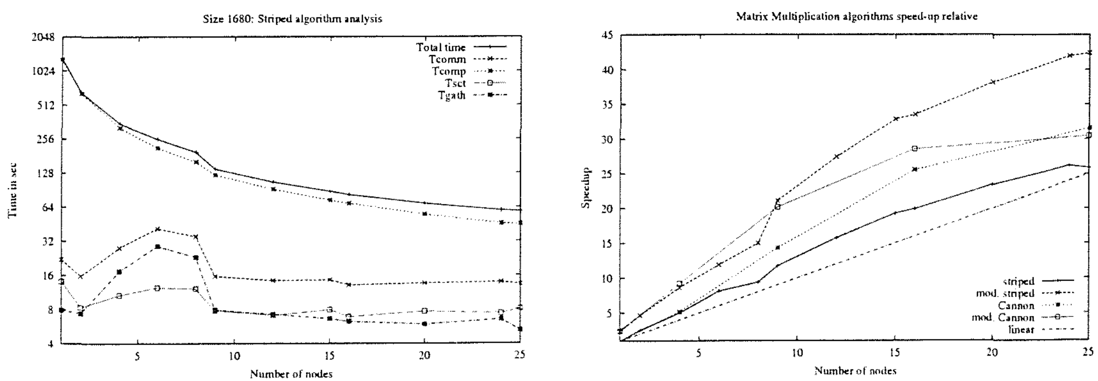

Figure G.2: Modified striped algorithm (second part)

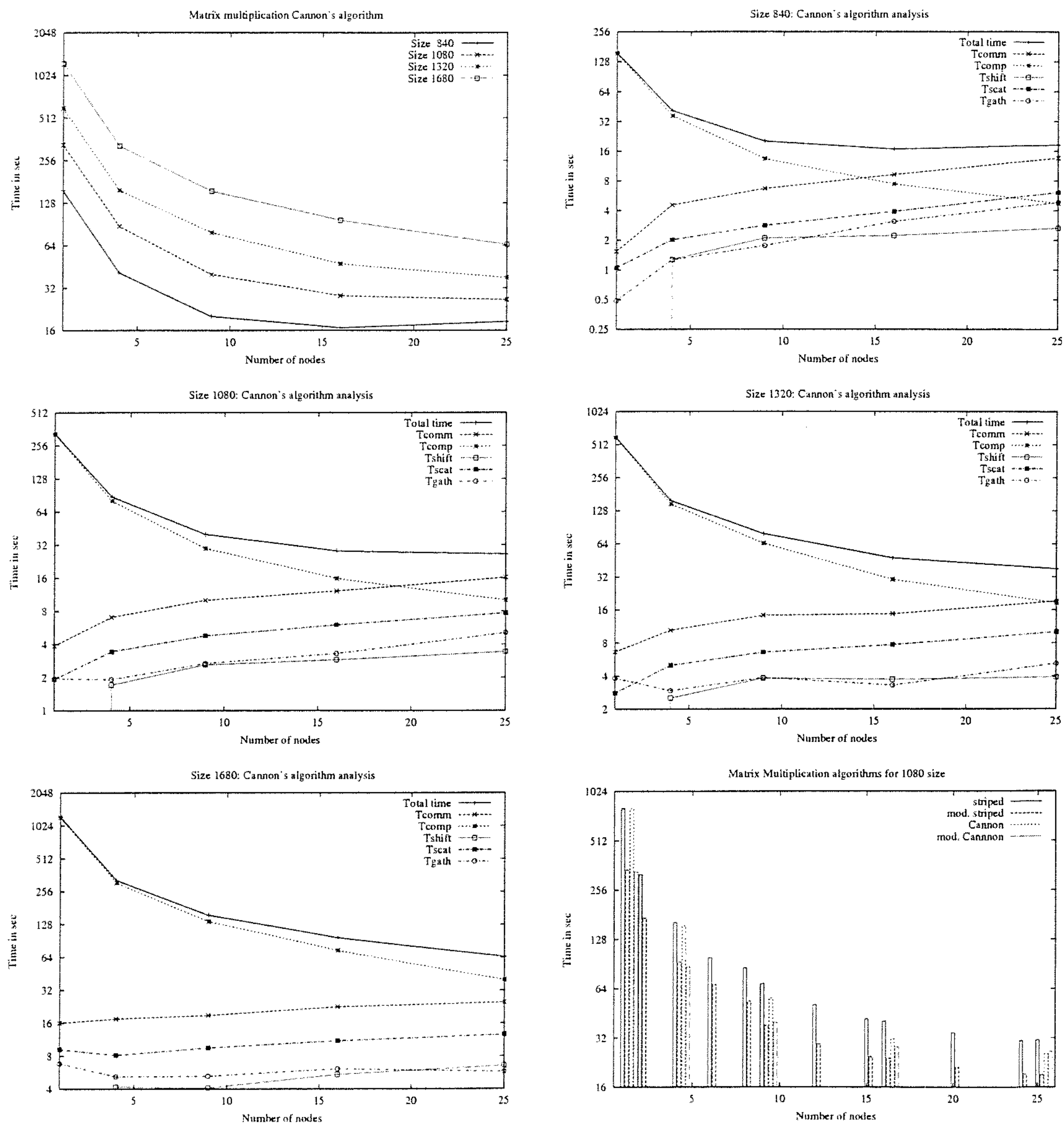

Figure G.3: Modified Cannon algorithm

Table G.1: Performance improvement for matrix-to-matrix algorithms using a transpose modification

Matrix size	Proc. grid	S/C	Mod. S/C	Impro-vement	Cannon	Mod. Cannon	Impro-vement
	1x1	344	157	2.19	346	160	2.17
	2x2	64.1	40.9	1.56	71.1	44.6	1.59
840	3x3	24.8	20.1	1.23	31.1	18.5	1.74
	4x4	17.8	16.7	1.06	19.5	13.0	1.50
	5x5	19	18.5	1.02	17.0	7.67	2.22
	1x1	805	330	2.43	805	340	2.37
	2x2	155	87.4	1.77	162	93.2	1.74
1080	3x3	56.2	39.9	1.41	68.7	38.1	1.80
	4x4	31.5	28.2	1.12	40.4	24.0	1.68
	5x5	25.5	26.4	0.96	31.1	19.0	1.64
	1x1	1494	599	2.49	1503	626	2.39
	2x2	303	157	1.92	319	170	1.88
1320	3x3	123	79	1.56	129	67.8	1.91
	4x4	59.2	47.5	1.25	74.9	42.9	1.74
	5x5	44.2	37.9	1.16	50.9	31.6	1.61
	1x1	3185	1238	2.57	3429	1315	2.61
	2x2	704	324	2.17	716	349	2.05
1680	3x3	274	155	1.77	284	138	2.06
	4x4	142	96.6	1.47	160	81.9	1.96
	5x5	90.6	64.9	1.39	104	58.5	1.78